A COMPILER DESIGN FOR THE AGENT-BASED PROGRAMMING LANGUAGE

Wei Zhao
Department of Computer Science
University of North Dakota
Grand Forks, ND 58203, USA

Chang-Hyun Jo
Department of Computer Science
California State University Fullerton
Fullerton, CA 92834-6870, USA
jo@ecs.fullerton.edu

Contact Point

Chang-Hyun Jo
Associate Professor
Department of Computer Science
California State University Fullerton
Fullerton, CA 92834-6870, USA
jo@ecs.fullerton.edu
(714) 278-7255

A COMPILER DESIGN FOR THE AGENT-BASED PROGRAMMING LANGUAGE

Wei Zhao
Department of Computer Science
University of North Dakota
Grand Forks, ND 58203, USA

Chang-Hyun Jo
Department of Computer Science
California State University Fullerton
Fullerton, CA 92834-6870, USA
jo@ecs.fullerton.edu

ABSTRACT
In this paper, we introduce a prototype compiler for the Agent-based Programming Language (APL) using the Belief-Desire-Intention (BDI) model. The compiler we developed can translate an agent-based program written in APL into the Java byte code, which is executable on the Java Virtual Machine (JVM).

This paper proposes a simple syntax for Agent-based Programming Language (APL). APL is designed based on the BDI model, which is a natural way to describe the behaviors of an intelligent agent. By using the associated database and reflection concepts, the APL can use different intentions intelligently and dynamically. This work shows how nicely and naturally the APL programming language can represent intelligent software agents and how we can translate the APL program into the Java byte code. In this work, we use the Belief-Plan mapping table and Java reflection feature to implement dynamic linking which makes it possible for APL to link to different code fragments at runtime without recompiling the APL source code. This feature of APL makes the language unique and it may change the way of programming.

Keywords
Agents, Agent-based Computing, Agent-based Programming Language, Agent-based Software Engineering

1. INTRODUCTION
Agent-based computing has emerged as a new computing paradigm. There have been many research results in this area [Busetta et al. 97, 98] [DeLoach 99] [Franklin et al 96] [Huber 99] [Petrie 96]. The BDI model is one of the most powerful techniques to describe autonomous intelligent agents. However, there are still no proper programming languages and software engineering methodologies that can naturally support the agent-based computing. In this paper we present the Agent-based Programming Language (APL) and a prototype compiler for the APL we developed.

The idea of the APL language was first introduced in the work of Jo and Arnold [2002]. The previous work includes the noble concept brought in the APL language design and a prototype interpreter (preprocessing) for it. This paper reports its continuing work as promised, and shows a real compiler implementation with a realistic application example. The compiler implementation of APL allows us to design and implement realistic agent-based applications. It also supports both a realization of the BDI agent concept and a certain degree of learning and adaptation, which might be a very valuable scheme for the agent-based programming. This paper describes the basic grammar of the APL. A sample source program has been developed based on the Agent-based Stock Trader (AST) system [Feng 01]. The APL sample program defines a stock recommendation agent that has its own Belief (environment), Desire (goals to achieve) and Intention (the actions it will perform to achieve the goals). By analyzing the environment (Belief) the agent can give clients some useful recommendation and help clients to make the right decisions. The agent has its own knowledge base (Belief). It can continually learn new knowledge and keep updating its knowledge base at runtime. The agent has the ability to choose different plans and give different recommendations at runtime if the belief stored in the knowledge base is changed. The main purpose of this work is to show how simply and naturally the APL language can describe the agent-based programming. Since we cannot access the real time stock market, we will use an example database to simulate it.

APL is a new programming language designed to describe the behaviors of the intelligent software agent. It has a different programming paradigm from the previous ones. In order to make our APL programming language work, we need a compiler and a runtime system. Since we do not have a runtime system for the APL right now, the Java Virtual Machine (JVM) is chosen to be the supporting runtime system for the APL at this time. The compiler translates the APL source program into the Java byte code, which can be run on the JVM with a database to manipulate the BDI knowledge base. We use special tools such as LEX and YACC to develop the compiler.

In the next section, we introduce some background about the agent-based programming concept and the BDI model, explain some basic concepts in our APL programming language, and compare our work with the related works.

The basic grammar of our new APL language is defined in Section 3. In Section 4, we describe our implementation scheme of the APL compiler. We also show precisely how to translate a sample APL application into the Java byte code step by step. The source APL program and the compiler are just a simple prototype with which we can describe how nicely and naturally APL can implement the intelligent software agents. We also discuss a way to make it possible for the agent to dynamically link to different plans at runtime without recompiling the APL source program. This ability makes the APL system reflexive and intelligent. The APL programming language may change the way of programming.

We conclude our work and specify the future work in Section 5. In Section 6 we list the reference. The source code will be available in public upon the publication of this work.

2. BACKGROUND OF BDI-AGENTS

A software agent is a piece of autonomous software, which has some level of intelligence. An agent is a high-level system component. It has its own goals (Desire) to be accomplished. An agent is a concurrent, autonomous, intelligent and self-contained object [Jo 01]. An autonomous agent is an object that senses the environment, and acts on it based on its own agenda [Petrie 96]. In order to achieve its goal, the agent analyzes the current environments and chooses proper plans. Agents should intelligently respond to the events that trigger them. They can actively respond to any changes in the environment on which they are in [Jo 01]. The agents may have their autonomy and are not controlled directly by others. Several agents can communicate among each other and their environment. They can cooperate with other agents to achieve the same goals. Agents can keep learning from and adapt to their environment. They can change their behaviors based on their previous experience and inference from the existing knowledge. The learning and adaptation can make the agent achieve the goal in a better way. This makes the agent more intelligent.

