The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

BDI PLAN UPDATE IN DYNAMIC ENVIRONMENTS

A Thesis in

Computer Science and Engineering

by

Anthony R. Fecteau

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2009

The thesis of Anthony R. Fecteau was reviewed and approved* by the following:

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

John Yen
Dean of Information Sciences and Technology
Professor of Computer Science and Engineering

Tracy Mullen
Professor of Information Sciences and Technology

Wang-Chien Lee
Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

Multi agent systems and agent oriented programming are becoming more popular each year and is even considered to be the successor to object oriented programming. Many agent frameworks implement various models of agency including one of the most popular models, known as Belief-Desire-Intention (BDI). The most researched part of the BDI model is the intention module which involves planning. The planning includes tasks that are run to satisfy a specific goal or condition that may be associated with the plan. The most common recent implementation of the intention module is known as the Conceptual Agent Notion (CAN) and its successor (CANPLAN). Some other implementations are JADE and SPARK agent frameworks. Most agent frameworks discuss updating only in the sense of belief updates about the environment. There is very little discussion about how to update a plan that is in the process of running when the environment changes. This type of plan updating is the focus of this thesis.

Most agent frameworks discuss plan updating in the context of conflict management, plan failure or plan aborting. This assumes that something has already gone wrong and that steps should be taken to fix the problem. Other methods of plan updating include an update when new important information is gathered by the agent. There are two methods for updating in the context of new information which are to update after the intention finishes and the other is to stop execution, update, and then restart. There are many examples of processes that effectively execute using one of the two updating methods.

However, these methods of updating are insufficient in many situations that come up in modern agents.

The method of plan updating that is introduced in this thesis is to update on the fly when new relevant information is gathered by the agent. Updating on the fly is different from the other two methods of updating in that the agent will pause execution of the intention, run update tasks and update bindings, and then continue execution of the intention. This thesis discusses in detail the full implementation of the BDI model used, as well as the new method of plan updating. An example showing the usefulness of updating on the fly is also discussed giving test data and results. The agent used is a driving simulation agent that drives from one place to another along a route and encounters traffic along the way. Using the implementation of the BDI architecture with plan updating on the fly the agent successfully updates the driving plan to take into account new information, which in this case is the traffic.

TABLE OF CONTENTS

List Of Figures

LIST OF SYMBOLS

P, P1, P2… Pn	Plan, Plan ‘1’, Plan ‘2’, Plan ‘n’
nil	Null, nothing, empty set
act	Action, can represent a task
φ or B |= φ	Logical formula over a set of beliefs ‘B’
+b or B ∪ {b}	Assertion of item ‘b’ into belief set ‘B’
−b or B \ {b}	Retraction of item ‘b’ from belief set ‘B’
!e	Occurrence of an event ‘e’
P1; P2	Sequential running of two plans
P1 ▷ P2	Run plan ‘P1’ then run plan ‘P2’ if ‘P1’ fails
P1 ‖ P2	Parallel running of two plans
φs or φf	Success condition and failure condition
ψ	Belief context condition

1 INTRODUCTION

1.1 Motivation

As computers are being used in more and more real world applications the need for systems that can handle continuously changing environments is becoming more important. Most systems previously were heavily reliant on domain specific information in a closed environment that is easy to monitor and keep track of. These systems rely on the fact that knowledge of the environment they are monitoring is completely within their set of beliefs. The problem with this is that if a piece of knowledge enters the system that it is incapable of handling, the system may fail or produce unwanted results. A system to detect anomalies in a machining process for example may have a goal of pointing out defects in a product. However, if the system is too specific, it may be unable to adjust according to increase in production or slowing down of assembly lines.

In more recent systems there have been advances in an agent's ability to handle dynamic environments. In the event that a goal experiences some sort of anomaly it is possible to choose a different method of satisfying that goal. These types of systems are more capable of adapting to dynamic environments where knowledge of the environment is not complete. It is possible for new facts to come in and be handled in a desirable manner. A system to navigate a car along a specific path may at some point run into construction or an accident that has the road shut down. With domain specific systems this sort of anomaly could easily halt progress towards the intended destination. This is a completely

undesirable execution of the goal to reach an endpoint for a trip. An agent system, upon receiving information from its sensors about construction, a traffic jam, or an accident, could either be designed to recalculate the route or to stay on route. A multi-agent system could go one step ahead and possibly receive information about obstructions to its route from other agents and avoid running into them altogether. The importance of these types of systems is that they are capable of changing their plans dynamically in order to satisfy a goal or sub-goal.

The next step in the development of intelligent agents is the adaption of knowledge bases towards completely dynamic data in the real world where absolutely no a priori knowledge is known. These systems have not been developed yet, therefore, systems must still have some knowledge about the environment they are working in. Simple knowledge about what a road is according to its attributes and different parts of the human body are still necessary for an agent to begin gathering information to be used.

1.2 Objective

In autonomous agents and multiagent systems there are two problems that are addressed within the contents of this paper. In these agent systems in the event of a change in the environment that causes an anomaly to a certain goal or plan the system must either halt the plan or let the plan continue until it finishes. Whether the agent stops execution of the plan or lets it finish, there may be cleanup efforts or rollbacks to be done. However, it may be much more desirable

to have the option of updating the plan according to the new information and continue execution. In this way the agent can salvage what has already been done and save time and effort by avoiding the process of executing costly cleanups and rollbacks, or having to run the plan from the start with the new information.

In the real world new information is constantly coming in. As sensors are able to gather more and more data about the dynamic environment agents must be able to handle this data. With more and more information to handle it becomes much more likely that a plan change or update is necessary. Instead of constantly halting execution or performing rollbacks it is much more desirable to finish executing plans with the new information taken into account.

There is also a problem of relevance of information and how it should affect a plan that is currently executing. This becomes a question of what is relevant to a plan or goal, how relevant is the new information, and how much information relevance should affect a plan change. For example, when a person is hungry they may want to go to the refrigerator and get some food to prepare. However, if there is already some prepared food in the kitchen when the person gets there, the person may want to change their plan if there is enough prepared food to satisfy hunger.

1.3 Chapter Overview

The main purpose of this thesis is to provide an overview and implementation of current methods of handling conflicts of plans in intelligent agents and multi agent systems through plan update, failure, and abort. Chapter 2 provides background information on agent systems and the BDI model. An overview of basic knowledge about intelligent agents is in 2.1, giving definitions of agent concepts including autonomous agents, multi agent systems, and agent communication. The BDI model of agency is described in 2.2 from Michael Bratman's work and the work of research dedicated to his "desire-belief" theory of intention.

The next two chapters discuss a more in depth look at an important topic of BDI, plan updating. Chapter 3 provides an overview of current work in the area. This includes the reasons for plan updating, which are conflict management and plan failure/aborting, discussed in sections 3.1 and 3.2 respectively. The chapter concludes with the current methods used for updating plans in section 3.3. Chapter 4 is focused on the implementation features for the BDI architecture as part of this thesis project. Section 4.1 involves an overview of the R-CAST agent framework which provides belief and desire modules for the BDI implementation. The ideas and motivation behind this thesis are given in section 4.2 with an example showing the necessity for a different type of plan updating. Section 4.3 explains the implementation of the BDI architecture and the proposed updating

method. This section contains a description of the partial implementation of CAN [24], which is used for the intentions module of the BDI architecture.

The thesis concludes with a discussion of test data used, results, and future work on the described project in chapter 5. Section 5.1 explains the test data that is contained in appendix A.1 and A.2. The test data used simulates a driving agent going from one city to another and uses the proposed update method. The results of the test data is in section 5.2, which shows the successful updating and completion of the running intention. The last section of chapter 5 lists various possible directions for future work involved with BDI planning and also with plan updating.

2 FUNDAMENTAL PRINCIPALS

An intelligent agent is an agent which has the capability to perceive information about its environment through the use of sensors and act upon its environment through the use of actuators. An intelligent agent should also have goals which it wants to achieve. Goals in a basic sense are specific states of the environment that the agent would like to move towards. The environment could be static, such as a factory, or dynamic, such as the world we live in. The environment can also be deterministic meaning that the effect of each action on each state is known or stochastic meaning that each state has a probability of going to another state based on each action. A deterministic world is not very realistic anymore and basically the solution to any goal can be reduced to a graph search algorithm [28]. Multi agent systems make the environment more dynamic and stochastic because there is more guessing involved in determining the state of other agents. This guessing can be reduced through the use of communication between agents with specified languages and ontologies [12].

