An Agent Architecture for Structured Uncertain Environments

December 13, 2010

Thesis Proposal Review Zahra Zamani

1 Introduction

This thesis proposal investigates a solution in building cognitive agents able to operate in a realistic environment. Agents have been the subject of study in different fields raising questions about their behavior, structure, relations with other agents and the environment. The main trend in the programming area is to model agents based on the knowledge of a problem domain, rarely learning or adapting to a variety of environments. On the other hand, researchers in machine learning seek solutions using a rather simplified view of the environment while providing the least prior knowledge to the agent. Although this abstract model is easier to work with, it is further away from real world applications.

We are interested in building an agent that can use the models programmed by the designer. The agent is situated in an environment closely resembling the real world while trying to learn how to behave optimally.

An agent is defined as an individual entity or program designed to operate in specific environments with the aim to achieve some goal. The program running the agent can be fully determined by the designer or left on the abstract level for the agent to learn and adapt. The structure used to develop an agent is refered to as its architecture which determines the main components used to build the agent and their interaction.

In reality, agents try to achieve their goals although they are not confident about the effect of their actions. Most of the environment and other agent's behavior and goals are hidden to them so instead of knowing their exact position in the world, they build a belief on their current situation with some probability greater than zero and then choose their actions according to that probability.

The agent's knowledge about its surroundings is uncertain and although receiving input from the environment can help the agent obtain a better understanding of its current situation, the source of input can itself have some degree of noise. This knowledge can be expressed with various mathematical

or logical representations to the agent. In most areas of Artificial Intelligence (AI), this is simplified and translated into mathematical elements such as matrices,vectors or points. On the other side, in Software Engineering (SE) practices knowledge is more important and remains in logical formats such as terms, lists and tuples. While both areas focus on different subjects for different purposes, bringing together the learning and adapting nature of AI and the rich knowledge representation of SE is inevitable in order to design an agent to work in the real world. The agent surroundings can be considered as a structured environment and the knowledge will be in the form of structures of terms and relations among them. This will not only bring together the research of these two fields but will also use the expressive power of logics in the context of learning agents.

Most software systems concentrate on well known architectures such as the Belief- Desire- Intension (BDI) model, but here a different approach well known in the Robotics context but not explored in depth for cognitive agent application is used. We define an architecture based on motion and sensor models defined for tracking and localizing a robot in the real world. These models are refined by learning parameters to show environment characteristics. Once the models are ready the agent can use them to select actions to achieve its goal.

This document explains the core content of the thesis and reports some of the early results gained in the past year. In part 2 basic definitions and literature on agent frameworks is provided. Part 3 gives briefings on the combination of logic and probability, explaining the tracking models and learning issues involved. In part 4 the Trading Agent Competition (TAC) is defined with some previous solutions to the game. Our architecture and models are then used practically and results of some early simulations is presented. Some of the future work and conclusions are reported in the end.

2 Agent Framework

To propose a new agent architecture the basic concepts of an agent framework must be defined. The key abstraction in agent-based frameworks is that of an agent. A software agent is a computer program similar to functions in object-oriented programming with some level of abstraction situated in an environment to achieve an overall goal. Formally it is denoted by a function $\phi : History \rightarrow Action$ that maps the history of interactions of the agent and the environment and chooses an action as an output.[9] It describes a complex software system acting with some degree of autonomy on behalf of its users.

According to the degree of autonomy, agents can be fully pre-programmed or purely intelligent and autonomous. This spectrum is very wide, starting from system agents for specialized programs, reactive agents, event based agents such as search engines, personal/user agents that behave according to user needs and adapt their behavior, data mining agents as in shopping bots on the web and finally intelligent agents on the other end using learning approaches in order to achieve their goal. Physical robots are the body of a mobile agent equipped with certain autonomy levels based on their application.

The agent discussed here is considered as a system that takes percept sequences as input from some environment and applies actions to the environment. Our definition of a rational agent falls in the definition of a rational agent by [15]: "For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has". A rational agent has bounded resources, limited understanding and incomplete knowledge of what happens in the environment in which it lives.

According to the main literature in the agent framework, in order to define a fully operative framework three sections have to be specified.

- Agent Architecture: Representing the move from specification to implementation
- Agent language: Programming language that may embody the various principles proposed by theorists
- Agent implementation: After the theories of the programming language has been specified, it will need to be implemented.

Although the main focus is on designing an architecture, but using appropriate language and implementing the agent according to the language theories is essential.

2.1 Agent Architecture

An agent architecture provides the definition of the function φ mapping the percepts to actions, it is the blue print of how the components of a system should be connected and arranged. A rational agent adds the constraint of selecting an action so as to maximize an expected utility function such as the performance measure. The agent is surrounded by an environment modeled by the agent designer or explored by the agent. The environment can be partitioned into regions with similar properties that is refered to as the states.

Building an agent system can be fully mathematical based on strong logic or it can be determined based on experience through simulations. The complexity decreases as we approach the simulation end, however a balance is required to build upon a formal, but non-computable logical model, against ad-hoc implementation approaches with weak mathematical reasonings. Several platforms based on logical theories are reviewed here to show how they are formally described.

Agent platforms are mainly categorized by two main distinction systems. In the first type the platform mainly handles interoperability and various infrastructure topics, leaving open the issue of internal agent architecture. While the second type focuses on the behavior model of a single agent trying to achieve rationality and goal-directness. Most cognitive architectures are based on theories for describing behavior of individuals. The most influential theories with respect to agent technology are the theory of Agent Oriented Programming

(AOP) [16], the subsumption theory[4], and the Belief-Desire- Intention (BDI) model[2] presented next.

2.1.1 AOP

Agent oriented programming addresses the need for software systems to exhibit rational, human-like behavior in their problem domains. An Agent Oriented model, the same as Object Oriented programming, is a scalable development by encapsulating behavior in modular units containing all structures required to operate.

The agent has modalities defined as a mental state extended from epistemic logic. This leads to operators for obligation, decision and capability. At any time the future will depend on the past history and the current action. All the components and operators for the agent state and actions are defined using modal logic. A generic agent interpreter is designed for this model that updates the mental state of the agent according to the messages the agent receives and acts according to the current belief.

2.1.2 Subsumption

Subsumption architecture is a reactive robot architecture associated with behavior-based robotics. It offers is a way of decomposing complicated intelligent behavior into simple behavior modules described as a layer. Each layer implements a particular goal of the agent, and higher layers are increasingly abstract subsuming the underlying layers.

Although this architecture offers high modularity and iterative builds of the layers to achieve sub-goals, however adding many layers might cause conflicts with the previous layers. Action selection is also complicated due to the suppression and inhibition of layers.

2.1.3 BDI

Many frameworks have been proposed based on the concept of mental state. These frameworks are mainly based on the well-known BDI agent architecture. The main concepts of this architecture are beliefs, desires and intentions. Beliefs are informational attitudes of an agent representing the information agent has about the world it inhabits and about its own internal state.

The motivational attitudes of the agent are captured in desires that represent the agent's wishes and drive the course of its actions. Plans are the means by which agent achieves goals and react to occurring events. When an agent decides on pursuing a goal with a certain plan, it defines an intention towards the sequence of plan actions.

Most software agent systems follow the basic framework of the BDI architecture and build models upon that. 3APL[6], Jadex[3], Impact[7] and Jack[8] are some of the frameworks based on the BDI framework. The framework presented

here will be compared with this model so a short overview of each of them is provided to give more insight on this framework.

A 3APL [6] multi-agent system consists of a set of concurrently executed 3APL agents that can interact with each other directly or through a shared environment. In the single agent programming language both mental attitudes and deliberation processes (instructions) are considered. Each agent has mental attributes that consists of

- belief base: describing the situation the agent is in,
- goal base: describing the situation the agent wants to realize,
- plan base; a way of reaching goals composed of a sequence of actions,
- action base; decomposed into mental, communication, external and test actions along with abstract plans,
- goal planning base; used to generate plans to achieve goals conditioned by beliefs
- plan revision base; used to revise plans from the plan base also conditioned by beliefs.

  The deliberation cycle works like an interpreter and performs tasks such as checking the relation between plans and goals, removing garbage plans and verifying goal existence.