Agent-based computing is a totally new programming paradigm. There is no programming language or modeling technique that can support it naturally yet. A lot of research has been done in this area. The BDI model [Bratman 87] is one of the most powerful techniques. BDI is the abbreviation of Belief, Desire, and Intention. The Belief of an agent represents the knowledge about itself and the world (outside environment). The Desire of an agent describes the goals that the agent can achieve. The Intention is a set of plans used to achieve the predefined goal based on a specific situation. It defines the action that the agent performs. Since BDI can describe the behavior of a software agent naturally, we developed our APL programming language based on this concept.

There are many possible ways to support agent-based programming on the BDI model. Multi-paradigm mixed with object-oriented programming, BDI concepts, and logic programming can be the possible ways to support it [Jo 02]. We can also use a database to specify the knowledge domain BDI. Using an embedded language such as SQL in an existing programming language is another solution to support the BDI-agent model. The multi-agent system needs good communication and coordination between agents. Distributed computing language models can be used to describe this feature. JavaSpace and JINI are examples of such models.

In a previous work on the AST system (Agent-based Stock Trading system) [Feng 2001], the Belief, Desire and Intention are defined as Java classes in Java programming language. It has shown us that the BDI model can be implemented nicely using Java with a database support. But Java class cannot support BDI naturally and nicely. We still need our own agent-based programming language. We also need a compiler or interpreter to make this language executable. In the next chapters we show how to build the APL language and how to translate the APL into the executable code. Since the Java Virtual Machine has a lot of interesting features, we translate our APL code into the Java byte code on the Java Virtual Machine at this time. The runtime system solely for APL can be developed later on.

There have been several research and experimental works based on the BDI model extending the Java programming language, and some of them are going for commercial. We describe them briefly here. They are BDIM, JACK and JAM.

BDIM (Belief, Desire, Intention, Message) Agent Toolkit is implemented as a Java package to provide a prototype of runtime architecture [Busetta et al. 97, 98]. BDIM is an architecture-independent definition of the basic elements of a BDI agent. Its purpose is to define an interface between the application and the runtime system. An agent is developed by deriving classes for each of the agent's Plans and Beliefs from the relevant base classes in BDIM [Busetta et al. 97, 98].

JACK Intelligent Agent is an environment for building, running and integrating commercial-grade multi-agent systems using a component-based approach. It offers Class, Interface, Method, Syntactic and Semantic extensions of Java program language that is implemented as a Java plug-in that provides an agent-oriented development environment [JACK 99]. The most important feature of JACK is that JACK uses the database class as its data storage device to describe the agent's belief. Then the database class is fully integrated with other JACK classes. With JACK, we can develop nice Java programs, which support the BDI concept.

JAM (Java Agent Model) is an implementation of the BDI agent model in Java. Implemented as an interpreted programming language, JAM enables users to build agents that can communicate and migrate. It integrates fairly easily with the legacy Java code via the Java reflection package. JAM is composed of five primary components: the world model, the plan library, the interpreter, the intention structure, and the observer [Huber 99]. JAM uses text-based files to specify an agent's BDI. Users need to write appropriate primitive functions in Java, and specify the agents' BDI in JAM file.

All of these works are based on Java. But they are not good and natural enough to describe the agent behavior. JACK can manipulate the belief as a database as well. But it cannot support dynamic manipulation of desire and intention. JAM and BDIM provide a way to manipulate BDIs somewhat. But they are not powerful enough to describe the BDI. Since they are Java extensions, they can only describe an agent program in Java classes. They are not really agent-based programming languages. They have their own syntax. Users have to spend a lot of time to understand them before they can use them to develop the software agents. Our APL is a new language that is developed specially for agent-based computing. It describes the program on the agent level. APL can dynamically manipulate the BDIs at runtime. The programmer can change the Belief, the Desire and the Intention at runtime without recompiling the program. The agent can execute a better plan if there is one at runtime. The dynamic linking technique of APL makes the agent more intelligent.

3. THE STRUCTURE OF APL

3.1 The Template of APL

The APL has been designed based on the BDI concept. The APL program is composed of four parts: the Agent, the Belief, the

Desire and the Intention. Each agent may have separate BDIs or may share some of them [Jo 01]. In order to make our APL easier to read and learn, we develop the APL language in a Java-like syntax. We define a basic template for each of the APL components in this section.

Figure 1 shows the template of the Belief definition in APL. Every variable in the APL is of the belief type, and every method in the APL is called a plan. One of the most important features of APL is that it uses a database to store the belief (the knowledge). The Beliefs is defined as the "class" level, which means that the Belief has its own beliefs and the plans that manipulate these beliefs. In most of the cases, the Belief defines a database that is used to store knowledge. In order to access and modify the belief (knowledge) stored in the database, two plans named getBelief () and setBelief () should be defined in the Belief. The only way to access and modify the belief stored in the database is to invoke these plans. In order to maintain the integrity of the database, this should be the only way for an agent to access and modify the database.

```
Belief Bi {
        belief<database> db;      //belief definition
        …
        public void setBelief (…) {
                                  // plan used to set the belief
        …
        }
        public int getBelief (…) {     // plan used to get belief
        …
        }
 }
```