2.1 Distributed Artificial Intelligence

2.1.1 Intelligent Agents

Before multi-agent systems and distributed artificial intelligence there were intelligent agent systems. There are many different types of intelligent agents based upon the capabilities programmed into them. Another term for an intelligent agent is a rational agent. A rational agent is an agent that receives information from the outside world by using any form of sensor and can act upon

that world by using any form of actuator [28]. In a sense, a human being is a rational agent, using sensors such as eyes, ears, touch, etc. and actuators such as speech and movement. Another characteristic of an intelligent agent is that it should attempt to optimize a performance measure. This performance measure is generally taken as an evaluation of how well the agent serves its purpose.

The simplest form of intelligent agent, which is a slight step up from conventional programming, is a reflex agent. A reflex agent has only the basic characteristics of an intelligent agent, in that it perceives its environment through its sensors, does any processing, and acts on its environment based on the new information through its actuators. There is no decision making process or actions based on prior information. In a very basic sense, a reflex agent is a pipeline from sensors to actuators where each piece of information has an associated action. The processing phase generally consists of any computations or communication that needs to be done. In terms of the different states that the world can be in, a reflex agent is only concerned with the current state.

Currently a large portion of research in artificial intelligence is focused on machine learning. An agent that has the ability to learn from the past and make decisions based on what it has learned is very useful. The general structure of a learning agent is similar to a reflex agent except that it will also collect information on what effect its actions had on the environment. The agent will positively reinforce the action for the particular belief set if the action had a

positive effect on the environment, and negatively reinforce for a negative effect. One of the famous algorithms used is the Bayes rule which is:

$$p(s_t|\theta_t) = \frac{p(\theta_t|s_t)p(s_t)}{p(\theta_t)}$$ [28]

This basically calculates the probability of achieving the state s_t given the action θ_t . This uses past information to calculate probability. In the best case the agent would depend on the state as being "a world state that summarizes all relevant information about the past" known as the Markov property [28]. Some other algorithms for optimal decision may make use neural networks. Neural Networks have nodes that do some computation on input data from other nodes or from the environment and pass the calculation as input to the next set of nodes in the network or as output to the environment. One of the most famous of the neural network algorithms is the back propagation algorithm which uses a performance metric to positively or negatively reinforce the nodes and links of the network. The back propagation algorithm tries to improve future decisions by changing the network after each decision based on how well it decided what action to take based on the previous decision.

In single agent systems the decision is based solely on its own actions and information. Generally speaking, an agent has its own set of knowledge that it can do computations on based on logic and inferencing. The agent will gain more information from its sensors but the agent itself decides what to do with this information, whether it is to assert directly to the knowledge base, infer more information based on a set of logical rules, take action upon the new information,

or any combination of these. There is also the concept of an autonomous agent which bases its decisions entirely on perception instead of prior knowledge. There is an increasing desire for autonomous agents because of their ability to adapt to different environments and fulfill different requirements without much reprogramming.

2.1.2 Multi Agent Systems

A multi agent system is a system in which many agents are interacting with each other. In most cases agents are working towards a common goal but there is also the situation where agents are competing with each other. There are many advantages to using multi agent systems over simply using a single intelligent agent. As with any distributed system a multi agent system provides a decentralized, reliable, scalable, and flexible solution. Each agent in a multi agent system is a standalone entity in that each agent is a peer. A multi agent system isn't a server client system, instead it is more of a peer-to-peer system in that each agent has the same responsibilities when it comes to keeping the entire system running.

A multi agent system is an ideal solution to many problems because they are reliable, scalable, and flexible. In the case where a single agent in the system fails, the other agents will be able to continue operating with very little impact. Also, multi agent systems scale very well because no single agent has more responsibility to the system than any other agent. With this characteristic the

only real impact on the system itself is in message passing between agents. The flexibility of a multi agent system is that a single agent can be taken out of the system if any changes or upgrades need to be made and the system will continue to function.

Another quality of multi agent systems is that they allow both asynchronous and parallel execution. This means that each agent can be run on completely different machines with message passing going on whenever necessary. An agent can run on its own machine and do its computations at the same time that another agent in the system is running on its own machine and doing its own computations. In this way, each agent is not reliant on the execution of other agents. The agents are also asynchronous in that they may send or receive a message at any time, they don't need to stop and wait for messages or stop and wait for a response from another agent. However, communication between agents requires both a communication language and also knowledge of the ontologies of other agents, both of which are described in the next subsection.

An agent in a multi agent system makes decisions based on much more than just its perception of the environment. Agents must also take into account the knowledge of other agents and also the action of other agents. The two different types of multi agent systems are homogenous and heterogeneous [28]. The former describes a system in which all agents in the system are designed to be exactly the same. This isn't to say that they won't be different at runtime,

because as an agent gathers data and makes decisions based on its own knowledge base differences will begin to develop. An example of homogeneous agents is a cleaning system for a large banquet hall. The agents in this system may all be programmed with the ability to clean and develop regions that become dirty the quickest. However, after a certain amount of runs, each agent will have a different set of regions that it monitors. On the other hand a multi agent system can also be one in which some or all of the agents are programmed a priori to behave differently. An example of this type of system is robocup, one of the most popular testing grounds for multi agent system algorithms. All agents have the ability to communicate about where the ball is and also the ability to "kick" the ball. However, there must also be a separate set of logic for the goalie, defense, and offense positions, otherwise all of the agents would either all be attacking the ball or defending the goal.

Another important concept within multi agent systems is the idea of allies, opponents, and neutral agents. Many issues arise within each of these areas. In a team of agents working together as allies there is a problem of how to coordinate the entire team. In robocup the team must be able to communicate such things as location of the ball, position of each agent, perceived position of opponents, and if anybody currently has possession of the ball. In game theory agents may have opponents that they must communicate with. The problem here is that opponents may lie, so how can one agent trust another agent. Even

in coalitions where teams of agents are connected but each agent has its own set of goals there is the issue of trust.

2.1.3 Agent Communication

There are many factors to take into account when communicating between agents in a multi agent system. Some of these factors include ontologies, languages, and transport protocols. The last of the three, transport protocols, is a simple decision that is a choice between such protocols as TCP or UDP [8]. This is basically a decision of network structure and doesn't really have much to do with agent design. However, the ontology and language used by the agents is much more important and it has many requirements.

An agent communication language is an agreement between agents on the structure and meaning of content of messages. The most common agent communication language used currently is the Knowledge Query and Manipulation Language (KQML) [9, 8, 10]. This language has a specification for structure of messages between agents and more importantly, a definition of message types to be passed between agents. Examples of this include tell, insert, delete, evaluate, reply, ask, stream, advertise, subscribe, forward, broker, and many more. An implementation of KQML need not include all of these message types but the most important of these are the insert, delete, and ask messages which are used for the most basic knowledge exchange. Some messages are even used to gain assistance from other agents on inferring

knowledge that may help towards a goal. Another agent communication language, which is similar to KQML, is the Agent Communication Language (ACL) [25].

An ontology is different from a communication language in that an ontology generally describes the information that is being passed between agents. Two popular ontologies currently used in multi agent systems are predicate calculus and more recently the Resource Description Framework [15]. The latter is commonly referred to as RDF triples because the format used is subject, predicate, object. Agents must also agree on how knowledge is represented using the ontology. If one agent describes a car with the attributes: motor, color and another agent describes a car with the attributes: color, motor it may be difficult for the two agents to communicate a yellow car with a V6 engine without getting confused.

One way for an agent to effectively communicate knowledge to another agent is to design a natural language to go along with the ontology. A natural language is basically a template for knowledge to be represented in the way that human beings would communicate the knowledge. Therefore, an agent can say the yellow car has a v6 and a human being can understand it much more easily or another agent can parse the useful information.

2.2 Belief – Desire – Intention Model

The “desire-belief theory of intention” [2] was developed by Michael Bratman in the late 1980’s in a series of theses discussed in his book “Intention, Plans, and Practical Reason”. This book is the basis for the research that started BDI architecture in intelligent agent systems in artificial intelligence. Bratman discusses the attributes of a planning agent that are still used in agent systems today. These attributes include the ability to act purposively and the ability to form and execute plans. An agent also has the ability to coordinate present and future activities, which brings up coordination as a major theme in agent related research.

2.2.1 The BDI Model

Intelligent agent systems have much of the same characteristics of agents described in the book by Bratman [2]. Figure 2.1 shows a chart mapping Bratman’s BDI model to the BDI model implementation in intelligent agents.