Jadex [3] is another framework based on the BDI architecture. While the agent platform is concerned with external issues such as communication and agent management, the reasoning engine covers the agent internal issues. Jadex does not use a logic-based representation of beliefs, instead ordinary Java objects of any kind can be contained in the belief base. Objects are stored as beliefs or belief sets. The engine monitors the beliefs for relevant changes, and automatically adjusts goals and plans accordingly. For any goal an agent performs actions until the goal is reached, unreachable, or not desired any more. Jadex uses the plan-library approach to represent the plans of an agent that defines the circumstances under which the plan may be selected and the actions to be executed.

The Impact framework [7] was initially built for controlling flight operations in the army. The agent language can have temporal or probabilistic reasoning as well as mental attributes. In this framework meta agents allow agents to reason about beliefs and actions of other agents, temporal agents allow agents to execute actions that have temporal extent and to schedule actions for the future and probabilistic agents allow the agent to reason about uncertainty in the world. Actions in this framework are performed from an action base and the reasoning is via a set of preconditions and effects defining the conditions a state must satisfy for the action to be considered executable, and the new state that results from such an execution.

JACK [8] is described as a research agent technology with integrated logic used for the needs of software engineers. It is an environment for building multi-agent systems based on the BDI architecture. In JACK each agent is defined in terms of its goals, knowledge and social capability. Each agent has to respond to a set of goals or events that rise in the system according to its set of plans. In addition to the BDI extension, JACK provides an extension to support team oriented programming, called simple-team that allows classification of team members in terms of abstract roles in order to provide a software infrastructure for the specification of coordinated behavior. It also uses the modular structuring elements called capabilities that represent a cluster of functional components that are considered as programming building blocks for reuse in the applications. The benefits of this architecture is the use of Java as its execution environment which enables integration with other systems, better performance and ease of using BDI characteristics.

This section provided definitions on the agent architecture and some of the most famous agent platforms in the agent community. The architecture used throughout this work is explained next.

2.1.4 Tracking and Acting Architecture

The proposed agent architecture is different from that explained in most artificial intelligent problems. The environments surrounding the agent are structured here so architectures that don't consider relations among objects in the world such as the subsumption architecture or similar Robotic frameworks working with simple objects or with no relations among the objects, are not suitable. The cognitive examples provided reveal this structure and a suitable language to capture the relations and use them to improve reasoning about the world is required. The second aspect of this architecture is the uncertainty very natural to appear in real world applications but kept to the side when it comes to designing software systems using architectures like the BDI. Some frameworks like Impact have been built upon BDI to include uncertainty which reveals that the actual architecture lacks this ability.

With this new architecture desires, plans and intentions are coded implicitly into the solution of the problem. BDI needs exact definitions of plans and goals before running the agent in the actual environment which can be a hard task to define in big applications. Instead the Robotics framework is used as a guideline design that needs no plan or goal bases, instead builds models that change according to the environment and the belief state. A learning component that allows the agent to reach its goals in terms of maximizing utility functions of any form will also be needed. These models will be explained in part 3 of this proposal.

Logical reasoning and a highly expressive logic is integrated to define the environment and the models of the agent, this language is explained in the next section. This architecture is roughly illustrated in figure 1.

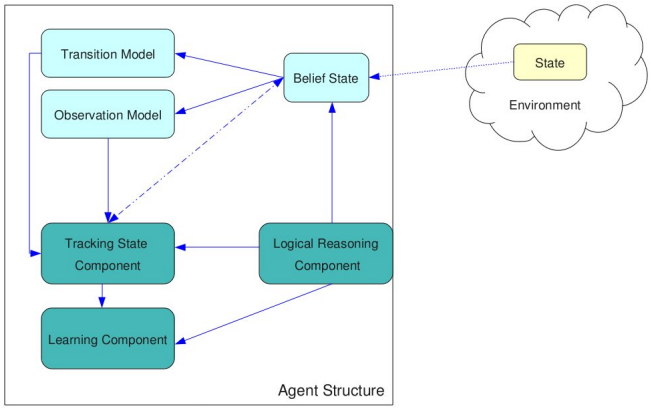

Figure 1: The tracking-acting agent Architecture

2.2 Agent Language

In defining software or intelligent agent frameworks many different languages can be used. Agents include a knowledge base which is a representation of the world. The language presenting these facts is named the knowledge representation language. Percepts are inputs to the knowledge base and acting is the task of asking a query from the KB which is done by logical reasoning to select the best action. The aim of knowledge representation is to present knowledge in a tractable form and logic is the language that provides the syntax and semantics for this.

Most intelligent systems use Propositional logic or descriptive languages such as C or Java to implement their machine learning algorithms. Other software framework willing to capture the relations between objects in the environment use other logics like First-order or Higher-order logic and functional programming languages. These languages provide rich knowledge in the form of predicates that express objects as terms with relations and functions defined on them. This allows a more flexible and compact representation of knowledge where we can define some sort of structure on the world.

The use of logic and relations in the intelligent agent community has been considered important and Statistical Relational Learning (SRL) is one of the current research fields active in this area. It is a sub discipline of machine learning concerned with models of domains that exhibit both uncertainty (which can be dealt with using statistical methods) and complex, relational structure. The knowledge representation formalisms developed in SRL uses (a subset of) First-order logic to describe relational properties of a domain in a general manner (universal quantification) and draws upon probabilistic graphical models (such as Bayesian networks or Markov networks) to model the uncertainty.

In the following a short review on propositional and First order logic and

then define the main concepts of the Higher-order logic.

2.2.1 Propositional Logic

In this logic, symbols represent whole propositions (facts) or logical constants True and False. Proposition symbols can be combined using Boolean connectives (and, or, not, implication and equivalence) or parentheses to generate sentences with more complex meanings. The semantics is specifying the interpretation of the proposition symbols , and using truth tables of the connectives.

A mapping from proposition symbols directly to truth and falsehood, or the label for a row in a truth table is defined as a model. The models of a sentence are just those mappings that make the sentence true. Inference in this logic is done using the truth table for n symbols. This means the complexity will be2^n which shows satisfiability is NP-Complete but of course the proof of a certain sentence refers to a part of the KB and is quick to find.

Unfortunately, propositional logic is not powerful enough for action selection and also we can not capture all change in the world according to attributes such as time (because of the huge complexity) so we need more tools than propositions and this is where relational logic is used considering relations as well as objects in the world. First-order and Higher-order logic are relational logic.

2.2.2 First-order Logic

First-order logic commits to the representation of worlds in terms of objects and predicates on objects (properties of objects or relations between objects), as well as using connectives and quantifiers, which allow sentences to be written about everything in the universe at once. Constants, variables and function symbols(refer to particular objects without using their names) make terms.

Terms and predicates define atomic sentences which are connected via logical connectives like the propositional case to make complex sentences. Quantifiers are used to show properties of a set of objects without enumerating them. Universal quantification makes statements about every object where as existential quantifier makes a statement about some object in the universe without naming it.

In knowledge representation, a domain is a section of the world about which we wish to express some knowledge. Axioms capture the basic facts about a domain, we then define other concepts in terms of those basic facts, and then use the axioms and definitions to prove theorems. First-order logic can quantify over objects but not over relations or functions on those objects. Higher-order logic allows us to quantify over relations and functions as well as over objects.

2.2.3 Higher-order Logic

Higher-order logic can be used as a direct translator of any informal description and also provides suitable foundation for mathematics itself. The most crucial property of this language is higher-order functions which take functions as arguments and/or return functions as results. Historically, this logic can be traced

back to Church's simple theory of types and extends it in that it is polymorphic and admits product types. The definitions below briefly summarize this logic:[13]

Definition 1: An alphabet consists of three sets: a set T of type constructors; a set C of constants; and a set V of variables.

Each type constructor in T has an arity. The set T always includes the type constructor Ω of arity 0. Ω is the type of the booleans. Each constant in C has a signature. The set V is denumerable. Variables are typically denoted by $x, y, z, ...$

Definition 2: A type is defined inductively as follows:

1. If T is a type constructor of arity k and $\alpha_1, ..., \alpha_k$ are types, then $T\,\alpha_1 ... \alpha_k$ is a type.
2. If α and β are types, then $\alpha \to \beta$ is a type.
3. If $\alpha_1, ..., \alpha_n$ are types, then $\alpha_1 \times ... \times \alpha_n$ is a type.