Figure 1. The Template of Belief Definition

Figure 2 shows the template of the Desire definition in APL. The Desire defines the goals that the agent tries to achieve. Since the agent is autonomous, we allocate a new thread to it when an instance of the agent is created. The Desire is defined as the "class" level, too. A special plan named run () has to be defined in the Desire definition. This plan can be automatically invoked when an agent wants to achieve the goal defined by the Desire. When an agent wants to achieve the goal, it first checks if this goal is achievable (if the desire is defined). Then the agent tries to get beliefs from the knowledge base (implemented as a database in APL). After the agent analyzes the belief that represents the environment, it checks the knowledge base from which the agent can choose a proper plan to achieve the goal based on the analyzed result of belief. This is called the reasoning procedure of agents in the APL system.

```
Desire Di {
        belief<database> db;     // belief definition
        …
        public void run (…) {
        …
        }
 }
```

Figure 2. The Template of Desire Definition

Figure 3 shows the template of Intention definition in APL. The Intention defines the actions that an agent performs in order to achieve a goal in a specific condition. Intention is defined as the Java "class" level, and is composed of several plans. In order to achieve a goal, an agent may use one or more plans defined in the Intention part. The plans defined in the intention can be written in the similar syntax to the Java method level at this time.

```
Intention Ii {
        belief<int> b1;                  //local belief definition
        …
        public void plan1 (…) {          // definition of plan 1
        …
        }
        …
        public void planN (…) {          // definition of plan N
        …
        }
 }
```

Figure 3. The Template of Intention Definition

```
Agent Ai {
        belief<int> b1;
        desire d2;
        intention i3;
        …
        public void initiator (…) {
        …
        }
}
```

Figure 4. The Template of Agent Definition

The last component we need to define is the Agent. Figure 4 shows the template for the agent definition. The Agent is also defined as the Java "class" level. This does not mean agent and class are the similar. An agent is totally different from class because our agents support intelligence (much more than classes) by providing the BDI concept, learning and adaptation. An agent has its own local belief and its own plans. The agent will declare the Belief (environment it bases on), the Desire (goals it can achieve), and the Intention (the actions it can perform to achieve the goals) that it will use. An agent is autonomous. It works in a separated thread. When an agent is asked to achieve some goals, the agent checks if it has the ability to achieve this goal. This is done by checking if the desire for this goal is defined and declared by the agent. If the goal is achievable, the agent will create a instance of the desire in a separated thread. Then the plans that the agent uses to achieve the goal is executed on this thread.

3.2 Calling Mechanism of Agents

Each agent has one or more desires (goals) to achieve. Figure 5 shows the calling mechanism between agent Ac and agent As. The Ac stands for Agent Client, and the As stands for Agent Server. Each agent is defined by its own belief, desire and intention [Jo 01]. When the agent Ac wants to ask Agent As to achieve a certain goal, Ac sends a request in the format of "As.goal (…)" to agent As. Further information can be sent to agent As as parameters in the request. After receiving the request, the agent As checks if the goal is achievable (if the desire is defined and declared in the agent). If the desire is defined, the agent As checks the current environment from its belief definition. By analyzing the environment, the server agent refers to the knowledge base from which it finds the most appropriate plan that is defined in its

intention definition and executes the plan. While running the APL system, the environment (belief), the goal (desire) and the plans (intention) can be modified and added. The new environment, the new goals and the new plans can be updated in the knowledge. The next execution of the system will act based on this new knowledge. This is a "learning" process. It makes APL as an intelligent language.

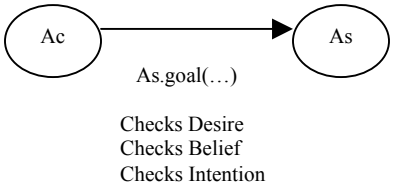

Figure 5. Calling Mechanism of Agents

This learning process of APL program execution can be described more formally like the following [Jo 02]:

Ac -> As.Gi : As(Bi, Di, Ii) => As(Bj, Dj, Ij)

The client agent Ac asks server agent As to achieve goal Gi. The agent As finds out that Gi belongs to the set of desire Di. So the agent As achieves the goal Gi based on the current Bi, Di and Ii. During the execution, Bi, Di and Ii have been modified to Bj, Dj and Ij. These changes are updated in the knowledge and replace the old records. Next time any client agent may invoke As with the information (Bj, Dj, Ij).

3.3 Example of APL Program

Following the templates that are described in previous section, we develop the *Stock Recommendation Agent (SRA)* system in the APL language based on the AST system [Feng 01]. This program is composed of 5 parts: a client agent mainAgent who sends a request of goal to a server agent stockAgent, a Belief stockBelief, a Desire recomDesire and an Intention stockIntention for the stockAgent. We describe the program in detail in the next sections as an example to show how to translate an APL code to Java byte codes.

4. THE COMPILER FOR APL

4.1 Implementation of APL

Figure 6 shows how to execute the APL program. Agent-based programming is a new programming paradigm that has arisen from research in distributed artificial intelligence.

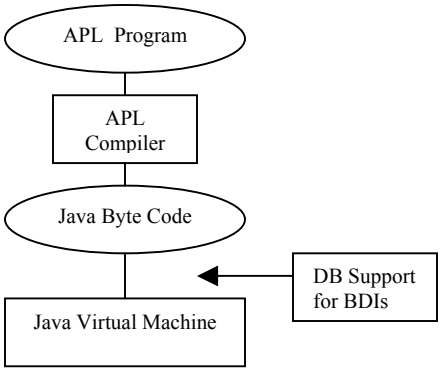

Figure 6. Implementation of The APL programming Language

There is no natural programming language that could support it well before. When we define the new language APL, we also need to develop a compiler or an interpreter. The Java Virtual Machine is a powerful tool to develop cross-platform programs. We use the JVM as our runtime system at this time. We translate our APL source programs into the Java byte codes that can be executed on the JVM.

4.2 How to Implement BDI-Agent In APL

In this section we show how the APL system handles the BDI agents. A relational database is used to represent the agent's belief, including the agent's knowledge base and environment states.