Philosophy:	Belief	Desire	Intention
Theory:	Belief	Goal	Intention
Implementation:	Relational DB (or arbitrary object)	Event	Running Plan

Figure 2.1: BDI Mapping - A description of the relationship between the different interpretations of the BDI model. [29]

The first part of the BDI model is the set of beliefs that an agent holds in any state. These beliefs are the set of information about the environment. From an

implementation standpoint beliefs are basically the data in a relational database or instances of objects in an object oriented language. The latter is the more common case where beliefs are held as facts in a knowledge base. Beliefs can come from three different sources. The first and simplest of the three is perception which describes information coming in through the sensors that the agent has. The second is communication from other agents which can be either other intelligent agent systems or human beings. The third source of beliefs is through contemplation within the agent. An agent can have built in logic known as rules which the agent can use to infer new information from previous beliefs.

The next part of the BDI model is the desires of the agent. As shown in figure 2.1 desires can be the goals the agent is trying to achieve or a specific event that the agent is trying to make occur. In a very basic sense the desires of an agent can be as simple as a set of facts that the agent wants to make part of the current state of the environment. This set of facts can represent a state of the environment such as “location = library” or they can also represent an event that the agent wants to take place such as “checking out book”. An agent may have multiple goals which it wants to achieve. These goals can be activated in different ways such as incoming facts (declarative), event triggered (procedural), or simply a goal initiated at startup [23]. Declarative goals can be further broken down into perform goals, achieve goals, and maintain goals [3].

One of the most important parts of the BDI model is the ability of an agent to intend to do something. Agent theory in philosophy describes an intention as a plan that the agent intends to execute in the future [2]. An intelligent agent may have an intention base to store the current intentions of the agent to satisfy particular goals that are active [23]. It is desirable to attempt to discover conflicts between plans in the intention base before committing to either the intention or the plan itself. Once a plan is dispatched from the intention base and is in the running state it is difficult to handle conflicts because the agent must execute expensive cleanup efforts and rollbacks [27].

There is also an extension to the BDI model that is proposed by Hong Jiang et al [13]. This extension is known as the Emotional BDI model (EBDI). The authors state that beliefs come from perception, contemplation, and communication. The emotion represented by the agent attempts to break down into two categories the way in which beliefs are handled. The first is primary emotion which is the initial reaction to any situation, also called instinct. Primary emotion corresponds to perception about the environment and communication with other agents, which could be allies (truthful) or enemies (may lie). The other type is secondary emotion which results from further deliberation about facts and can replace primary emotion. In a sense, secondary emotion is used to determine how the agent “feels” about the environment. For example, if another agent is truthful about what they know about the environment then the emotion towards that agent will be in a thankful state. The emotional state of one agent towards

another can help determine if the knowledge exchange should be trusted and therefore have a higher priority. If another agent lies about what they know about the environment then the emotion towards that agent will be anger.

2.2.2 The BDI Language

The BDI language is broken up into three parts consisting of the Beliefs, Desires, and Intentions. The first two are a much smaller portion of the language while the intentions portion is where the bulk of research is. The implementation used in this thesis is described in the specification of Conceptual Agent Notion (CAN) [30] and later refined and expanded to CANPLAN [24]. Other examples of BDI implementations are the JADE framework [1] and the SPARK agent framework [18]. The beliefs part of the language consists of (1) adding facts, (2) retracting facts and (3) inferring new facts from existing facts. Adding facts is represented by a set union between the (B)elief base and the (b)eliefs to be asserted. Retracting facts is represented by a set subtraction between the (B)elief base and the (b)eliefs to be retracted. Inferencing is represented by a logical formula such that the facts φ follow from a (B)elief set.

(1) $B \cup \{b\}$ (assertion)

(2) $B \setminus \{b\}$ (retraction)

(3) $B \models \varphi$ (logical formula, ie – rule)

The desires part of the language consists of handling the two types of goals. The first goal type is known as (4) event driven, which is satisfied upon the occurrence of a specified event, e. The other goal type is known as (5)

declarative, which is satisfied upon the completion of a plan, P. The latter type is more common in BDI agents due to the intention driven nature of the system.

(4) Goal(φs, !e, φf) (event driven)

(5) Goal(φs, P, φf) (declarative)

The intention part of the language can be represented by the following grammar:

(6) P ::= nil | act | ?φ | +b | −b | !e | P1; P2 | P1 ▷ P2 | P1 ‖ P2 | Goal(φs, P1, φf) | {ψ1 : P1, . . . ,ψn : Pn}

This states that a plan P can be nil, or nothing, it can be an action, also known as a task, or many other more complex components of a plan. Some of the previously discussed actions are assert into the belief set (+b), retract from the belief set (-b), and trigger an event (!e). Some components that allow multiple components to exist in a single plan are sequential running (P1 ; P2), which means to run P1 then run P2, also running two components in parallel (P1 ‖ P2). Another type of compound plan is to run P1 then run P2 only if P1 fails, which is different from the other two compound plans (P1 ▷ P2). Two of the more complex plan types are to check a logical formula over the belief set (?φ) and to run a single plan out of a set of plans based on a belief context condition.

(7) {ψ1 : P1, . . . ,ψn : Pn}

The second of the complex plan types is used when the agent needs to choose a plan (P1, P2… Pn) based on a set of conditions (ψ1, ψ2… ψn). The last type of plan is a declarative goal which will run the specified plan repeatedly until either the success condition (φs) or failure condition (φf) is satisfied. One of the most important aspects of the CANPLAN model is the description of how to start running a plan (8).

(8) e : ψ ← P

This describes an initialization of a plan such that in the occurrence of an event e plan P should execute if condition ψ is true.

2.2.3 Plan Update, Failure, and Abort

The main goal of this thesis is to review current methods of handling conflicts of plans through the means of updating, failing, or aborting a plan. At first glance the distinction between the latter two may not be very clear. A plan update may occur because of incoming data that may change the parameters of a plan that is in the intention base or currently running. This may not necessarily be a bad thing for the agent and may not require any sort of cleanup or rollback.

On the other hand, a plan failure is completely undesirable for an intelligent agent. Most of the time a plan failure will cause an abrupt end to a running plan because there is no need to continue running a plan that has failed. A plan failure may not necessarily be caused by the plan but instead may be caused by the agent or other outside forces. In single agent systems a plan failure may be caused by multiple plans running concurrently. A sub plan may come in direct conflict with the parent plan that it came from or even the goal that it came from. In the worst case a plan may come in direct conflict with a global goal of the agent. There are many other causes of conflict between plans when dealing with multi agent systems. In multi agent systems where resources are shared between agents there may be a resource conflict between agents in which one of

the agents will have to fail their running plan so the other agent will have access to the resource [20]. In a multi agent system in which a team of agents are working together, one agent may inform another agent that a piece of information they have discovered will cause their running plan to fail.

A plan abort is somewhat of a cross between the need to update a plan and a plan failure. In a sense, plan abort most of the time is not undesirable but it will cause an abrupt halt in the plan. An example of this is provided in [30]; if a cat is trying to acquire food on the table and jumps on a shelf to get to the table but finds food on the shelf, the cat aborts the task of jumping to the table. Note here that the overall goal to acquire food is still satisfied, however, the plan to jump from floor to shelf to table no longer needs to be completed. In this example there is no cleanup effort necessary. In a different example where a person begins to make food and is later informed that there is already food prepared, the person should need to put supplies away before eating the already prepared food.

3 OVERVIEW OF APPROACHES

The theory of planning and formulating intentions towards the achievement of desires or goals was first introduced by Michael Bratman in 1987 [2]. In his book on Intentions, Plans, and Practical Reason he discusses many theories that are still cited in many recent papers on BDI architecture. The theories are based on human reasoning processes but are quite relevant to agent systems. Most notably are the theories about reasoning using planning based on the specific context of a situation. The planning and context together are used to satisfy a goal that the human is currently working to achieve. These three concepts of context, goals, and planning are still used today in agent systems as Beliefs, Desires, and Intentions. Most agent frameworks based on the BDI architecture attempt to function similarly to the human cognitive process to solve complex problems. One of the first attempts to connect Bratman's work to intelligent agents using the BDI architecture was done by Rao & Georgeff in 1991 [21].

Each of the three areas of the BDI architecture have a significant amount of research dedicated to them. However, the most complex of the three is the idea of how to go about accomplishing goals through the use of planning and intentions. Many papers have been written on how to handle changes in the environment while intentions are executing. Although some research has been done in the area of updating goals [4] most research in updating focuses on planning. Some of the topics that are relevant to the topic of dynamic environments are conflict management, plan updating, plan aborting and plan

failure. The two latter topics are somewhat similar but have important differences.

