A COMPILER DESIGN FOR THE AGENT-BASED PROGRAMMING LANGUAGE

Wei Zhao
Department of Computer Science
University of North Dakota
Grand Forks, ND 58203, USA
zhao@cs.und.edu

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 [2, 3, 4, 5, 7, 8, 10, 13]. The BDI model [1] 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 [11]. 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, 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.

--
ISCA CATA-2003
March 26-28, 2003, Honolulu, Hawaii

This paper describes the basic structure of the APL system. A sample source program has been developed based on the Agent-based Stock Trader (AST) system [6]. 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 the agent can give clients some useful recommendation and help clients to make the right decisions. The agent has own knowledge base. 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. To simulate a real time stock market, we use an example database.

APL is a new programming language designed to describe the behaviors of the intelligent software agents. 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 its own 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. Our APL compiler translates the APL source program into the Java byte code, which can be executable on the JVM with a database to manipulate the BDI knowledge base. We use special tools such as LEX and YACC to develop the APL compiler.

2. Background of BDI-Agents
A few research have been done in agent-based programming languages [2, 3, 4, 5, 7, 8, 10, 13]. The BDI model [1] 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 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. BDIM (Belief, Desire, Intention, Message) Agent Toolkit is implemented as a Java package to provide a prototype of runtime architecture [2, 3]. JACK Intelligent Agent is an environment for building, running and integrating commercial-grade multi-agent systems using a component-based approach [9]. JAM (Java Agent Model) is an implementation of the BDI agent model in Java [8]. All of these works are based on Java. But they are not 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. BDIs and agents can be defined as reusable components just like classes. BDIs can be declared in agent definitions. 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. Implementation of BDI-Agents in 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 [11]. In order to make our APL easier to read and learn, we defined the APL language in a Java-like syntax. The Java Virtual Machine (JVM) 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 executable on the JVM.

We develop the *Stock Recommendation Agent (SRA)* system in the APL language based on the AST system that was previously developed in Java [6]. 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. The database serving for the SRA system is similar to the database of the AST system [6]. Two tables are necessary to support the SRA system. The table, Knowledge, stores the knowledge of the agent. Another table, BPmap, stores the information about mapping from belief to plans that the agent executes.

The agent defined in our SRA system gives the client agent certain recommendations on stock trading. This software agent works in the same way as a real stock agent does. After evaluating all the environmental information carefully, the stock agent suggests that the customer would better buy, sell or hold the stocks. The SRA can simulate this intelligent work. The server agent defines a recommendation desire, so that the server agent can give the client agent a recommendation on stocks by achieving this goal. 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.

There is another knowledge in the SRA system called Bpmap. BPmap contains the plan information that the agent needs to perform in order to achieve the goal. When a 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)" which is defined as a desire of the server agent. 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, the server agent refers to the mapping table (BPmap) that shows which plan is suitable at this time. Once a chosen plan returns a recommendation result, then the server agent returns a recommendation to the client agent based on the execution result of the plans. We assume that there can be a supporting agent that is designated to update the knowledge base when those factors are changed. The administrator or the agent designated to maintain the mapping table updates the BPmap if there is a better plan available. 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 in this example: stock symbol, six belief values, and belief value. 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 [14]. Six components of knowledge A through F represent beliefs related to the stock. In our SRA system, we let the system administrator update the knowledge table. But in the real applications, we can designate other agents to update the knowledge. The beliefs the agent gets from the knowledge can be calculated in different ways depending on the algorithm used in a particular application. 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. The purpose to have this example application is just showing how APL handles the BDIs. Therefore, the algorithm we chose to analyze the belief values is just trivial in this example. Here we just add up all belief values from A to F. However, in the real application, appropriate mathematical functions and stochastic methods for the stock market forecasting can be used.

4. Translation of APL Code to Java Byte Code

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 [12] to assemble the Java byte code into the binary executable class file that is directly executable on the Java Virtual Machine.

The *Belief* part of APL is translated into a class as a reusable component, and each plan defined in the Belief is translated into a Java method in the Java byte code. 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.

```
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 tableName1where symbol1=goal1;
            return i;
         }
   }
```

In APL, all variables are the belief type. A belief has its definition: "belief<data type> variable name". The "data type" can be any primary data type or "database". If the date type 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. We give special treatment to these statements.

With the Belief definition, we have three special statements such as the knowledge base connection statement, the knowledge base access statement and the knowledge base update statement. The first special statement is the APL knowledge base connection statement that is used 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 for the APL knowledge 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 knowledge connection statement by the APL compiler. The code is in a special format that the Jasmin assembler can assemble it to binary code. After the compiler finds out that the statement db = "SRAdb" is a knowledge base connection statement instead of assignment, 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 to the connection. The compiler automatically generates the exception handling code that is necessary for database manipulation.

We can see that 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. This feature makes APL a convenient language for BDI manipulation.

The translation rules are very similar for these two other statements – knowledge base access and update. 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 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. 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".

We allow an 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 code is the source of the desire component of 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);
     :
     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 as a reusable component. However, since we want the desire running with its own thread, the desire is translated to inherit the Java Thread. 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 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 alternative plans. The dynamic linking makes APL a unique language among many agent-based languages.

The Desire first declares the belief that it needs to evaluate the goals and the Intention 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. 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), 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 several plans such as planA, planB, planC, planD and planE. The BPmap table contains links to these plans. Mapping is done through the BPmap table at runtime. 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 will use a special mechanism 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 corresponding plan and achieve the goal.

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 reusable class. 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");
    }
    .
}
```

We also need both a *sever agent* and a *client agent* to run the SRA system. The prototype compiler works well enough for this application. We may add more features 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 we built based on the BDI agent concept.

This work shows how naturally APL can describe the behaviors of intelligent software agents. By constructing the compiler, we have a really executable APL program that can run on the Java Virtual Machine. APL provides an 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.

An updated research with source codes is located at our web page at http://www.ecs.fullerton.edu/~jo.

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, 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.