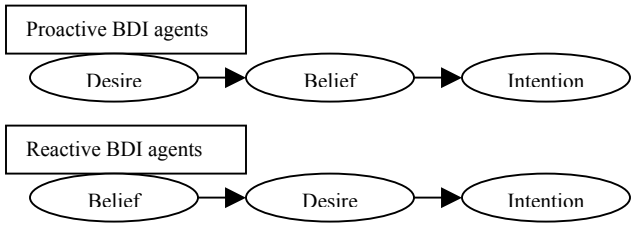

Figure 7. (7a & 7b) Proactive/Reactive BDI Agents

We have two scenarios on the behaviors of BDI agent: Proactive BDI agents and Reactive BDI agent. A proactive BDI agent acts based on a goal. It scans the beliefs, searches for the intention based on the belief and the goal, and achieves the goal by executing the intention [Figure 7a]. A reactive BDI agent reacts based on the beliefs, looks for a goal and its intention [Figure 7b].

To explain how we implemented the APL system, we will use an realistic application example – the *Stock Recommendation Agent* (SRA) system. The database serving for the SRA system is similar to the database of the AST system [Feng 01]. Two tables are necessary to support the SRA system. One stores the knowledge of the agent. The other stores the information about plans that the agent executes. Table 1 shows the database contents for the SRA system.

Table 1. Database Contents

Database for the SRA system	
Table name	**Content**
Knowledge	This table contains agent's knowledge, including desire and belief value.
BPmap	This table indicates which plan will be chosen by reasoning the belief from the knowledge table.

The agent defined in our SRA system gives the client agent certain recommendations on the stock trading. This software agent works in the same way as a real stock agent does. In the real world, when a customer wants to get some advise from a stock agent, he asks the stock agent for some professional advise (the goal, the desire) on the stocks he is interested in. When the stock agent gets this request and the stock name, he checks the prices of those stocks, the world economy, the buy and sell market, the financial markets and lots of other factors. After evaluating all this information carefully, the stock agent suggests that the customer would better buy, sell or hold the stocks. It is exactly how our software agent

works. The SRA can simulate this intelligent work that only humans can perform.

The server agent (stockAgent) defines a recommendation desire, so that the stockAgent can give the client agent recommendation on stocks by achieving this goal. The stockAgent is composed of 4 parts: the Belief stockBelief that represents the environment, the Desire recomDesire that defines the goal of the agent stockAgent, the Intention that defines several plans that the agent can perform and the Agent that makes use of all the BDI components. There is also a knowledge base (represented by a database) used by the stockAgent that stores important factors and affects the recommendation on the stocks (as shown in Table 2). An instance of the value of knowledge is shown in Table 3. (It will be explained later.)

Table 2. SRA Database Knowledge Table Definition

BDI Knowledge for the SRA system		
Field Name	**Data Type**	**Description**
Symbol	Text	Stock name
A	Number	World economy
B	Number	US economy
C	Number	Financial markets and institutions
D	Number	Buy or sell market
E	Number	International currency
F	Number	Others
Belief	Number	The result of analysis of A~F

Table 3. Sample Values In Knowledge Table

Values in Knowledge							
Symbol	**A**	**B**	**C**	**D**	**E**	**F**	**Belief**
YHOO	5	2	-1	8	0	10	24
…	…	…	…	…	…	…	…

The integrity of the knowledge base is a big issue. The server agent can only give out valuable decisions based on the right belief (environment). In order to maintain the correctness of the database, only agents who have the access privilege can access and modify the knowledge base. In our SRA, the client agent cannot access the belief database that belongs to the server agent. Only the server agent can access and modify the contents in database. Access control can be done by using a password. Some locking techniques should be applied if the knowledge is accessed by multiple agents.

There is another knowledge in the SRA system called Bpmap (as shown in Table 4). BPmap contains the plan information that the agent needs to perform in order to achieve the goal. When the user wants to ask for a suggestion on a stock such as YHOO (the stock name for Yahoo), he gives the stock name to the client agent named mainAgent. Then the mainAgent invokes the server agent in the format "stockAgent.recomDesire(YHOO)". The server agent stockAgent is activated by this request. It calls the desire recomDesire, accesses the knowledge base and gets the belief (which are important factors). After calculating this belief in a certain way (Table 3), the server agent refers to the mapping table (Bpmap) (Table 4 and Table 5) and checks which plan is suitable at this time. Then the server agent returns a recommendation to the client agent based on the execution of the plans. We assume that there is another agent that is defined to update the knowledge base when those factors are changed. The administrator or the agent that defined to maintain the mapping table updates the BPmap if there is a better plan developed. These changes do not affect the stockAgent itself. The server agent can always be kept alive without stopping and recompiling the source code.

The Knowledge table contains 8 columns (Table 2 and Table 3). The stock name "symbol" is the goal that we want to get recommendations on. For example, it can be YHOO (Yahoo), IBM (IBM), MS (Microsoft) and so on [Yahoo 00]. The knowledge A through F represents the beliefs related to the stock. The values of these beliefs are maintained and updated by the system administrator or other agents that are defined to access and modify the contents of this knowledge. In our SRA system, we let the system administrator do this job. But in the real world, we can develop an agent to update the knowledge. The beliefs the agent gets from the knowledge can be calculated in different ways depending on the algorithm used by those agents. The last field holds the value after the agent analyzes the belief A to F. The server agent will choose the plans based on this value. This value can be calculated either by the server agent or by some other agent who is designed to do it. We will let the server agent do the job this time. Table 3 shows an example of a record in the Knowledge.