3.1 Conflict Management

An agent may have conflicting intentions due to the ability to run multiple intentions in parallel. The common method for handling conflicts between intentions is to have an associated priority or payoff for each goal [20]. This problem can occur in cases where the agent is running two intentions that are conflicting and also cases where two agents are each running intentions that are conflicting. In the latter case it is common to have a resource agent that determines which agent can access which resource at any given time. When a conflict occurs the agent needs to know what to do to handle the particular situation.

An example of a resource conflict is discussed in the Agent Oriented Software Engineering (AOSE) paper [17]. The authors describe a situation in which an airline booking agent needs to access a database for information and booking purposes. The problem here arises when multiple airline booking agents require access to the same ticketing data at the same time. To solve the problem there should be a resource allocation agent that handles database accesses from each booking agent. There will be a resource conflict if two agents find a flight with one seat left and then both attempt to book the flight. One agent will be able to

successfully book the flight and the other agent must resolve the conflict in some way.

3.2 Plan Failure and Plan Aborting

A running intention may fail or need to abort for many different reasons. However, plan failure and plan aborting are quite different from each other. An intention may fail when trying to check out a book at the library, for example, because the book is not there. On the other hand, the same intention to check out a book may need to be aborted because on the way to the library your friend calls and tells you that he has a copy that you can borrow. Plan aborting occurs when the agent finds a more desirable solution to achieve success whereas plan failure occurs when the agent cannot find any solution to achieve success [30]. This thesis is focused more on the latter, such that an agent should attempt to utilize new information to avoid failure. Updating is the utilization of information that has become part of the agent's knowledge base after an intention has already begun to run.

3.3 Plan Updating

There are two types of plan updating currently being used in agent systems. These types of updating are update after the plan has finished and stop the plan then update. The type of updating presented by this thesis is a third type such that the plan pauses execution, updates, then continues execution. In the case of the "Rockwell Automation Agents for Manufacturing" [16] the agents use a sort

of finish and update. Upon failure of the system the agents will negotiate with each other to find a solution. It is important to note here that failure of the manufacturing process system does not mean failure of a running intention. Therefore, the agents will recognize a change in the environment corresponding to the mechanical failure and realize that an update is needed. In this case the agents have finished executing their current plan and communicate to devise a plan taking into account the new information.

The second type of updating causes the agent to stop execution of the plan and start over with the new information. An example of this type of updating is discussed in Rana et al [20]. In this case there are application agents and resource agents that communicate with each other in order to complete intentions. However, if two agents need the same resource at the same time there will be a conflict that the resource agent must resolve. The resource agent reports the conflict back to the losing agents and they must handle the conflict. At this point the running intention has not failed because the resource may become available after a period of time. In this case the agent will stop execution of the intention, update, and then restart the intention with the new information.

In both cases discussed above the update method used may be sufficient. However, as discussed in the following section, there are cases in which the update must occur during execution of an intention. For updating on the fly the agent will pause execution temporarily in order to execute any update tasks and

update bindings associated with the intention, and then continue executing the intention.

4 MODIFICATION TO APPROACHES

4.1 Intro to R-CAST Framework

The focus of my work is to improve the R-CAST framework by adding a module based on the BDI agent model and improving upon the method of updating plans. The current framework of R-CAST includes the main modules, known as, the knowledge base, communication module, information manager, recognition primed decision, and the request whiteboard [32]. With the exception of the request whiteboard any module can be configured to be included for any agent. R-CAST is used for making decisions based on teamwork and collaborating in order to gather information. On the other hand, BDI is used for executing plans in order to satisfy global goals that are specified to the agent prior to runtime.

An R-CAST agent may use any combination of the main framework modules with the addition of many other provided modules. One of the most important modules used in R-CAST is the knowledge base [33]. All of the facts and other information are stored in the knowledge base to be used for processing by other modules. There is also rule based processing done in the knowledge base used to infer more facts and information. The knowledge base uses a Multi-Agent Logic Language for Encoding Teamwork (MALLET) for representing information [7]. The communication module and information manager also use MALLET for transmitting knowledge to other agents and also to other modules within an agent.

The Recognition Primed Decision (RPD) module is used to making somewhat quick or hurried decisions based on decisions made in the past by the agent or other agents [6]. The purpose of this is to more accurately simulate human decision making in real-time applications. The whiteboard is also used for simulating human interaction. All requests for information by other agents are posted to the whiteboard for tracking purposes. The whiteboard records requests by the current agent and other agents and tracks progress towards fulfilling the requests.

As part of the R-CAST framework there is a GUI used for displaying the current state of an agent to a human user. An agent can run completely autonomously, which means no outside help is needed for the agent to function. Although an agent is fully capable of running autonomously, a human user may modify the state of an agent by asserting or retracting information from the knowledge base through the GUI.

4.2 Ideas/Motivation

Current implementations of BDI architecture are lacking in the ability to handle dynamic environments in an efficient way. One way to handle an update in the environment is to halt processing immediately and restart with the new information. The other is to wait until processing has finished and then restart with the new information. Selecting either of these solutions in many situations doesn't make sense. Consider an example in which you need to go to the store

to purchase a book. You would start at home, then get in your car, then begin driving to the store. But what happens if you remember that you left your bookstore discount card in your desk at the office. With the first solution to updates you would stop driving, then go home, then update your plan, then go to your office, then to the bookstore. The second solution is similarly inefficient. You would first finish driving to the bookstore, then update your plan, then drive to the office, then drive back to the bookstore.

An agent that is able to update on the fly would be much more desirable than either solution previously discussed. In this case the agent would remember the discount card, delete all tasks left to get to the bookstore, go to the office, then to the bookstore. In the situation where the office is right next to either the bookstore or your home there wouldn't be much difference than updating on the fly. However, in the situation where you remember your store discount card because you drive by your office, dynamically updating is most desirable.

4.3 Methods and Implementation – BDI Framework

The Knowledge Base module can be used as part of the BDI model for information storage. Other modules in the BDI model can process information into the knowledge base and also query information for use in processing. For the implementation of the BDI framework into the existing R-CAST GUI the system will automatically place the Knowledge Base module into the Beliefs tab

of the BDI module. In this way there will be a convenient place to view all three modules of the BDI framework.

The desires module of the BDI framework is not needed for the purposes of this research project. However, for a more complete BDI agent, the desires module will be added into the project in the future. Currently, it is possible to add in either of the already available goal driven processing modules, RPD and Recommender. Although these modules provide slightly different methods of goal driven processing they can still fulfill the desires part of the BDI model.

The R-CAST agent framework provides a very nice model for creating and dropping in new modules. The BDI module was created as a completely separate package and added as part of the R-CAST configuration properties. The R-CAST agent framework also provides a convenient model for adding new modules into the GUI. Figure 4.1 shows a modified version of the bookstore driving agent described above. The agent shown has the required Request Whiteboard module and the Belief-Desire-Intention module. The Intentions module GUI has functionality that supports interaction from a human user, as well as a convenient display of the current operating state of the agent. The button at the bottom of the intentions module GUI allows the human user to switch between viewing the available plans of the agent and the currently running intentions of the agent. In the available plans view a human user can manually activate any given plan.