The set C always includes the following constants.

1. $\top$ and $\bot$, having signature Ω.
2. $=_\alpha$, having signature $\alpha \to \alpha \to \Omega$, for each type α.
3. $\neg$, having signature $\Omega \to \Omega$.
4. $\vee$, $\wedge$and$\to$having signature$\Omega \to \Omega \to \Omega$.
5. Σ_α and Π_α, having signature $(\alpha \to \Omega) \to \Omega$, for each type α.

Definition 3: A term, together with its type, is defined inductively as follows.

1. A variable in V of type α is a term of type α.
2. A constant in C having signature α is a term of type α.
3. (Abstraction) If t is a term of type β and x a variable of typeα , then $\lambda x.t$ is a term of type $\alpha \to \beta$.
4. (Application) If s is a term of type $\alpha \to \beta$ and$t$ a term of type α, then $(s\,t)$ is a term of type β.
5. (Tuple)If $t_1, ..., t_n$ are terms of type $\alpha_1, ..., \alpha_n$, respectively, then $(t_1, ..., t_n)$ is a term of type $\alpha_1 \times ... \times \alpha_n$.
6. (Modal Term) If t is a term of type α and $i \epsilon\, 1, ..., m$, then $\Box_i t$ is a term of type α.

Definition 4: Let (X, A, μ) be a measure space and $f : X \to R$ a measurable function. Then f is a density (wrt the measure μ) if (i) $f(x) \geq 0$, $\forall x \epsilon X$, and (ii) $\int_X f \, d\mu = 1$

Definition 5: The function § : $Density\, Y \to (Y \to Density\, Z) \to Density\, Z$ is defined by

$$(f \S g)(z) = \int_Y f(y) \times g(y)(z) d\nu(y)$$

where $f : Density\, Y$, $g : Y \to Density\, Z$and $z \epsilon Z$. .

Specialized to the discrete case, is $(f \S g)(z) = \sum_{y \epsilon Y} f(y) \times g(y)(z)$. Intuitively, § is used to chain together a density on Y and a conditional density on Z to produce a density on Z by marginalizing out Y .

Definition 6: The function \$: $Density\, Y \to (Y \to Z) \to Density\, Z$ is defined by

$$(f \$ g)(z) = \int_{g-1(z)} f(y) d\nu(y)$$

where $f : Density\, Y$, $g : Y \to Density\, Z$ and $z \epsilon Z$.

Specialized to the discrete case, the definition is $(f \$ g)(z) = \sum_{y \epsilon g-1(z)} f(y)$. Intuitively, \$ is used to obtain a density on Z from a density on Y by passing it through a function from Y to Z.

This language is highly expressive and as shown in the next section it can be used for various agent application that are situated in an uncertain structured environment. The uncertainty in representing the world properties and the models obtained from Robotics can by explained using the definitions given in this section.

2.3 Agent Implementation

Agent theorists develop formalisms for representing the properties of agents, and using these formalisms, try to develop theories that capture desirable properties of agents. The representational language is just one part of any agent framework. In order for a language with specific syntax and semantic to work the right proof system or the set of rules for deducing the entailments of a set of sentences from the language is required.

First-order logic uses the axioms for logics of knowledge or belief. The axiomatic method is defined given some situation that one wants to capture, writing down a logical theory that has the intended interpretation as a model; then one can determine the value of any term in the intended interpretation by a (sound) reasoning procedure. An inference rule is sound if the conclusion

is true in all cases where the premises are true. Inference rules for propositional logic are mainly Modus Ponens, And-Elimination, And-Introduction, Or-Introduction, and Resolution. These rules hold for First-order logic as well along with additional inference rules to handle sentences with quantifiers: Universal Elimination, Existential Elimination and Existential Introduction.

For Higher-order logic we need a reasoning system for the logic in which to carry out ordinary as well as probabilistic computations. This is provided by the Bach system which is a probabilistic modal functional logic programming language. Bach programs are equational theories in Higher-order logic and these are what we employ to model applications. The general inference mechanism that Bach uses is defined below:[13]

Definition 7: Let τ be a theory. A computation with respect to τ is a sequence $t_i{}_{i=1}^{n}$ of terms such that for$i = 1, ..., n-1$, there is a subterm s_i of t_i at occurrence o_i , a formula $\forall(u_i = v_i)$ in τ, and a substitution ϑ_i such that $u_i\vartheta_i$ is α-equivalent to s_i and t_{i+1} is $t_i[s_i/v_i\vartheta_i]o_i$.

The term t_1 is called the goal of the computation and t_n is called the answer. Each subterm s_i is called a redex (short for reducible expression). The formula $\forall(t_1 = t_n)$is called the result of the computation.

The occurrence of a subterm s of t is a description of the path from the root of t to s. An occurrence of a variable$x$ in a term is bound if it occurs within a subterm of the form $\lambda x.t.$ otherwise it is free.

The notation $t[s/r]_o$ denotes the term obtained from t by replacing s at occurrence o with r. If x is a variable, the notation $t\{x/r\}$ denotes the term obtained from t by replacing every free occurrence of variable x in t with r. Two terms are equivalent if they are identical up to a change of bound variable names.

Theorem: The result of a computation with respect to a theory τ is a logical consequence of τ.

Bach is closely related to the Haskell programming language that allows pattern matching only on data constructors[12]. Bach extends this by also allowing pattern matching on function symbols and lambda abstractions. Further, Bach allows reduction of terms inside lambda abstractions, an operation not permitted in Haskell.

The reasoning system for the logic underlying Bach combines a theorem prover and an equational reasoning system. The theorem prover is a fairly conventional tableau theorem prover for modal Higher-order logic. The equational reasoning system is a computational system that significantly extends existing declarative programming languages by adding facilities for computing with modalities and densities. The proof component and the computational component are tightly integrated, in the sense that either can call the other.

Higher-order logic is used as a probabilistic modeling language here. The application in part 4 will show the wide range that it can be used, capturing uncertainty and change in the environment. The reasoning and theorem proving powers of Bach will be required for the inference of our model.

3 Tracking and Acting

The language and theory required for this architecture has been proposed in the previous section. Here this language is used to model the architecture components of tracking and action selection.

The tracking framework is inspired by the Robotics sensor and motion model and the action selection is any potential learning that an agent can perform in this context. In this section the main elements required to build an agent are described which will be put in practice in section 4.

3.1 Combining logic, probability and learning

The components of this architecture are defined by integrating the selected logical theory with probabilities. This integration has been the topic of extensive research in the recent years.

Logical and probabilistic AI have been developed independent of each other for years. The side effect was developing probabilistic reasoning only for the propositional case. Most research in the AI community use complex probability theories only with propositional statements, eliminating the power of relations that exists in logics. More recent research has concentrated on lifting these to the First-order logic. First-order representation and structure allows performing lifted inference, affecting a group of individuals as terms. This can gain speed and rich knowledge in the inference procedure.

If an environment has uncertainty, a distribution is used on the First-order interpretations to show the most probable environment or the answer to some unknown probability. The structure of the product distribution can be represented using graphical models and patterns can be learned from these models. A whole field of research exists here mainly with the overall topic of Statistical Relational Learning (SRL).

Many (if not most) real-world application domains are characterized by the presence of both uncertainty and complex relational structure. Statistical learning focuses on the former, and relational learning on the latter. Statistical relational learning seeks to combine the power of both. A simple survey on the most famous models are given in[10] explaining Probabilistic Relational Models(PRM), Relational Dependency Networks(RDN),Markov Logic Networks(MLN) and Bayesian Logic Programming(BLP).

The main structure in the literature represented by the SRL community defines the logic as a mean to express the representations of the graphical model in a compact way. For most cases Prolog or First-order logic is used as the logical language and some probabilistic model such as Bayesian networks, Dependency networks or Markov logic networks are used to support the structure of the model.

The parameters involved in the probabilistic model has to be learned using machine learning techniques such as Expectation-Maximization algorithm(EM), boosting, gradient descent methods or Inductive learning programming(ILP) approaches. Logical representations allow inference to take place at the lifted

level. In some complex situations this has to be reduced to partial grounding by approximate inference or other belief propagation methods.