The purpose to have this example application is just showing how APL handles the BDI. The algorithm we chose to calculate the belief values is just trivial in this example. So now we just add all these values from A to F. However, in the real application, appropriate mathematical functions and stochastic methods for the stock market forecasting can be used.

The BPmap table is a very important table in the SRA system. The server agent chooses different plan based on the value that is stored in it. Table 4 shows the BPmap table Definition and Table 5 shows an example record in the BPmap table.

Table 4. BPmap Table Definitions

BPmap table for the SRA system		
Field Name	**Data Type**	**Description**
LowLimit	Number	Lower limit of belief
UpLimit	Number	Upper limit of belief
Plan	Text	Corresponding plan
PlanChoice	Text	Choice of a plan
Intention	Text	Plan name in intention

Table 5. Sample Values In BPmap Table

Values in BPmap				
LowLimit	**UpLimit**	**Plan**	**PlanChoice**	**Intention**
-3	3	Hold	4	planeC

In the BPmap table, the condition fields, LowLimit and UpLimit, represent the lower limit and upper limit of the value of belief, from which the agent can select corresponding plans. The Intention field stores the plan names that are predefined in the Intention definition, which can be called during execution of the APL program. The server agent compares the belief value with the conditions and gets the plan name that is stored in the Intention field. Then the agent invokes the plan and performs the action that is defined in the intention. To make it simple, the plans defined in the stockIntention of our SRA system just print out the plan names and simple recommendations. But in the real world it can perform more complicated jobs.

4.3 Translate the APL Code into the Java Byte Code

We have defined the template of each component of APL above. We also described how the APL programming language works with an example of the SRA system. The previous example shows only rough idea without showing the code, but with showing an example of the knowledge base for the BDI agents. Here we will show the real code for the SRA system written in the APL language, and we will show how to translate it to the Java byte codes. The APL compiler translates the APL source program into the ASCII format Java byte code. Then we use Jasmin [Meyer and Downing 97] to assemble the Java byte code into the binary executable class file which is directly running on the Java Virtual Machine.

The Belief part of APL is translated into a Java class in the Java byte code, and each plan defined in the Belief are translated into a Java method. The APL system involves a lot of database manipulation. But this is hidden from the programmers. A programmer only needs to concentrate on how to implement the BDIs and leaves the database manipulations to the runtime system. This makes the APL language easy to read, and it describes the BDI more naturally.

The following is the Belief definition of the SRA system in APL. It is a fairly simple program that only contains one belief definition, one constructor and two member plans. In APL, all variables are the belief type. So we have a new definition syntax: "belief<data type> variable name". The "data type" can be any primary data type or "database". When it is a primary type like integer or string, we translate it into the corresponding Java primary data type. But if it is "database", then we translate it into the Connection object in Java. Most of the APL statements in the APL plan level are just like Java; we can translate them into the Java byte code easily. But there are some new statements that are especially invented for APL. Most of them are related to the database manipulation. We give special treatment to these statements.

```
Belief stockBelief {
        belief<database> db;
  public stockBelief(){
        db = "SRAdb";
        }
  public void setBelief (belief<database> dbName,
                belief<String> tableName,
                belief<String> columName,
                belief<String> symbol,
                belief<String> goal, belief<int> value)
        {
            set columName = value where symbol = goal;
        }
  public int getBelief (belief<database> dbName1,
                belief<String> tableName1,
                belief<String> columName1,
                belief<String> symbol1,
                belief<String> goal1)
        {
            belief<int> i;
            i = get columName1 from tableName1
                where symbol1=goal1;
            return i;
        }
  }
```

In the Belief definition, we have three special statements. They are the database connection statement, the database access statement and the database update statement. We describe how to translate them into the Java byte code in detail here.

The first special statement is the APL knowledge base connection statement that is defined in the constructor of Belief definition. It is "db = "SRAdb";". At first glance, it looks like an assignment statement. But what it actually does is to connect to the database and store the reference of the database connection object in the local variable db. The compiler first checks if the type of left side of this construct is the Connection type. Then it checks if the right side is a String (name of the database). If either of them is not true, then the compiler will treat it as a regular assignment statement. Otherwise the compiler will give it a special treatment. The following code shows how the above database connection statement is translated to the Java byte code.

```
:
.catch java/lang/ClassNotFoundException
        from Label1 to Label2 using handler1
  Label1:
            ldc "sun.jdbc.odbc.JdbcOdbcDriver"
            invokestatic
java/lang/Class/forName(Ljava/lang/String;)Ljava/lang/Class;
            pop
  Label2:
            goto Label3
  handler1:
            pop

            getstatic java/lang/System/err Ljava/io/PrintStream;
            ldc "no such a driver exists!"
            invokevirtual
                java/io/PrintStream/println(Ljava/lang/String;)V
            iconst_1
            invokestatic java/lang/System/exit(I)V
  Label3:
            .catch java/sql/SQLException from Label4 to Label5
using handler2
   Label4:
            aload_0
            ldc "jdbc:odbc:SRAdb"
            invokestatic
java/sql/DriverManager/getConnection(Ljava/lang/String;)
 Ljava/sql/Connection;
```

```
            putfield stockBelief/db Ljava/sql/Connection;
 Label5:
            goto Label6
 handler2:
            pop
            getstatic java/lang/System/err Ljava/io/PrintStream;
            ldc "cannot connect to database!"
            invokevirtual
java/io/PrintStream/println(Ljava/lang/String;)V
            iconst_1

            invokestatic java/lang/System/exit(I)V
 Label6:
:
```

The above Java byte code is generated from the database connection statement by the APL compiler. The code is in a special format that the Jasmin assembler can assemble to binary format. After the compiler finds out that the statement db = "SRAdb" is a database connection statement instead of assignment statement, it uses the string "SRAdb" to form the URL of the database "jdbc:odbc:SRAdb". Then the compiler generates the code that is needed to connect to the database and stores the reference of the connection. The compiler also generates the exception handling code automatically.