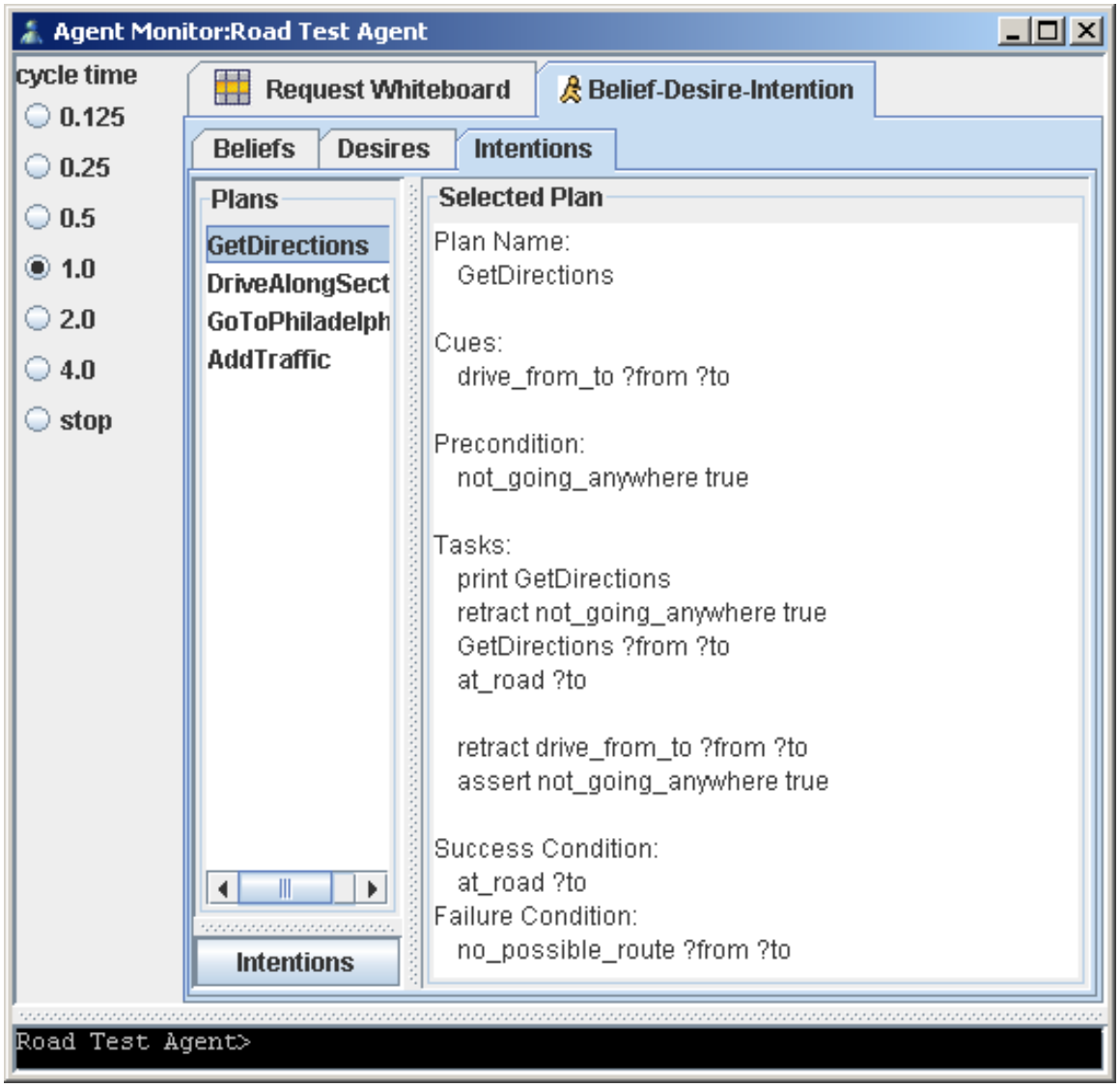

Figure 4.1: A screen shot of the R-CAST GUI with the BDI module. The two plans, GoToPhiladelphia and AddTraffic, are activated by the user in order to begin processing and to show an example of updating.

The plan will be added to the list of running intentions if the preconditions are satisfied for the plan, otherwise it will be rejected and no further action will be taken to start running the intention. On the other hand, in the intentions view

there will be a list of currently running intentions. A human user has the ability to abort a currently running intention, in which case the intention will begin a rollback. The agent will still be able to autonomously add or abort plans or intentions based on cues and fail conditions, but there are some situations where it is desirable to have human user interaction.

One of the important concepts dealing with plan and intention processing is the idea of variable bindings. This subject is not discussed to a significant extent in literature, however, variable binding is necessary for plan processing and updating to occur. Variable binding allows the variable ?x in the precondition to correspond to the same variable ?x in the task list. Each task associated with an intention may read from the binding list and use the values to do processing. However, a task should not update variables bindings because all further processing is based on the initial values of the variable bindings. The only reason variable bindings should ever be changed is when an update in the dynamic environment occurs in a way which changes the way the intentions should continue to be processed. In this way the use of variable binding is also very useful for updating intentions on the fly.

Another important part of dynamic intention updating, and agent processing in general, is the use of steps within tasks. In the implementation for this project a task is required to have a method known as a step. A step is an intermediate portion of a task that can be broken up to form a smaller part of a task, but is not

large enough to be considered a task itself. An example of a task is to get directions to a place in the form of smaller sections of road. On the other hand, examples of steps for this particular task are lookup directions, create road sections, and import road sections into the knowledge base. These are all steps because they are small portions of this task that are always required to complete the get directions task. They are single units of execution whereas the get directions task is a collection of multiple units of executions.

4.3.1 Importance of Task Steps

There are many reasons for using a step style of execution in agent programming, other than to simply break up a task. It may seem difficult at first to understand the difference between a task and a step. A task is created for reusability within an agent. A step is used for separating simple points of execution into trivial processing units. The problem with using a step instead of a task is that there will be very many steps listed in the plan and it may be difficult or inconvenient for a human user to program or view how the agent is running. Another reason for using tasks in a plan is that generally a grouping of steps is always used for a particular task so it is better to simplify how the agent is programmed by writing a task once and using it multiple times in various plans. This brings about the issue of overloaded or over-customized tasks versus too small or too simple of tasks. However, when considering a plan in an agent, it should be fairly intuitive to decide where to draw the line between tasks and steps.

Step style execution is not only useful is breaking up tasks but it is also useful in updating, doing rollbacks, and simulating human plan execution. In the situation where multiple plans are running simultaneously there is a task dispatcher that randomly selects a task, then runs that task and selects another task and so on. However, with step style execution it is possible to take this further so that the task dispatcher selects a task at random and runs a single step of that task, then selects another and so on. In this way it is possible to further interleave the execution of multiple plans such as walking to the grocery store and deciding what to cook for dinner both at the same time. The larger tasks are broken up into shorter steps so that the task dispatcher doesn't get stuck executing a long task that would otherwise be broken up into tens of smaller steps.

In a dynamic environment when a change occurs that causes the agent to fail a particular intention it is much more desirable to use step style execution. In the case where a task is broken up into smaller steps, there is a sort of tracking of the progress of the task. This is very important when there is an update that causes the intention to fail because the agent only needs to roll back the steps that happened. Consider the example of asserting a fact into the knowledge base. It is easy to see how this single task only requires a single step of execution. However, it can be broken up in a way that helps significantly with rolling back if a failure occurs. For this reason, the assert task should be broken up into the steps of (1) build the fact, (2) check knowledge base connection, (3)

add fact to knowledge base, and finally (4) check if fact is now in the knowledge base. An agent may be executing an assert task at the second step and the plan may fail. Without step style execution there is unnecessary rollbacks that occur such as retracting the fact. However, if the agent knows that it is executing the second step of the task and has not yet made it to the third step then there is no need to retract because the fact has not yet been added to the knowledge base.

To manage the rollback procedure for an intention it is necessary to keep track of which tasks have run already and which tasks have not. Each task however must keep track of which steps have run already and how to handle rollback depending on which steps have been executed. To store information about which tasks have run already there is a rollback stack. Upon completion of a task the agent will move the current task onto the top of the rollback stack and then grab the next task from the end of the task queue. The rollback procedure of each task is determined by the user that defines the task. The user may choose to have a single rollback procedure that is not dependent upon the current step. On the other hand the user may choose to do full tracking of steps and define a rollback procedure that is fully dependent upon the current step. Either way, the agent will begin the rollback procedure after the current step finishes and execute the rollback procedure of each task that has run in reverse order.

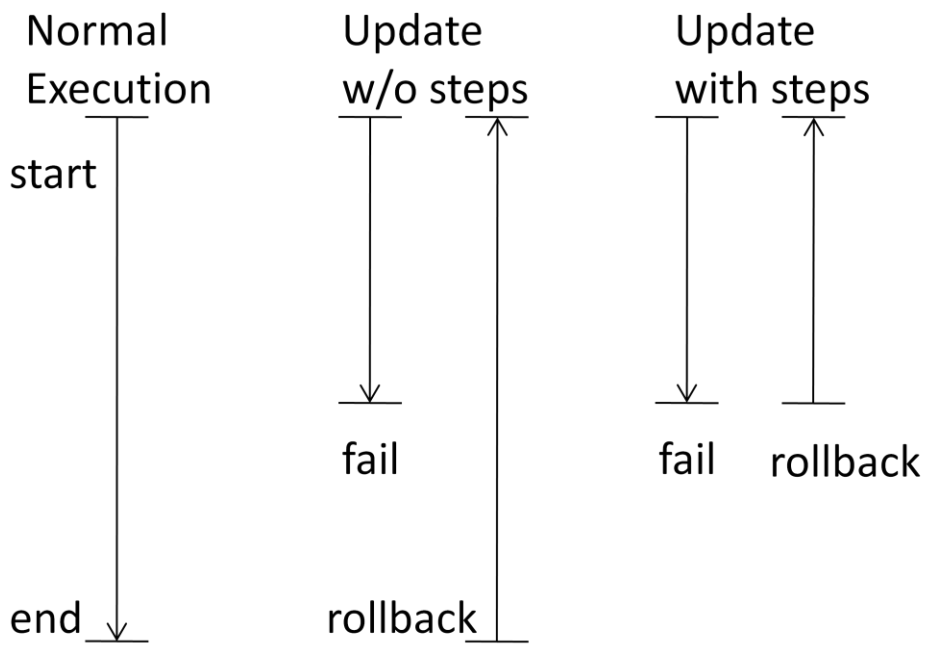

Figure 4.2: Failure and rollback in comparing step style execution versus normal task execution without steps.