Here a different logic that already incorporates probabilities and is powerful enough to perform learning algorithms is considered. Using First-order logic won't allow reasoning about densities to happen in theories. In Higher-order logic, a formula is a term whose type just happens to be boolean. It is also an advantage to compute the value of arbitrary terms, not only formulas.

The key idea in the integration is to allow densities to appear in theories. According to Definition 4 in the previous section, in this logic, the type synonym $Density\ a \ \equiv a \rightarrow Real$ is defined such that any term of type Density τ, for some τ, is called a density. Considering a function $f : \sigma \rightarrow \tau$ for which there is some uncertainty about its values to model, a function $f' : \sigma \rightarrow Density\ \tau$ exists where for each argument t, $(f't)$ is a suitable density for modeling the uncertainty in the value of the function $(f\ t)$.

In First-order logic there is assumed to be a distribution on interpretations and answering queries involves performing computations over this distribution. This approach is intrinsically more difficult than computing the value of terms in the no probability case. The approach used by most of the AI research is building a theory and proving theorems with this theory instead. Instead in Higher-order logic a theory is used to model a situation and produce results that are correct in the intended interpretation. As mentioned before this allows using probabilities in an ordinary fashion similar to other functions in a single intended interpretation where as in previous approaches SRL methods suggest using various machinery. Probabilistic reasoning is handled by Bach's computation system which is an equational reasoning system with special support to make probabilistic reasoning efficient. This is provided by equations that implement variable elimination and lifted inference.

Previous work has proved that methods in the SRL field can be represented and inferred with Higher-order logic[13]. By defining the Bayesian framework as the main component of our agent we show the ability to track and learn. The structure and parameters of this Bayesian framework are learned using methods such as EM and Reinforcement Learning. Inference is also performed according to section 2.3.

3.2 The Tracking framework

In the Robotics sensor and motion models by Thrun[17] the aim is to estimate the location of a robot given noisy measurements. The robot (or agent in our case) has a belief about where it is, so at any time it does not consider one possible location, but the whole space of locations. Tracking is considered as a localization problem consisting of estimating the probability density over the space of all locations and for agents the location is the state of the agent within the environment. The state more likely to be the location of the agent is the highest probable state, given a probability distribution or belief state. The real distribution of the agent state has a single peak at the true state and is zero everywhere else. If the agent achieves this goal, then it knows exactly where it

is located.

Two probabilistic models are used to update the beliefs:

- Transition model: The probability density of the transition function gives the probability that if at time step k - 1 the agent was at state x_{k-1} and performed action a_{k-1}, then it will end up at state x_k at time step k: $P(x_k|x_{k-1}, a_{k-1})$
- Observation model: The probability that a sensor observes s_k from a certain state x_k at time k by the density: $P(z_k|x_k)$

The agent starts with an initial belief. If the agent knows where it initially is, then $P(x_0)$ is a point mass distribution on the state where the agent knows it is. In the other extreme case that the agent does not know where it starts, the initial belief is a uniform distribution. Any other prior knowledge leads to more informed distributions in this range.

To formulate the above according to the Bayes rule in Robotics we have:

$$Bel(x_t) = \eta p(o_t|x_t) \int p(x_t|x_{t-1}, a_{t-1}) Bel(x_{t-1}) dx_{t-1}$$

The agent has a prior over the set of states (possible belief states), at each step it propagates the transition model and updates the observation model using Bayes rule. This results in a posterior probability over the set of states which again becomes a prior for the next stage.

- Prediction step: $\overline{Bel}(x_t) = \int p(x_t|x_{t-1}, a_{t-1}) Bel(x_{t-1}) dx_{t-1}$
- Measurement update step: $Bel(x_t) = \eta p(o_t|x_t)\, \overline{Bel}(x_t)$

An autonomous agent is an agent receiving input from the environment and producing output to the environment. The input is in the form of observations via the sensors and rewards or feed backs from the environment and the output is the action performed by the agent that has the goal of maximizing its expected future rewards based on some utility functions. There are two extremes in building autonomous agents. On one end is the universal agent that assumes nothing about the environment or its structure and learns everything. This universal agent captures the unknown environment aspects by a probability function which has to be learned or learning the mapping from histories to actions (policy) so it is a model-free technique.

On the other end is the Robotics view where the environment is known to the agent so it limits its learning to models of the environment through the agent sensors. Here the agent keeps track of its location at every time step through the percepts received and acts according to that. This approach is based on the observation and transition models of the environment and it is a model-based approach towards acting optimally.

In our framework we tend to approach between these two extremes. Here we do not have full knowledge about the environment like the Robotics case and we don't want to act without any prior knowledge either. So we build in

some prior knowledge in the form of partially observed environment models. This means that we can have partially defined $\eta - functions$ partitioning the environment model into several sub-models. Partial models like the transition and observation models are some of these defined functions that partition the state space.

3.3 The learning framework

The agent architecture is based on a Markov decision processes (MDP) and in the partially observable case this will be extended to a Partially Observable MDP. Reinforcement learning is an area of research that covers learning by choosing actions so as to maximize an expected cumulative reward over a time horizon. It is considered as a different approach to supervised and unsupervised learning where the agent learns from examples or a teacher in the first case and learns patterns or principles in the second.

A Markov decision process consists of the following:

- A finite set S of states, for each time step t ∈ T where T={0,1,..}
- A finite set A of actions, for each t ∈ T
- For each state $s_t \in S$ and each action $a_t \in A$, a transition probability distribution $p_t(s_{t+1}|s_t, a_t)$, for each $t \in T$
- A reward function $r_t : S \times A \to R$, for each t ∈ T

If the environment can be modeled as an MDP, we need only the transition model from the tracking framework to compute the probability of the next state. The observations in the observation model are not defined because the environment is fully available to the agent so taking an action and state, the transition function can fully determine the next state. In order to act optimally we will use the reward function to find the future expected value of each set of action and states. This will lead to an optimal policy $\pi^* : S \to Prob(A)$ defining the sequence of actions that by performing them the agent can receive the highest possible reward, considering future states. Bellman equations are used to learn this policy.

The state space can be define by the agent designer or left as a learning task for the agent. In the latter, the agent will use the set of histories of actions, observations and rewards to partition the history into a suitable state space and the function is defined as $\phi : H \to S$. [9]

In most real world problems, full knowledge of the effects of actions or certainty within the received percepts is not available Uncertainty is provided in this case to capture the partially observable environment surrounding the agent, while the environment has to be modeled using a POMDP .

A Partially Observable Markov decision process consists of the following:

- A finite set S of states, for each t ∈ T
- A finite set A of actions, for each t ∈T

- A finite set O of observations, for each t$\in$ T
- For each state $s_t \in S$ and each action $a_t \in A$, a transition probability distribution $p_t(s_{t+1}|s_t, a_t)$, for each $t \in T$
- For each action $a_t \in A_t$ and resulting state $s_t \in S$, an observation probability distribution $p_t(o_t|s_{t+1}, a_t)$, for each $t \in T$
- A reward function $r_t : S \times A \to R$, for each t $\in$ T

In this case we are supplied with three functions defining our models:

- Transition model: $A \to S \to Prob(S)$
- Reward model: $A \to S \to R$
- Sensor(observation) model: $S \to Prob(P)$.

In order to obtain an optimal policy $\pi^* : S \to Prob(A)$ the actual state space is not suitable as there is uncertainty on it according to the observation model, instead a distribution is maintained on the state space named the belief state. This means that partitioning the history now takes the form of $\phi : H \to Prob(S)$. The policy is much harder to obtain in this framework because of the size of the potential belief state.

During this thesis we will approach the learning problem from different perspectives. This is highly dependent on the application and the needs to incorporate different learning algorithms to improve agent performance but in general the following areas are subject to learning algorithms:

- Learning the transition function.
- Learning the observation function or obtaining it by training examples.
- Learning the policy function depending on the environment.
- If the state space is very large, features are defined on the state space and work is done on a smaller set of equivalent classes of states defined by the features. Features might be needed to be learned by training examples.
- Learning to act optimally or selecting the action in order to maximize some utility function.