We can see that the only programmers concern in APL is to decide which APL knowledge base is to be used. After the programmer specifies the database name, the runtime system will be responsible for connecting to the actual database. The programmer does not need to write a code to load the database driver, connect to the database and catch an exception. Programmers only takes care of the APL knowledge base by giving its name ("SRAdb" in this example). This feature makes APL a great language for BDI manipulation.

The other two special statements are the knowledge access statement and the knowledge update statement. They are used to get the belief from the database and update the belief in the database. The translation rules are very similar for these two statements. We describe how to translate the database access statement first.

The code for the belief access statement is defined in the getBelief(…) plans of Belief stockBelief. It is:

i = get columName1 from tableName1 where symbol1=goal1;

This is a statement that we do not have in the Java programming language. This statement contains the information we need to construct a SQL statement. SQL is an embedded programming language that can be embedded in a lot of other languages. In this statement, "columName1" is the name of the field that we want to query from a database. "tableName1" is the name of the table in the database, "symbol1" is the name of the field in the database, which stores the stock symbol name, and "goal" is a string that stores the stock name that the user wants to get suggestions on. This information is provided by an agent. From this statement, the compiler can construct a SQL statement "select A from Knowledge where symbol = 'YHOO'". Then the compiler generates a code to invoke the executeQuery method, which is defined in the Java Data Base Connectivity (JDBC) API. Using this SQL statement to query the database, the agent gets the belief and stores the belief in the local variable "i". The following is the code generated from this statement by the compiler.

```
.catch java/sql/SQLException from Label7 to Label8 using
handler3
 Label7:
            aload_1
            invokeinterface
java/sql/Connection/createStatement()Ljava/sql/Statement; 1
            new java/lang/StringBuffer

            dup
            invokespecial java/lang/StringBuffer/<init>()V
            ldc "select "
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            aload 3
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            ldc " from "
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            aload 2
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            ldc " where "
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            aload 4
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            ldc " = '"

           invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            aload 5
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                        Ljava/lang/StringBuffer;
            ldc "'"
            invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                         Ljava/lang/StringBuffer;
            invokevirtual
java/lang/StringBuffer/toString()Ljava/lang/String;
            invokeinterface
java/sql/Statement/executeQuery(Ljava/lang/String;)
                          Ljava/sql/ResultSet; 2
            dup
            invokeinterface java/sql/ResultSet/next()Z 1
            pop
            bipush 1
            invokeinterface java/sql/ResultSet/getInt(I)I 2
 Label8:
            goto Label9
```

```
  handler3:
            pop
            getstatic java/lang/System/err Ljava/io/PrintStream;

            ldc "erro get data from DB"
            invokevirtual
java/io/PrintStream/println(Ljava/lang/String;)V
            iconst_1
            dup
            invokestatic java/lang/System/exit(I)V
  Label9:
:
```

Again, the programmer does not have to worry about how to use the Java Database Connectivity (JDBC) technique to get the belief from the knowledge base. In APL the programmer only specifies the belief he wants. The runtime system takes care of the knowledge access through the database manipulation. It is just like a real human agent, who does not need to know who calculates the data and how the data is transformed from one stock market to another. All he has to do is to make a call asking for the numbers. The APL works exactly in this way. It can simulate the real world more naturally than other programming languages.

The belief update statement is treated almost the same way as the belief access statement. The update statement is defined in the plan setBelief (…) in Belief stockBelief. It is: "set columName = value where symbol = goal;". Compared it with the code generated for the belief access statement, there are only two differences. Instead of making a Select SQL statement, an Update SQL statement is generated. The SQL statement for the above code is: "set A = 12 from knowledge where symbol = 'YHOO'". Then the method executeUpdate (sql) is the JDBC method used to execute the SQL statement. The rest of the code generated by the compiler is similar to the one from database access statement.

So far we have seen how nicely and naturally APL can manipulate the database. Now we show that the APL agent has a certain level of intelligence that can make it act like a real intelligent agent. This is done in the definition of the Desire. The following is the source code of desire component from the SRA system.

```
Desire recomDesire {
 belief<database> db1;
 belief<String> stockId;
 stockIntention stockIn;
 stockBelief stockB;
 public recomDesire (belief<String> g, stockBelief Bi,
                          stockIntention Ii) {
            db1=Bi.db;
            stockB=Bi;
            stockId = g;
            stockIn = Ii;
}

  public void run(){
       belief<int> result;
       belief<String> maptable;
       belief<int>b1=
stockB.getBelief(db1,"knowledge","A","Symbol",stockId);



       belief<int>b2=
stockB.getBelief(db1,"knowledge","B","Symbol",stockId);
       belief<int>b3=
stockB.getBelief(db1,"knowledge","C","Symbol",stockId);
       belief<int>b4=
stockB.getBelief(db1,"knowledge","D","Symbol",stockId);
       belief<int>b5=
stockB.getBelief(db1,"knowledge","E","Symbol",stockId);
       belief<int>b6=
stockB.getBelief(db1,"knowledge","F","Symbol",stockId);
            result = b1 + b2 + b3 + b4 + b5 + b6;
            maptable = "BPmap";
         reason {db1.maptable.belief(result)}
         using   {db1.maptable.stockIn()};
       }
   }
```

The Desire is translated to a Java class. But since we want the desire runing in its own thread, the desire is translated to inherit the Java Thread super class. In the Desire definition, at least one plan named "run (…)" needs to be created. This plan will be automatically invoked when the Desire is called.