Step style execution also allows for faster and more accurate intention updating. Generally with normal task execution the check for an update occurs after an intention task has finished executing or before the next task in that particular intention is about to run. The process of updating an intention may take a very long time to occur if the task that is currently running is a large task. However, if the task is very large but broken up into many significantly smaller steps then the intention will update either after the step finishes or before the next step starts. This style of execution is much more representative of human processing and is also much more efficient for running intentions.

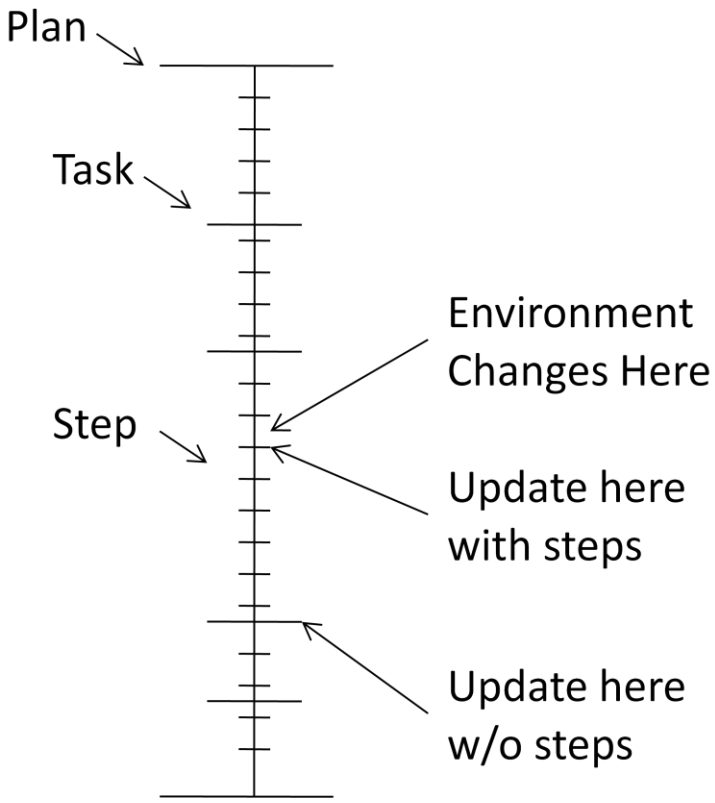

Figure 4.3: Dynamic environment change when comparing step style execution versus normal task execution without steps.

4.3.2 Dynamic Plan Execution Through Updating

The bulk of the work for this project is for updating a plan in a dynamic environment. The current methods of updating an intention are inefficient and in most cases don't make sense in their context. One method of updating is to stop execution of the intention immediately upon recognizing a changed environment and restart based on new information. The other method of updating is to finish execution and then restart based on the new information. Neither of these methods will work well for the example of driving to the bookstore and remembering your discount card in transit. In this case it is important to

introduce the method of updating such that the intention is updated on the fly and the intention continues to execute from its current state. This method for updating may seem simple at first but a fully useful implementation becomes rather involved. In some cases it is enough to update the current bindings and continue execution with the new data. However, this may not be enough for an agent to properly adjust its current state to a more desirable state based on new data. In the example of driving to the bookstore, the agent needs to do more than to update its destination. In the agent's current state there may already be a route that will take the agent to the bookstore. So it will be necessary to run tasks to retract that route data and to create a new route.

When the dynamic environment changes and an intention update is required, sometimes it is necessary to modify a binding to handle the change. For example, if you are driving to the bookstore and find out that the bookstore you are going to is closed today, you should update to a different bookstore. In this example where ?x corresponds to the bookstore, the agent should change ?x from "Bob's bookstore" to "Jim's book outlet".

The other necessary part of dynamically updating an intention is to drop new tasks into the current state of the intention. It is desirable to pause the current task after the completion of the current step then drop in some update tasks at the front of the task list. Using this method the agent has the ability to do necessary processing in order to handle incoming information. An example of a

use of dynamic updating is a driving agent that gets knowledge of traffic. The agent should first check if the traffic is along the current route to the destination. If the section of traffic exists within the current route then the agent should get directions from its current location to the destination taking into account the speed of traffic along the section of road. The processing done here is to first remove all current data about the directions to be taken. The next step is to update the bindings about where the starting point for getting directions is. Then the agent should get new directions using information about the new traffic. After these steps are taken the agent can continue with the execution of the intention.

Another smaller but significant part of the update process is to ensure that the running intention doesn't try to update again before the update has completed. This case may come up because a fact that makes the update condition true has not been retracted yet. What will happen in this case is the update logic will run again, which drops the update tasks into the intention again. This has the chance of creating an infinite update loop where the update keeps executing and the task list keeps growing. To void the infinite update loop problem the agent should disable the update function for the intention and add an UpdateCompletedTask. This task should be added after all of the update tasks but before the task that was running when the update began. It is important to note that this will not prevent the infinite update loop entirely. Instead, the purpose of this extra logic is to allow the agent to run tasks to retract facts or do other necessary tasks to stop the same update from executing again. However,

it is the responsibility of the agent developer to add the tasks to prevent the infinite update loop problem.

The final part of an update is the condition or conditions to check for. An update condition is similar to an intention precondition or postcondition. Therefore, the full process of running an update using step style execution is to check the update condition, change bindings, run update tasks, and continue execution. This processing is all done within an update dispatcher. When a plan becomes an intention it will be checked for the existence of update conditions. These conditions are added to the update dispatcher when the intention begins and is removed when the intention finishes. The update dispatcher is similar to the task dispatcher in that it cycles through the update conditions at random and executes an update if the condition is true. The intention dispatcher and the task dispatcher are allowed to run separately and will continue to function while an update is taking place. The first event that takes place in an update is to create a busy task for the intention to stop any tasks to be run during the update. Using this process an agent can quickly and efficiently handle dynamic environments.

5 FINAL EVALUATION

5.1 Test Data

The driving agent is used to show the capabilities of the update on the fly functionality described in this thesis. The purpose of the agent is to “guide” a human user to a specified destination using execution of plans in the plan base. The agent is also able to handle update of intentions on the fly to change directions in the event of overwhelming traffic conditions. The agent stops executing intentions and sits in idle state once the destination is reached. This thesis is in no way meant to discuss mapping algorithms or GPS technology so these functions are simulated as described below. The plan data found in the “roadtest.xml” file is shown in Appendix A.1. Also, the knowledge base representation found in the “roadtest.kb” file is shown in Appendix A.2. Both of these files are described in detail below.

The driving agent has an initial location that it starts at which is hard coded into the knowledge base at start time. The initial location for the agent is “state_college” which is shown by the “at_road” fact type in the knowledge base. This is updated as the agent “traverses” the route. Initially there is no route specified to the agent, instead, there is only knowledge of the initial location and a fact to specify that the agent is “not_going_anywhere”. These facts are necessary for the agent to start going somewhere.

The bulk of the agent execution takes place when the user decides he/she wants to “DriveToPhiladelphia”. To manually begin running a task a user can double click on a plan in the plan tab. However, the user must note that double clicking on an intention in the intention tab will cause that intention to abort. Therefore, care must be taken to ensure that the plan tab is in view when the user wants to manually add an intention. The DriveToPhiladelphia intention will trigger the plan, “GetDirections”, to start. This plan simulates adding sections of roads that will create a route from the current location to the destination. In a real driving agent this would be done by some sort of mapping software, but for the purposes of this thesis the route is simply hard coded into the task for getting directions. In the task for getting directions there are individual steps for asserting each section of the route into the knowledge base.