The environment is modeled according to one of the situations explained. The parameters of the transition and observation functions have be learned if not provided by the problem design. In the case of problems like Robotics these functions are given to the agent as a design requirement but in many AI problems and also in cognitive examples, defining these functions are complicated and learning method are required to help extract useful information.

Features on the state space are defined by predicates in logic and this helps partition the state space. Belief acquisition methods can be used to learn the features or extract them from available data. Action selection will the last goal of

our agent as it tries to optimize its utility function or the rewards. Reinforcement learning methods along with other algorithms are used at this level. In the next section the application of this architecture and some of the early stage results obtained by applying our modeling and development are presented.

4 Trading Agent Competition

4.1 Game overview

In supply chains the task of planning and coordinating the procurement of materials and delivering the goods is a management task and involves meeting changing market conditions in good timing and low costs. This dynamic environment has become the topic of research in recent years and the trading agent competition is one of the many platforms[1]. The game has interdependent and incomplete data caused by concurrent market conditions and other agent behaviors which makes it suitable for an application of our framework.

A general game server provides the environment for agents by simulating customers ordering personal computers and suppliers offering parts for each PC which consists of a motherboard, disk drive, memory and CPU. Six agents compete in 220 days as manufacturers against each other to gain maximum profit by winning bids on customer requests and buying materials from the suppliers. They also have to coordinate their own inventory and factory capacity.

At the beginning of each day, customers issue requests for quotes(RFQ), all agents receive them and according to their inventory levels of components or computers or their free factory capacities to produce them, bid on the RFQ. The next day the customer selects the best offer made according to the lowest price and best offer date. This order is then issued to that agent and the agent is responsible for delivering the product in time.

The computer components are bought from the 8 suppliers that offer prices for their product and the best price has to be chosen by the agent. The agent has limited assembly lines and keeping components and ready computers in the inventory has a cost, so it must consider the trade-off of buying components and having them ready in the inventory to win bids and producing computers on the demand of receiving requests.

In order to clarify the main roles on an agent that will need to operate in this supply chain, we review some of the main points of the game.

- Each day the agent must autonomously perform tasks such as:
  - Negotiating supply contracts
  - Bidding for customer orders
  - Managing assembly activities by sending production schedule to factory
  - Shipping orders to customers by sending delivery schedule to factory
- Each day our agent receives the following messages:

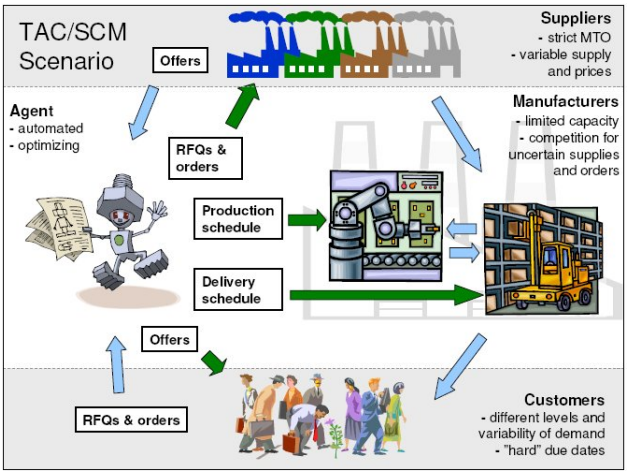

Figure 2: TAC/SCM game scenario from the official game site: http://www.sics.se/tac/page.php?id=13

  - From customers: RFQ, offers won, penalty and cancellation (late response)
  - From suppliers: response of component offers, delivery of earlier supplies
  - From Bank: statement (adds money when customer pays, reduces when penalized or supplies are received)
  - From Factory: inventory report (daily inventory report to agent reveal components and finished PC's in the inventory will cost a storage charge)

- The customer and agents interaction each day is in the form of the following messages:

  - Customer demand(cRFQ): taking the format of [product type, quantity,due date, reserve price, penalty amount] all amounts are chosen randomly from intervals where the maximum and minimum is given in game specifications to all agents.
    cRFQs are categorized into 3 ranges and the number of cRFQs in each range is determined by a random walk.
  - Bid: If agent wants to respond to a cRFQ, it sends a bid in the form of [cRFQ, price, quantity, due date]. It has to satisfy the whole quantity requested by the customer and the exact due date while the price can be less that the one offered by the customer.
    The customer will accept one of the bids from an agent who has made the bid with the lowest price and will issue an order to that agent.

  - Fulfilled Orders: The orders ready to be shipped by agent to the customer. This event will trigger the payment on the agent's account or issues a penalty for delays per day (after 5 days the order is canceled).

- On the other side, the suppliers and agents engage in the relation explained below:

  - Request supplies: Each agent can send 5 RFQs for each product (10 per supplier) in the form of [quantity, component, delivery date, price]
  - Offer generated: Each supplier computes the price of each component it has to offer lower than the request price from the agents. This price is determined according to the production capacity of suppliers dependent on the inventory and capacity of each supplier.
    The supplier will offer prices according to better agent reputations and lower prices are issued to longer due dates. Again there is a trade off of buying earlier with a higher price of waiting and buying it a lower price while gaining a risk that the supplier has no capacity left to offer components.
  - Order: the best offer from the suppliers that maximizes an agent revenue is chosen in the form that comes from the supplier [RFQ, price, adjusted quantity and due date]. Rejecting offers on the request that an agent has made will reduce its reputation.

According to this brief summary of the game, the high interaction among the agent and the environment and the great levels of uncertainty is obvious. We will focus on the uncertain aspects of the game and try to resolve it using our tracking framework. The main sources of this ambiguity within the trading agent competition in identified in the future customer demand and pc pricing, supplier capacity and pricing and also in the behavior of other agents. We will look into each of these problems and try to solve them using techniques from our framework and other related learning mechanisms.

4.2 Solution

The solution to this dynamic supply chain management problem has been proposed by many researchers since the game started in 2003. Most solutions divide the problem into two or more components such as the customer, supplier, inventory and factory part and try various algorithms to solve them separately. Because of the high interaction between the components the results have to be passed to each other in an efficient way and models like the multi-agent model or message passing in a single-agent model are used to coordinate the procurement and selling of the computers. An example of the methods in the leading agents are provided below, these agents are the first three winners of the 2010 competition:

- TAC TEX[14]: This agent has two main modules that interact with each other: Supply manager module that deals with the suppliers to buy components to store in the inventory and the Demand manager that deals with the customers handling pc orders and factory production schedules. There are some predictive modules in this agent to help the agent perform the best actions.
  The Supplier modules predicts future component price and availability using statistical methods, the Demand module keeps track of the customer demands for RFQs using a Bayesian approach, which involves maintaining a probability distribution over expected number of RFQs on a certain day and the demand trend on that day, also the Offer Acceptance Predictor predicts the probability of acceptance of a certain bid using previous bid results and a linear regression algorithm on the minimum and maximum daily prices.
  For the inventory it maintains a safety stock threshold for each component and bids according to the projected inventory. As for the factory production it uses greedy algorithms to produce each order as late as possible.

- DEEP MAIZE[11]: The main contribution is to use a multi-agent equilibrium analysis using distributed feedback control. To balance the price of each computer which is the sum of base prices, a level equilibrium is defined as a value such that the aggregate quantity demanded at that level equals the quantity supplied.
  The marginal values for each finished product and component input is estimated given predictions about market conditions and constraints on production. These valuations allow the problem to be decomposed into manageable sub problems, while retaining the ability to do high-level planning. The agent starts by using any new information it receives to update its beliefs about the hidden state of the supply chain. It then makes comprehensive predictions about the conditions it faces in the markets for components and finished PCs, accounting for the behavior of other agents.
  The next stage makes high-level decisions about long-term production scheduling. The projected production schedule approximately optimizes the agent's expected profit margin with respect to the market predictions, subject to constraints on factory capacity and component arrival.