The reasoning action is defined in the Desire definition. It is the reasoning procedure that makes agents intelligent. The BPmap table (Belief-Plan mapping table) plays a very important role during the reasoning process. By using the BPmap table, the agent can dynamically link to different plans at runtime. With the help of dynamic linking, we can modify the plans or even add new plans without halting the agent and recompiling the source program. We can ask an agent to achieve the goal in a better way by changing the links that is stored in the BPmap to the other plans. The dynamic linking makes APL a unique language.

The Desire first declares the belief that it needs to evaluate the goals and the Intention (action) it might need to achieve the goal. Then it calls the database access plan that is defined in the Belief to retrieve the beliefs from the table Knowledge. After calculating the beliefs (by add them all), the desire uses the result to get the plan names that are stored in the table BPmap. The program executes these plans and returns something to the caller if

necessary. The reasoning statement defines this process. In our SRA system, the reasoning statement is defined in the "run ()" plan in the Desire recomDesire. We describe how to translate it to the Java byte code here.

The reasoning statement in the SRA system is:

reason {db1.maptable.belief(result)}
using {db1.maptable.stockIn()};

This statement means by reasoning the value "result" in the table maptable (BPmap table), it invokes the plans specified in the intention field in the BPmap table. These plans are predefined in the Intention. In the SRA system, Intention stockIntention defines 5 plans: planA, planB, planC, planD and planE. The BPmap table contains links to these plans. It sounds very logical in the real world. It involves the database access steps and plan invocations. The compiler generates a SQL statement that looks like "select intention from BPmap where result >LowLimit and result <=HighLimit". The compiled code will connect to database and use this SQL statement to get the plan name stored in "intention" field.

Even though we have the plan name at this point, the execution of the plan is a little bit different. Since the program will not know which plan will be executed until an agent is really invoked, we need a special tool called the Java reflection to execute it. Reflection allows JVM to load a class and execute the method at runtime. So with the help of reflection we can finally execute the plan and achieve the goal.

Reflection is a powerful tool to make the dynamic link possible. With help of reflection, we can ask agents to load and execute any plan by specifies the intention and plan in the BPmap table. The following is the generated code from the reasoning statement that is showed above.

```
:
.catch java/sql/SQLException from Label10 to Label11 using handler4
  Label10:
    aload_0
    getfield recomDesire/db1 Ljava/sql/Connection;
    invokeinterface
java/sql/Connection/createStatement()Ljava/sql/Statement; 1
    new java/lang/StringBuffer
    dup
     invokespecial java/lang/StringBuffer/<init>()V
    ldc "select intention from "
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    aload 2
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)

                  Ljava/lang/StringBuffer;
    ldc " where UpLimit>"
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    iload 1
    invokevirtual
java/lang/StringBuffer/append(I)Ljava/lang/StringBuffer;
    ldc " and LowLimit<="
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    iload 1
    invokevirtual
java/lang/StringBuffer/append(I)Ljava/lang/StringBuffer;
    invokevirtual
java/lang/StringBuffer/toString()Ljava/lang/String;
    invokeinterface
java/sql/Statement/executeQuery(Ljava/lang/String;)
                      Ljava/sql/ResultSet; 2
    dup
    invokeinterface java/sql/ResultSet/next()Z 1
    pop
    bipush 1
    invokeinterface
java/sql/ResultSet/getString(I)Ljava/lang/String; 2
    astore 26
    new java/lang/StringBuffer

    dup
    invokespecial java/lang/StringBuffer/<init>()V
    ldc "public void stockIntention."
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    aload 26
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    ldc "()"
    invokevirtual
java/lang/StringBuffer/append(Ljava/lang/String;)
                    Ljava/lang/StringBuffer;
    invokevirtual
java/lang/StringBuffer/toString()Ljava/lang/String;
    astore 26
    ldc "stockIntention"
    invokestatic
java/lang/Class/forName(Ljava/lang/String;)Ljava/lang/Class;
    iconst_0
    anewarray java/lang/Object
    astore 29
    invokevirtual
java/lang/Class/getDeclaredMethods()[Ljava/lang/reflect/Method;
    astore 27
    bipush 0

   istore 28
   goto Labelwhile
  Labelcond:
    iload 28
    bipush 1
    iadd
    istore 28
  Labelwhile:
    aload 27
    iload 28
    aaload
    invokevirtual
java/lang/reflect/Method/toString()Ljava/lang/String;
```

```
    aload 26
    invokevirtual java/lang/String/compareTo(Ljava/lang/String;)I
    iconst_0
    if_icmpne Labelcond
    aload 27
    iload 28
    aaload
    aload_0
    getfield recomDesire/stockIn LstockIntention;

   aload 29
   invokevirtual
java/lang/reflect/Method/invoke(Ljava/lang/Object;
                    [Ljava/lang/Object;)Ljava/lang/Object;
    pop
  Label11:
    goto Label12
  handler4:
    pop
    getstatic java/lang/System/err Ljava/io/PrintStream;
    ldc "erro get data from DB"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    iconst_1
    invokestatic java/lang/System/exit(I)V
  Label12:
:
```

The intention is a set of plans that the agent may execute. A plan can be written in the similar syntax to the method in the Java. A plan has been translated to the Java byte code similar to the Java method. However, intention itself is translated into the Java class level. Plans are translated into the method level. The following is the source code of an intention that has one plan defined in it.

```
Intention stockIntention {
        public void planA (){

       System.out.println("recommand using plan A: Strong Sell");
        }
}
```