The plan used for traversing a route is called “DriveAlongSection”. The agent will begin, execute, and end an instance of this intention once for every section of the route. The agent begins an instance of this intention when it has a section of the route that begins at the current location. The DriveAlongSection intention simulates driving by adding blank tasks that take a small amount of time to complete. The blank “busy” tasks exist so that the agent doesn’t reach its destination too quickly for the user to cause an update. In a real driving agent, in the DriveAlongSection intention the blank tasks would be replaced by an “UntilTrue” task that waits for a specified condition to be true before the intention can continue.

Once the agent arrives at Philadelphia the DriveToPhiladelphia intention will complete. The agent will again be in an idle state. The update condition is removed from the update dispatcher and the intention is removed from the intention base. This represents normal operation of the driving agent that is going from a starting location to its destination. What happens if something happens while the agent is driving? For example, a major traffic jam could cause the speed of ten miles of road to be reduced from 45 miles per hour to 5 miles per hour.

Part of the simulation for the driving agent is to add traffic to the last section of the route. The intention to do this, “AddTraffic”, is another user activated intention that will only run if the user double clicks on it in the plan tab. When the update dispatcher runs it will find that the conditions for DriveToPhiladelphia update are true. The agent will run the necessary update tasks and change any necessary bindings to continue going to the destination. The update tasks in this case are to remove all sections of the current route, get directions with the new information. The agent also needs to update bindings to show where the agent is at so when new directions are retrieved the agent won’t be advised to start back in State College. After all update tasks and bindings are completed executing the agent will run the UpdateCompletedTask to inform the agent to continue running the intention in a normal state.

5.2 Results

The driving agent operates as expected using the R-CAST agent framework with BDI module. When the agent first begins running there will be two facts in the knowledge base and the agent will idle. Although the agent does not seem to be doing anything the BDI framework is constantly checking the cues of plans for any matching bindings in the knowledge base. As shown in figure 4.1 the user has the ability to bypass the cues by double clicking on a plan. However, the agent will not execute GetDirections unless the user first activates the intention to GoToPhiladelphia. This occurs because even though the cues were bypassed, the preconditions must always be satisfied for an intention to start. In this case the precondition checks if not_going_anywhere is equal to true, which is asserted by GoToPhiladelphia.

In the normal operation of the driving agent the user starts GoToPhiladelphia and the intention begins as expected. Each intention prints data about its current operating state as part of the task list as an informative to the user. The agent simulates driving to Philadelphia by printing a statement for each section of the route and when the agent reaches the destination. The scenario in which traffic is “added” into the environment also executes as expected. The user may start the AddTraffic task at any time after GetDirections is finished. This is necessary because if the user is still getting directions based on data prior to the traffic and also tries to get directions based on new data there will be problems with the route. Outside of the simulation environment this isn’t an issue because

GetDirections executes rather quickly and it is unlikely that an agent will get traffic data immediately after starting to get directions. When the user adds traffic the agent updates the route as expected and continues on its way.

5.3 Conclusion and Future Work

This project shows a new method of updating a plan in an agent framework. The method for updating involves updating on the fly when important data is discovered by the agent. The example of driving from one place to another when high volumes of traffic are present shows the usefulness of the new method of updating. The driving agent is able to successfully update on the fly upon receiving new traffic data that corresponds with its current route plan.

There are many other applications for agent programming where dynamically updating is very useful. There may be some agents that make decisions based on partial data because complete data is not available. In this situation it is desirable for the agent to update its plan when a more complete set of data is available. For example, if a boat is approaching very quickly and has a flag raised with a skull and crossbones on it the agent may want to prepare for an attack. The agent shouldn't attack yet because we don't know yet if the boat is coming to attack and hasn't expressed any hostility. If the agent gets information that the other boat has weapons onboard then it may be necessary to take other steps as part of the preparation plan such as getting ready to call for help.

However, when the agent gets information that the other boat is a cargo ship for a toy company the agent can change the plan to welcome the other boat.

There are many tasks that were outside the scope of this thesis that are planned for future work in this BDI framework. There are many items that exist within the framework that are not displayed to a human user that may be useful. Also, various other concepts discussed in this thesis may be useful for the implementation of the BDI framework. Lastly, there are many other concepts that are important for updating and the BDI framework that are also outside scope of this thesis.

The inner workings of the BDI framework are very complex but some items may be useful for a human user. In the intention view it is useful for a human to be able to know which task of an intention is currently running. This could be shown by a simple marker next to the particular task in the intention specification window. The human user may also benefit from information about all currently running tasks from all intentions. This could be shown in a side view list of some sort. One small area in the intention view where it should be different from the plan view is with bindings. Once a plan becomes an active intention there are specific bindings associated with it. It would be useful if the intention view showed the values of these bindings instead of the binding variable that is shown in the plan view.

There are some concepts discussed in this thesis for the BDI framework that are not included in the implementation for the project. One of the major capabilities of the R-CAST agent framework is the communication module used for multi-agent systems. The BDI framework would benefit if agents could communicate with each other as part of the intention module. In some cases it may be necessary to use the previously discussed forms of plan updated such as stop and restart updating or finish and restart updating. However, as a side effect to adding these methods of updating, it will also be necessary to modify the configuration for the user to specify which method of update should be used for each individual plan.

Some future work will involve adjusting the BDI module so that the framework will run more smoothly in different circumstances. Currently, if a specific task step takes a very long time to complete there is no way to allow other tasks to run their steps. It may be useful to temporarily allow multiple tasks to concurrently run a step when a single step takes a long time. An agent may need to check if two running intentions contradict each other. In the current BDI framework an agent is able to run the plans “ride the train” and “take milk out of the refrigerator” at the same time even though it is physically impossible.

There are also other topics for further research in the area of updating in the BDI framework. The two topics in mind are to allow rollback to a specific task and to allow multiple updates to be associated with an intention. The idea to rollback to

a specific task in the task list is non-trivial. This would require aliasing of tasks in a plan to select a specific task. Aliasing of tasks is needed because the intention may have multiple instances of a task and the agent should know which instance to roll back to. The other topic involves the ability to have multiple updates associated with a single intention. Currently, the user may only specify one set of update conditions for an intention. The problem is that this doesn't allow for different circumstances to cause an update so the user must choose the most important one or somehow combine the conditions into one.

APPENDIX

A.1 Plan List for Driving Agent

```
<Plans>
<Plan name="GetDirections">
   <cues>
      <cue name="drive_from_to">
         <cuevariable name="?from" value="?from"></cuevariable>
         <cuevariable name="?to" value="?to"></cuevariable>
      </cue>
   </cues>
   <precondition>
      <predicate name="not_going_anywhere">
         <predicatevariable name="?true_false" value="true"></predicatevariable>
      </predicate>
   </precondition>
   <tasks>
      <task type="print" value="GetDirections: Starting"></task>
      <task type="retract" value="not_going_anywhere true"></task>
      <task type="user" class="edu.psu.bdi.test.GetDirections"
            value="?from ?to">
         </task>
      <task type="print" value="GetDirections: Done"></task>
   </tasks>
   <failure>
      <predicate name="no_possible_route">
         <predicatevariable name="?from" value="?from"></predicatevariable>
         <predicatevariable name="?to" value="?to"></predicatevariable>
      </predicate>
   </failure>
</Plan>

<Plan name="DriveAlongSection">
   <cues>
      <cue name="at_road">
         <cuevariable name="?road_name" value="?road_name"></cuevariable>
      </cue>
      <cue name="route_section">
         <cuevariable name="?road_name" value="?road_name"></cuevariable>
         <cuevariable name="?end_road" value="?end_road"></cuevariable>
      </cue>
   </cues>
   <tasks>
```

```
      <task type="print" value="DriveAlongSection"></task>
      <task type="blank" value="--Simulate Driving Time--"></task>
      <task type="blank" value="--Simulate Driving Time--"></task>
      <task type="blank" value="--Simulate Driving Time--"></task>
      <task type="blank" value="--Simulate Driving Time--"></task>
      <task type="retract" value="route_section ?road_name ?end_road"></task>
      <task type="retract" value="at_road ?road_name"></task>
      <task type="assert" value="at_road ?end_road"></task>
   </tasks>
</Plan>

<Plan name="GoToPhiladelphia">
   <precondition>
      <predicate name="at_road">
         <predicatevariable name="?at" value="?at"></predicatevariable>
      </predicate>
   </precondition>
   <tasks>
      <task type="print" value="GoToPhiladelphia: Start"></task>
      <task type="assert" value="drive_from_to ?at philadelphia"></task>
      <task type="untiltrue">
         <predicate name="at_road">
            <predicatevariable name="?to" value="philadelphia">
               </predicatevariable>
         </predicate>
      </task>
      <task type="retract" value="drive_from_to ?at philadelphia"></task>
      <task type="assert" value="not_going_anywhere true"></task>
      <task type="print" value="GoToPhiladelphia: Done"></task>
   </tasks>
   <success>
      <predicate name="at_road">
         <predicatevariable name="?to" value="philadelphia"></predicatevariable>
      </predicate>
   </success>
   <update>
      <updatecondition>
         <andoperator>
            <predicate name="traffic_at_section">
               <predicatevariable name="?start_section" value="?start_section">
                  </predicatevariable>
               <predicatevariable name="?end_section" value="?end_section">
                  </predicatevariable>
               </predicate>
            <andoperator>
               <predicate name="at_road">
```

```
                <predicatevariable name="?current_road" value="?current_road">
                  </predicatevariable>
              </predicate>
              <predicate name="route_section">
                <predicatevariable name="?start_section" value="?start_section">
                  </predicatevariable>
                <predicatevariable name="?end_section" value="?end_section">
                  </predicatevariable>
              </predicate>
            </andoperator>
          </andoperator>
        </updatecondition>
        <updatetasks>
          <task type="retract" value="route_section ?x ?y"></task>
          <task type="print" value="GoToPhiladelphia Start Updating"></task>
          <task type="retract" value="traffic_at_section ?start_section
              ?end_section">
            </task>
          <task type="user" class="edu.psu.bdi.test.GetAlternateDirections"
              value="?current_road ?to">
            </task>
          <task type="print" value="GoToPhiladelphia Done Updating"></task>
        </updatetasks>
    </update>
</Plan>

<Plan name="AddTraffic">
    <tasks>
        <task type="assert" value="traffic_at_section road1f philadelphia"></task>
        <task type="print" value="AddTraffic: Done"></task>
    </tasks>
</Plan>
</Plans>
```