- MINNETAC[5]: This agent has a component-oriented design using the Apache Excalibur component framework, and the "agentware" package distributed by the TAC SCM game organizers. The MinneTAC design is a general architecture for blackboard-oriented agents with multiple decision processes. It is a set of components layered on the Excalibur container, four of these components are responsible for the major decision processes: Sales, Procurement, Production, and Shipping.
  The Repository, Oracle and communication components which are not obvious are explained here. Repository is the component that is visible to all the other components and at the beginning of each day of the game, new

incoming messages are deposited into the Repository. The basic communication behaviors are implemented in the agentware package, provided by the game organizers so the agent wraps the entire agentware package with an Excalibur component that has responsibility for communications with the game server.The Oracle component is essentially a meta-component, since its only purpose is to provide a framework for a set of small configurable components that may be used to perform analysis and prediction tasks mainly Evaluators.
Modeling and analysis tools (Evaluators) are designed as simple services, strung together at run time to provide input to decision components. For example to maximize the price for sales, it sets a price that not all customers will accept and then one can increase the information content of orders by "spreading" prices.
The strategies for the other 4 components are very different for example for the offer predictor a reverse Linear Cumulative Density Function along with such additional factors such as quantity and due date is used. For the selling pc part, the Bayesian approach similar to that of TAC TEX is used. For the supplier offer an internal marketplace structure with competing bidders is defined. The inventory orders components and produces goods according to customer orders and free factory cycles.

Many other approaches have been considered in presenting an agent (or agents) that can win this competition and gain some insight into the research field of SCM. We want to propose a new method based on our agent framework. We will model the environment using our two available models which would need the exact definitions of states, actions and observations of the agent. Once we have the transition and observation model, we perform tracking to get an estimate of the current belief state and then we use learning methods to solve the action selection problem.

By reviewing the current research in this field it seems that partitioning the problem is inevitable so in order to reduce the state size we will partition the problem. The overall problem can to be divided into 3 parts: customer interaction,supplier interaction and the factory. Ideally if we had enough information about all parts of the game including other agents, we would define the state as below:

$$\begin{aligned} State = (&Agent\ InternalState \times CustomerState \times \\ &SupplierState \times OtherAgents\ InternalState) \end{aligned}$$

$$\begin{aligned} State = (&inventory_{1..16+4}, profit, factory) \\ &\times (cRFQ\ List \times (price \to \Omega) \times trend_{1..3}) \\ &\times ((offeredprice \to \Omega) \times capacity_{1...8} \times reputation_{1...8}) \\ &\times (reputation_{1..8}, inventory_{1..16+4}, Pc - price, profit)_{1..5} \end{aligned}$$

Some parts of the state are unknown or have uncertainty in their nature which we will try and capture using our models. Different parts of the state is defined as below:

- For determining the acceptance of a cRFQ we will need the list of all the requests which include their price, date and quantity. The winning price of each computer type has to be determined using previous data in order to predict future selling prices.
  To estimate the future demand levels or the number of cRFQs we will also need the demand and trend parameters in the state.

- To buy components from the suppliers we will need to estimate the price of each component per supplier for each due date. This depends on the capacity of each supplier so an estimate of that using pricing formulas is desired. Also to keep track of our reputation we need this number per supplier.

- To predict future inventory levels we will need the current inventory of the components and computers and the capacity of each factory line.

- The opponents details are all hidden for us and this means that their reputation, inventory levels, pc prices and also their profit may me modeled with uncertainty if required.

The actions our agent will have to perform in order to gain profit are as below:

- bidding on customer orders
- sending RFQs to suppliers
- issue order on components
- deliver products to customers
- send production schedule
- send delivery schedule

The set of observation that will be available to our agent are:

- receive list of cRFQ from customer
- receive orders on accepted bids from customer
- receive cancellation on an order
- average low and high selling price of all agents per product type of the previous day
- receive component offers from suppliers
- receive bank statement
- receive inventory report
- receive market report

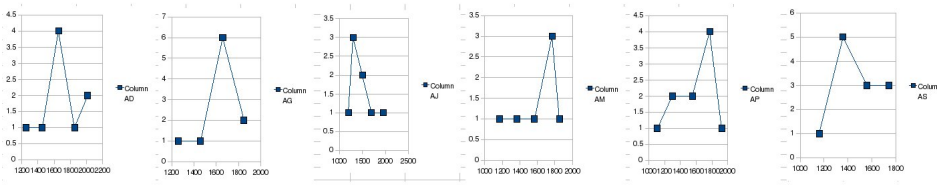

Figure 3: The price outcomes for one computer type in 6 subsequent days

- receive supplier report
- receive product

In the next section the modeling and implementation of the customer part of this abstract agent is explained in more details.

4.3 Applied method

In this section the customer demand part of the trading agent is looked at from a modeling perspective. The Robotics sensor and observation models will be defined in order to track the state of the agent and before that we have to define the action, observation and state.

Each day the agent receives new cRFQs and according to the previous days' accepted prices for the 16 computer types and the highest and lowest bid accepted by the customer, it assigns a bid over this set of cRFQ with the desired (optimum for the agent) price.

The utility function is defined as the expected sum of the immediate reward and the long-term reward under the best possible policy. The utility in our problem is to obtain maximum profit (sales versus penalties). The action is the price set on each bid, in response to every cRFQ.

$$BidPrice, ProductType, BasePrice, HighPrice, LowPrice = Int$$
$$ProductType = 1...16$$
$$basePrice : ProductType \rightarrow BasePrice$$
$$(basePrice\ 1) = 2250$$
$$CRFQ = Price \times ProductType \times Quantity \times DueDay$$
$$Order = Price \times ProductType$$
$$bidding : List\ BidPrice \rightarrow Action$$

Every day new cRFQs arrive as inputs to the system and we receive the list of previous day's successful bids (named orders) and also a report indicating the highest and lowest price of the successful bids on each computer type. These are considered as the system observations.

$$Observation = (ProductType \times HighPrice \times LowPrice) \times (List\ Order)$$
$$CRFQBundle : (List\ CRFQ) \rightarrow Observation$$

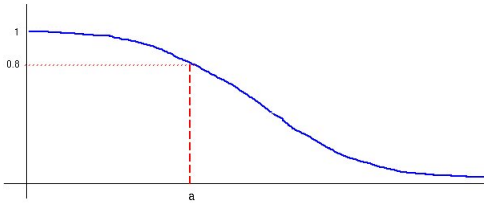

Figure 4: Gaussian Cumulative function used for the outcomeProb function

In this model the difference between the daily lowest and highest price of each computer type can be modeled as a Gaussian on the price and the acceptance rate can also follow a Gaussian (or uniform distribution) on the price. The Gaussian distribution is chosen due to the fact that the observations per day suggest a uni-modal distribution, this diagram has been shown for six subsequent days in figure 3.

The state of the system in each time step is defined to be the list of cRFQs, whether they will be accepted as a bid from the customer, which is a list of booleans, the mean price of each product type and the demand trend. To evaluate the state the bid outcome parameter is essential inside the state.

$$
\begin{aligned}
&Demand, Range, Day = Int \\
&Trend = Real \\
&Mean, Variance = Real \\
&meanPrice = ProductType \times Mean \times Variance \\
&DemandParam = Demand \times Trend \times Range \\
&Range = 1..3 \\
&BidOutcome = Bool \\
&State = (List\, CRFQ \times List\, BidOutcome) \times MeanPrice \times \\
&\qquad DemandParam \times Day
\end{aligned}
$$

Taking any random action for now (action selection is preformed in the learning phase following the tracking phase) which means any price for each of the cRFQs, along with the current list of cRFQs in the state, we want a probability over all the states this action can lead us to. The probability that a certain cRFQ will be accepted according to the action depends on the bids of the other agents.