The following is the generated code by the compiler for the intention defined above. One of the differences between APL and other languages such as Java is that APL has a dynamic mapping mechanism via the Belief-Plan mapping table, BPmap, which provides intelligent behavior of the BDI agents, which Java does not support at all. The complex mechanism of the BPmap is hidden from the programmers, and programmers only need to concern about the definition of BDIs for intelligent agents. This concept well maps from models in agent-based design [Jo 01] [Einhorn and Jo 02] to the implementation in APL naturally.

```
.source Intention.j
.class public stockIntention
.super java/lang/Object
;SPECIFY THE CONSTRUCTOR METHOD FOR THE CLASS
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method
.method public planA()V
  .limit stack 30
  .limit locals 30
  getstatic java/lang/System/out Ljava/io/PrintStream;

  ldc "recommand using plan A: Strong Sell"
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```

We also need both a sever agent and a client agent to run the SRA system. The following are the two agents we defined in the SRA source program.

```
Agent stockAgent {
    belief<String> stockID;
    stockBelief  B1;
    recomDesire  D1;
    stockIntention I1;

    public stockAgent(belief<String>  goals) {
              stockID = goals;
              B1 = new stockBelief();

              I1 = new stockIntention();
              D1 = new recomDesire(stockID, B1, I1);
   }
}
Agent mainAgent {
   public static void main(String args[]){
      stockAgent a2 = new stockAgent("YHOO");
      a2.D1();
   }
}
```

The stockAgent is the server agent. It declares and creates an instance for the goals (Desire) it can achieve, the Belief (environment) it needs to analyze and the Intention (action) it can perform. The client agent mainAgent creates an instance of server agent, stockAgent. Then it asks the stockAgent to perform a goal recommendation “recomDesire” on “YHOO” by the statement “a2.D1()”. This statement will invoke the plan run() that is defined in the recomDesire definition, which will starts the process of recommendation.

We have shown this sample APL program to show how APL can implement BDI nicely and naturally and how we can translate it into the Java byte codes. The prototype compiler works well. We can add more useful features and develop a new runtime system for APL in the future work.

5.CONCLUSIONS

Agent-based computing is emerged as a new computing paradigm. The BDI model is one of the most powerful techniques to describe intelligent agents. In this paper we have introduced a new agent-based programming language, APL, and its compiler based on the BDI agent concept.

This work shows how nicely and naturally APL can describe the behaviors of intelligent software agents. By constructing the compiler, we have a real executable APL program that can run on the Java Virtual Machine. APL provides a nice and easy, Java like syntax, which makes it easy to learn and more efficient to define software agent. APL will use database to store the knowledge, but access to the database is hidden from programmers. So programmers can focus on defining agents instead of writing codes accessing the database and catching the exception. This feature makes APL more natural than other languages to describe intelligent agents. The use of BPmap and dynamic link technique enables APL to execute different plans at runtime without recompiling the source program. It makes APL agents more flexible and more intelligent.

In the future work, we need to improve the dynamic link technique. So the agent can automatically find parameters for the new plan and execute it right. We can add more learning techniques such as the Belief-Desire table, so that the agent can change its goal at runtime. We also need to develop a new runtime system solely for APL that can support BDI agent-based programming more naturally.

6. REFERENCES

[1] Bratman, Michael E., Intention, Plans, and Practical Reason, Harvard Univ. Press, 1987 (also by CSLI Publication, 1999).

[2] Busetta, Paolo and Ramamohanarao, Kotagiri. The BDIM Agent Toolkit Design, Technical Report 97/15, Department of computer Science, The University of Melbourne, Australia, 1997. http://www.cs.mu.oz.au/publications/tr_db/TR.html.

[3] Busetta, Paolo and Ramamohanarao, Kotagiri. An architecture for Mobile BDI Agent, Mobile Computing Track, ACM SAC '98, 7-8, 1998.

[4] DeLoach, Scott A. Multiagent Systems Engineering: A Methodology and Languages for Designing Agent Systems, http://en.afit.af.mil/ai/publications/Conference/aois-99/MaSE-AOIS99.htm, 1999.

[5] Einhorn, Jeffrey M. and Jo, Chang-Hyun. A BDI Agent Software Development Process, University of North Dakota, 2002.

[6] Feng, Xin and Jo, Chang-Hyun. Design and Simulation of an Agent-based Stock Trader, University of North Dakota, 2001.

[7] Franklin, Stan and Graesser, Art. Is it an Agent, or just a Program? : A Taxonomy for Autonomous Agents, http://www.msci.memphis.edu/~franklin/AgentProg.html, Also in the Proc. of the 3rd International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 2-7, 1996.

[8] Huber, Marcus J. JAM: A BDI-theoretic Mobile Agent Architecture, Proc. Of the Autonomous Agents '99, Seattle, USA, 236-243, 1999.

[9] JACK, Agent Oriented Software Pty. Ltd., JACK Intelligent Agents User Guide, http://www.agent-software.com.au, 1999.

[10] Jo, Chang-Hyun, A Seamless Approach to the Agent Development, ACM SAC 2001 Conference, Las Vegas, 641-647, March 2001.

[11] Jo, Chang-Hyun and Arnold, Allen J., Agent-based Programming Language: APL, ACM SAC 2002 Conference, Madrid, Spain, 27-31, March 2002.

[12] Meyer, J. and T. Downing, *Java Virtual Machine*, O'Reilly, 1997.

[13] Petrie, Charles J. Agent-Based Engineering, the Web, and Intelligence, http://www-cdr.stanford.edu/NextLink/Expert.html, also appeared in the IEEE Expert, (December 1996).

[14] Yahoo Stock Quotes, http://finance.yahoo.com, 2000.