A.2 Driving Agent Knowledge Base

```
#     The Fact Types

      (FactType drive_from_to (?from ?to))
      (FactType route_section (?road_name ?end_road))
      (FactType at_road (?road_name))
      (FactType no_possible_route (?from ?to))
      (FactType not_going_anywhere (?true_false))
      (FactType traffic_at_section (?section_start ?section_end))
```

```
#     The Facts

      (Fact not_going_anywhere (true))
      (Fact at_road (state_college)
```

BIBLIOGRAPHY

[1] F. L. Bellifemine, G. Caire and D. Greenwood, *Developing multi-agent systems with JADE*, John Wiley, Chichester, England ; Hoboken, NJ, 2007.

[2] M. Bratman, *Intention, plans, and practical reason*, Harvard University Press, Cambridge, Mass., 1987.

[3] M. Dastani, M. B. v. Riemsdijk and J.-J. C. Meyer, *Goal types in agent programming*, *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM, Hakodate, Japan, 2006.

[4] B. Dunin-K, plicz and R. Verbrugge, *Evolution of collective commitment during reconfiguration*, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, ACM, Bologna, Italy, 2002.

[5] K. Erol, J. Hendler and D. S. Nau, *HTN planning: complexity and expressivity*, *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, American Association for Artificial Intelligence, Seattle, Washington, United States, 1994.

[6] X. Fan, S. Sun, M. McNeese and J. Yen, *Extending Recognition-Primed Decision Model For Human-Agent Collaboration*, *Fourth International Joint Conference on Autonomous Agents and Multi Agent Systems*, The Netherlands, 2005, pp. 945-952.

[7] X. Fan, J. Yen, M. Miller, T. Ioerger and R. Volz, *MALLET—A Multi-agent Logic Language for Encoding Teamwork*, IEEE Transaction on Knowledge and Data Engineering, Vol. 18 (2006).

[8] T. Finin, R. Fritzson, D. McKay and R. McEntire, *KQML - A Language and Protocol for Knowledge and Information Exchange*, *13th Int. Distributed Artificial Intelligence Workshop*, Seattle, WA, 1994.

[9] T. Finin, R. Fritzson, D. McKay and R. McEntire, *KQML as an agent communication language*, *Proceedings of the third international conference on Information and knowledge management*, ACM, Gaithersburg, Maryland, United States, 1994.

[10] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro and C. Beck, *Specifation of the KQML Agent Communication Language*, External Interfaces Working Group ARPA Knowledge Sharing Initiative (1993).

[11] W. Gerhard, ed., *Multiagent systems: a modern approach to distributed artificial intelligence*, MIT Press, 1999.

[12] T. R. Gruber, *Toward principles for the design of ontologies used for knowledge sharing*, Int. J. Hum.-Comput. Stud., 43 (1995), pp. 907-928.

[13] H. Jiang, J. M. Vidal and M. N. Huhns, *EBDI: an architecture for emotional agents*, *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ACM, Honolulu, Hawaii, 2007.

[14] D. Kinny, *ViP: a visual programming language for plan execution systems*, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, ACM, Bologna, Italy, 2002.

[15] O. Lassila and R. R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, http://www.w3.org/TR/PR-rdf-syntax/, 1999.
[16] V. Mařík, P. Vrba, K. H. Hall and F. P. Maturana, *Rockwell automation agents for manufacturing*, *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, ACM, The Netherlands, 2005.
[17] P. Massonet, Y. Deville, C, d. N and ve, *From AOSE methodology to agent implementation*, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, ACM, Bologna, Italy, 2002.
[18] D. Morley and K. Myers, *The SPARK Agent Framework*, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, IEEE Computer Society, New York, New York, 2004.
[19] L. Penserini, L. Liu, J. Mylopoulos and L. Spalazzi, *Modeling and Evaluating Cooperation Strategies in P2P Agent Systems*, *Proceedings of the Agents and Peer-to-Peer Computing, AAMAS02*, Bologna, Italy, 2002, pp. 87-99.
[20] O. F. Rana, M. Winikoff, L. Padgham and J. Harland, *Applying conflict management strategies in BDI agents for resource management in computational grids*, *Proceedings of the twenty-fifth Australasian conference on Computer science - Volume 4*, Australian Computer Society, Inc., Melbourne, Victoria, Australia, 2002.
[21] A. S. Rao and M. P. Georgeff, *Modeling Rational Agents within a {BDI}-Architecture*, in J. Allen, R. Fikes and E. Sandewall, eds., *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning ({KR}'91)*, Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, Cambridge, MA, 1991, pp. 473-484.
[22] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, Prentice-Hall, Inc., 2003.
[23] S. Sardina and L. Padgham, *Goals in the context of BDI plan failure and planning*, *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ACM, Honolulu, Hawaii, 2007.
[24] S. Sardina, L. d. Silva and L. Padgham, *Hierarchical planning in BDI agent programming languages: a formal approach*, *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM, Hakodate, Japan, 2006.
[25] J. M. Serrano and S. Ossowski, *The design of agent communication languages: an organizational approach*, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, ACM, Bologna, Italy, 2002.
[26] s. Sevay and C. Tsatsoulis, *Multiagent reactive plan application learning in dynamic environments*, *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, ACM, Bologna, Italy, 2002.
[27] J. Thangarajah, J. Harland, D. Morley and N. Yorke-Smith, *Aborting tasks in BDI agents*, *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, ACM, Honolulu, Hawaii, 2007.
[28] N. Vlassis, *A concise introduction to multiagent systems and distributed artificial intelligence*, Morgan & Claypool Publishers, [San Rafael, Calif.], 2007.

[29] M. Winikoff, L. Padgham and J. Harland, *Simplifying the Development of Intelligent Agents*, *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, Springer-Verlag, 2001.
[30] M. Winikoff, L. Padgham, J. Harland and J. Thangarajah, *Declarative and procedural goals in intelligent agent systems*, *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002, pp. 470-481.
[31] M. Woolridge and M. J. Wooldridge, *Introduction to Multiagent Systems*, John Wiley \& Sons, Inc., 2001.
[32] J. Yen, T. R. Ioerger, M. S. Miller, S. Sun, K. Kamali, S. Zhu, X. Fan and R. A. Volz, *The CAST Manual*, (2004).
[33] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu and R. A. Volz, *CAST: Collaborative Agents for Simulating Teamwork*, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, Seattle, WA, 2001, pp. 1135-1142.