The estimated price that the customer will accept for a bid on any cRFQ is predicted using the transition model. For a certain bid to win, we need the price in the action to be lower than the price estimated by the state. The cumulative distribution function of each of the 16 Gaussians estimated by our agent will be used for our transition model and an example of it is illustrated in figure 4.

$$
\begin{aligned}
&transitionModel: \ Action \rightarrow State \rightarrow (Density\ State) \\
&(transitionModel\ bidding[a_1, ..., a_n]\ ([(r_1, \perp), ...(r_n, \perp)]\ p\ t\ d)) =
\end{aligned}
$$

$$\lambda s.if\ (s = [(r_1, o_1), ..., (r_n, o_n)], p', t, d))then\ \prod(outcomeProb\ r_i\ a_i\ pdo_i)\ else 0.0$$

To track the state of the winning price of each computer type (relative to the probability of acceptance of a bid) we can use the Kalman filter. Kalman filter is a recursive filter that estimates the state of a linear dynamic system from a series of noise measurements. The price of the received orders of the next day show that they tend to follow a Gaussian distribution so we can use this filter for our analysis of the winning price. The multivariate normal distribution is stated as the following:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\}$$

The density over the variable x is characterized by the mean $\mu$and covariance Σ. In order to get Gaussian posteriors the following properties must hold:

- Markov assumption of the Bayes filter
- State transition probability must be a linear function of its arguments with added Gaussian noise. With the following equation: $x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t$ where $\varepsilon_t$models the uncertainty by a Gaussian (0,R_t), the posterior is as below with the mean the first two parts of the equation and the covariance R_t:

  $$p(x_t|u_t, x_{t-1}) = \det(2\pi R_t)^{-\frac{1}{2}} \exp\{-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^T R_t^{-1}(x_t - A_t x_{t-1} - B_t u_t)\}$$

- The measurement probability must be linear in its arguments with added Gaussian noise with the equation: $z_t = C_t x_t + \delta_t$ where $\delta_t$models uncertainty with the Gaussian (0,Q_t). The posterior probability is as below:

  $$p(z_t|x_t) = \det(2\pi Q_t)^{-\frac{1}{2}} \exp\{-\frac{1}{2}(z_t - C_t x_t)^T Q_t^{-1}(z_t - C_t x_t)\}$$

- The initial belief must follow a normal distribution (μ_0, Σ_0):

  $$p(x_0) = \det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\{-\frac{1}{2}(x_0 - \mu_0)^T\Sigma_0^{-1}(x_0 - \mu_0)\}$$

For our problem, we consider the state to be the price of each computer type which is linear in respect to the previous price and the action which is a new bid price with some added noise and the initial belief state is a Gaussian on the base price of each computer type. The observations are also linear in accordance to the state with some Gaussian noise added because of the uncertain environment. This suggests that using the Kalman Filter is appropriate for our application.

The Bayes filter formulas for the transition ($\overline{bel}$) and observation model (bel) is given below:

$$\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$$

$$bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t)$$

By substituting the probabilities for the belief states, the Kalman filter algorithm is shaped in which instead of the belief state, the mean and covariance changes in each time step. The mean and covariance for the transition model (and before applying the new observation) is stated below:

$$\overline{\mu}_t = A_t u_{t-1} + B_t u_t$$

$$\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$$

As for the observation model the mean and covariance change according to the following:

$$\mu_t = \overline{\mu}_t + K_t(z_t - C_t \overline{\mu}_t)$$

$$\Sigma_t = (I - K_t C_t) \overline{\Sigma}_t$$

$$K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Qt)^{-1}$$

The parameters of this model (A,B,C,R,Q,μ_0, Σ_0) can be determined using learning methods such as the maximum likelihood through the EM algorithm.

In the E step which is an inference problem, using the sum-product algorithm the posterior marginals for the latent variables are determined. This part has a solution similar to the Kalman filter and defines the latent variable state by the observations and previous state.

In the M step the expectation of the complete-data log likelihood with respect to the observed data and the old parameters is defined and the maximum of this function is desired.

We will use a simplified version of the Kalman Filter as the starting point and assume identity matrices and no control in the estimation phase. The parameter values have to be learned using previous game logs as an extension to this current work. For modeling purposes we have the following according to this first stage Kalman filter:

$projProduct : cRFQ \rightarrow ProductType$
$(projProduct\ (, p, ,)) = p$

$outcomeProb :\ cRFQ \rightarrow BidPrice \rightarrow MeanPrice \rightarrow Day \rightarrow (Density\ BidOutcome)$

$(outcome\ r\ a\ (p\ m\ v)\ d\ \top) = if\ (=(projProduct\ r)\ p) then$
$\qquad (1 - \int_0^a (kalmanEstimate\ (p\ m\ v)\ d\ a))$
$(outcome\ r\ a\ p\ d\ \bot) = 1 - (outcome\ r\ a\ p\ d\ \top)$

$NoiseMotion, NoiseSensor = Real$
$kalmanEstimate : MeanPrice \to Day \to BidPrice \to (Density\ Price)$
$(kalmanEstimate\ ((p\ m\ v)\ d\ a)) = \lambda q.if\ (d = 1)\ then$
$\quad (gaussian \quad (basePrice\ p) \quad ((basePrice\ p)/12 * (basePrice\ p)/12) \quad q)$
$\quad else (gaussian\ m\ NoiseMotion + v\ q) else\ 0.0$

The belief state contains a distribution on the current state for each type of computer. By updating the transition model - convolution - the new belief state distribution is defined:

$transitionUpdate : Action \to (Density\ State) \to (Density \quad State)$
$(transitionUpdate\ a\ ds) = \lambda x. \sum_y (ds\ y) \times (TransitionModel\ y\ a\ x)$

The next day the orders are received to show which of the bid outcomes were estimated correctly. The kalman correction algorithm is applied here and the observations considered will be the high and low prices of the report and our own order prices. This average is used as the correction value on the estimated price.

$observationModel : State \to (Density\ Observation)$
$(observationModel([(r_1, b_1), ...(r_n, b_n)]\ pr\ t\ d)) =$
$\quad \lambda o.if\ ((o = (p\ h\ l)\ lo)$
$\quad then\ (kalmanCorrect\ o\ pr)$
$\quad else\ if\ (CRFQBundle\ lr\ )\ then\ (demandProb\ lr\ t\ d)\ else\ 0.0$

$kalmanCorrect : Observation \to MeanPrice \to (Density\ Price)$
$(kalmanCorrect\ ((p_1\ h\ l)\ lo)\ (p_2\ m\ v)) = if (p1 = p2) then$
$\quad \lambda q.if((m' = m + K \times ((observedPrice\ o) - m)) and$
$\quad\quad (K = (v + NoiseMotion) \times (c + NoiseMotion + NoiseSensor))$
$\quad\quad and (v' = (Identity - K) \times (v + NoiseMotion)))$
$\quad then\ (gaussian\ m'\ v'\ q) else\ 0.0$

$observedPrice : Observation \to Price$
$(observedPrice\ (p\ h\ l)\ lo) = \lambda q.if (p = q) then\ (h_q + l_q + lo_q) else\ 0$

$orderPrice : Observation \to Price$
$(orderPrice\ o) = \lambda p.if (o = (p, ,)[l_1, ..., l_n]) then\ if\ (p = projProductOrder\ l_i) \Sigma_{i=1}^n l_i$

$projProductOrder : Order \to ProductType$
$(projProduct\ (p, , ,)) = p$

The other observation we receive each day is the cRFQs and in order to estimate the future demand levels in each range we use the following model to estimate the trend of requests:

$demandProb : (List\ cRFQ) \rightarrow DemandParam \rightarrow Day \rightarrow (Density\ Demand)$
$(demandProb\ lr\ t\ d) = normalize\ \lambda q.(demandLikelihood\ q\ d)$

$\epsilon = uniform[-0.01, 0.01]$

$demandModel : DemandParam \rightarrow Day \rightarrow Demand$
$(demandModel\ (q\ t\ r)\ d) =$
$if\ (d = 0)\ then\ if\ (r = 1\ or\ r = 3)$
$\quad then\ (uniform[25, 100])$
$\quad else\ if\ (r = 2) then (uniform[30, 120])$
$else\ (min\ qmax\ (max\ qmin\ (trackTrend\ (q\ t\ r)\ d) \times q))$

$trackTrend : DemandParam \rightarrow Day \rightarrow Trend$
$(trackTrend\ (q\ t\ r)\ d) = if\ (d = 0)\ then\ if\ (r = 1\ or\ r = 3)$
$\quad then\ (uniform[0.95, 1/0.95]))$
$\quad else\ if\ (r = 2)\ then\ 1$
$\quad else\ if\ ((q \times t < 25) and (r = 1\ or\ r = 3)) then\ 1$
$\quad else\ if\ ((q \times t > 100) and (r = 1\ or\ r = 3)) then\ 1$
$\quad else\ if\ ((q \times t < 30) and (r = 2)) then\ 1$
$\quad else\ if\ ((q \times t < 120) and (r = 2)) then\ 1$
$else\ (max\ 0.95\ (min\ 1/0.95\ t + \epsilon)$

$demandLikelihood : DemandParam \rightarrow Day \rightarrow (Density\ Demand)$
$(demandLikelihood\ dp\ d) = (poisson\ (demandModel\ dp\ d))$

For the observation update, we apply the Bayes rule to normalize the results obtained in the observation model:

$ObservationUpdate : Observation \rightarrow (DensityState) \rightarrow (DensityState)$
$(ObservationUpdate\ o\ ds) = (normalize\ \lambda z.(ds\ z) \times observationModel\ s\ o))$

In order to test this model, we used the starter agent from the official site on TAC SCM and built the customer model on top of it. The agent plays against dummy agents as a default and since they all show similar trends and don't compete with our agent, we downloaded other agents from the agent repository and played against the stronger agents.

The agents we chose are Tac-Tex, DeepMaize, MinneTac, Botticelli and the Crocodile agent. Our goal in this section was to show that our simple model of the customer can track the price changes in the environment with a rough estimation of the true price of each computer type.

Simulation results for this phase is demonstrated in figure 5 and 6. We compared the results of our estimated mean price using the simplified Kalman filter (represented by the yellow line) with the actual order prices (offers won) from all the other agents extracted from the game logs on the server.

Figure 5 shows the result of tracking for the case that the correction phase uses the high, low and average of all winning orders of our agent as the observation. The result is for computer type 15 and similar graphs can be shown for other 15 computer types too. The result shows that in general our estimation

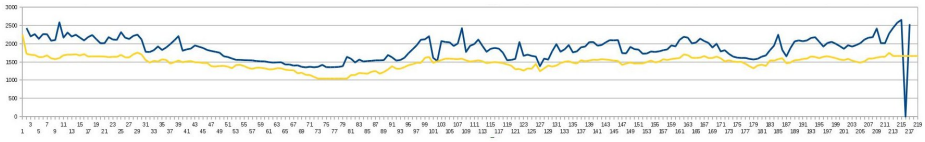

Figure 5: Comparing estimated prices of product 15

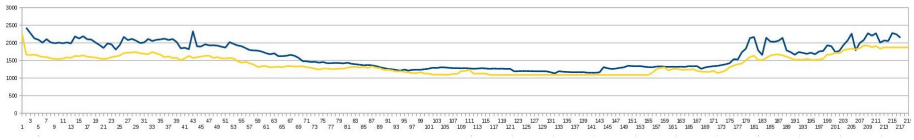

Figure 6: Comparing estimated prices of product 14 without considering the maximum price

can track the winning price. In the middle phases of the game this estimation is more accurate due to the fact that the agents aren't fighting for requests but in the first 40 days and the last days of the game where the other agents are bidding aggressively, our simple estimate fails to track the exact changes in the environment.

Figure 6 has the same conditions as figure 5 except the fact that the highest order price is not considered in our observation for the correction phase. This result shows that by excluding the maximum price report, the agent can report better tracking because customers are interested in the lowest possible price for a computer request so the orders that offer the lower prices are winning with higher probability.

The second figure has better tracking results. Again during the mid game phase results are very similar to the actual winning price where as in the beginning and end of the game, the game conditions change dramatically so we need a more complex model designed for these stages. To complete this work, we will consider learning the parameters of our model using the EM algorithm and based on past game logs available on the server.

To conclude this section, the state of the world and the sources of uncertainty are captured using the models of our framework. The observation and transition models for the customer section of the agent was modeled and implemented using a simplified version of the Kalman Filter. Although parameters of this filter will have to be tuned and set for this specific part of the game, early results show actual tracking of the price of computer types. In the next stage, learning the parameters and modeling using a more complex filter and tracking accurately will be investigated. This localizing then has to be used in the action selection stage and should lead to optimal acting according to the maximum profit.

5 Conclusion and Future Work

The main focus in this thesis is to present an efficient architecture for agents that are situated in an uncertain environment with complex structures. This overall framework will allow designers to use the powers of probabilistic modeling and inference using a highly expressive language as well as the ability to learn desired features. Tracking the agent situation and learning the appropriate actions are the most important aspects of this thesis that have been emphasized throughout the application of the Trading Agent Competition. We have pointed out some of the methods applied but a more abstract view of the expected outcome is given below:

- Using Higher-order logic to represent and model the entire problem domain. This logic is highly expressive and is able to catch various aspects of cognitive problems, handling probabilities and densities over functions.
- Defining transition and observation models for the problem domain. If the environment is an MDP the transition function is only required to track the state and select the action. When the environment forces uncertainty on the state space, the state will become partially observable and related observation model must be defined.
- For the models explained above, we consider discrete and continuous distributions over the state space depending on the problem and verify the output in the logical format. Conjugate priors like the Gaussian model are desired since they remain the same during the filtering process. Examples will be provided to show how the distributions will work in practice.
- Complex state spaces with structured knowledge will be represented using Higher-order logic and cognitive applications show the effectiveness of using structured knowledge. The distributions on these structured states must be defined and tracked. Simple examples from our application or other fields will be demonstrated to show this result.
- The tracking framework must be defined and shown in practice (on applications like TAC SCM). Updating models and how the actual state will be close to the most probable belief state has to be proved. In our application we will use the game logs on the server to view how the hidden variables will be tracked according to our defined model. This has partially been done for the customer side but for the inventory and supplier model we have to perform similar operations.
- Learning parameters and structures of the probabilistic models has to be done using machine learning algorithms such as EM. Training examples from previous game logs will determine the likelihood of the parameters in the case of TAC.
- Selecting the proper action after obtaining a good estimate of the belief state is the next task. Methods such as Reinforcement Learning for both

MDP or POMDP environments are looked at. Benefits and costs of using relational representation in the learning case are examined.

- An overall guideline on how to define a framework for similar problems will be defined and the parts of the framework that will need the designer attention and the parts that will emerge automatically are identified.

References

[1] R. Arunachalam, N. Sadeh, J. Eriksson, N. Finne, and S. Janson. The supply chain management game for the trading agent competition 2004. *SICS Research Report*, 2004.

[2] M.E. Bratman and P. Intention. Practical Reason, 1987.

[3] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A short overview. In *Main Conference Net. ObjectDays*, volume 2004, pages 195–207, 2004.

[4] R. Brooks. A robust layered control system for a mobile robot. *IEEE journal of robotics and automation*, 2(1):14–23, 1986.

[5] John Collins, Wolfgang Ketter, Maria Gini, and Amrudin Agovic. Software architecture of the MinneTAC supply-chain trading agent. Technical Report 08-031, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, MN, 2008.

[6] M. Dastani, M. Birna Riemsdijk, and J.J. Meyer. Programming multi-agent systems in 3APL. *Multi-agent programming*, pages 39–67, 2005.

[7] J. Dix and Y. Zhang. IMPACT: a multi-agent framework with declarative semantics. *Multi-Agent Programming*, pages 69–94, 2005.

[8] N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents-summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*. Citeseer, 2001.

[9] M. Hutter. *Universal Artificial Intelligence:Sequential Decisions based on Algorithmic Probability*. Springer, 2005.

[10] H. Khosravi and B. Bina. A Survey on Statistical Relational Learning. *Advances in Artificial Intelligence*, pages 256–268, 2010.

[11] C. Kiekintveld, J. Miller, P.R. Jordan, and M.P. Wellman. Controlling a supply chain agent using value-based decomposition. In *Proceedings of the 7th ACM conference on Electronic commerce*, pages 208–217. ACM, 2006.

[12] J. Lloyd and K. Ng. Probabilistic and logical beliefs. *Languages, Methodologies and Development Tools for Multi-Agent Systems*, pages 19–36, 2008.

[13] KS Ng, JW Lloyd, and W.T.B. Uther. Probabilistic modelling, inference and learning using logical theories. *Annals of Mathematics and Artificial Intelligence*, 54(1):159–205, 2008.

[14] D. Pardoe and P. Stone. An autonomous agent for supply chain management. *Business Computing*, page 141, 2009.

[15] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2009.

[16] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[17] S. Thrun, W. Burgard, and D. Fox. Probabilistic robotics (intelligent robotics and autonomous agents), 2005.