An interpreter for 3APL

Submitted April 20, 2023, in partial fulfillment of
the conditions for the award of the degree
BSc Computer Science with Artificial Intelligence.

Ming Kai
20126414

Supervised by Yuan Yao

School of Computer Science University of Nottingham Ningbo China

Acknowledgements

First, I would like to express my sincerest thanks to my supervisor, Dr. Yuan Yao, for his patient guidance, and advice for a year. Without him, I would not have been able to successfully complete my project and it has been an honor to have him directing this project. Secondly, I would like to thank all my friends, especially my roommates Weikai Kong and Shuhong Ye, for their technical advice, In addition, and my parents for their silent support. Finally, I would also like to thank every teacher in the School of computer science who had taught and helped me.

Abstract

Agent-oriented programming is a programming paradigm in which intelligent agents and their interactions within a system. An agent is typically a software entity that can sense its environment, reason about it, and take actions to achieve goals. Agent-oriented programming often uses multi-agent systems, where multiple agents work together to achieve a common goal or solve complex problems. Today, agent-oriented programming has been applied in many fields, starting with robotics and social networks. 3APL is a unique agent programming language, but 3APL has only one interpreter, so all its functions cannot be realized, so a new interpreter will be developed for 3APL. This project aims to implement a new interpreter for 3APL. Such an interpreter can easily compile 3APL files and also realize the interaction between multiple agents.

Contents

List of Figures

List of Abbreviations

3APL An Abstract Agent Programming Language

APL Agent-oriented programming language

BDI Belief-Desire-Intention

EBNF Extended Backus–Naur form

GUI graphical user interface

MAS Multi-agent system

MVC Model–view–controller

PG rule Goal planning rule

PR rule Plan revision rule

Chapter 1

Introduction

1.1 Background

What is an intelligent agent? In computer science and artificial intelligence, an agent is an object that senses its environment and takes action to achieve a specific goal[3]. Furthermore, the agent is autonomous and can make decisions independently without human intervention[4]. Through the above definition, it can be found that many software entities are intelligent agents, For example, during the exploration of Mars, because the communication time between Mars and the Earth is too long, people deploy intelligent agents for the Mars rover. In the absence of human instructions, the intelligent agent of the Mars rover will choose the target by itself and execute the plan, thereby effectively improving the efficiency of scientific research projects[1].

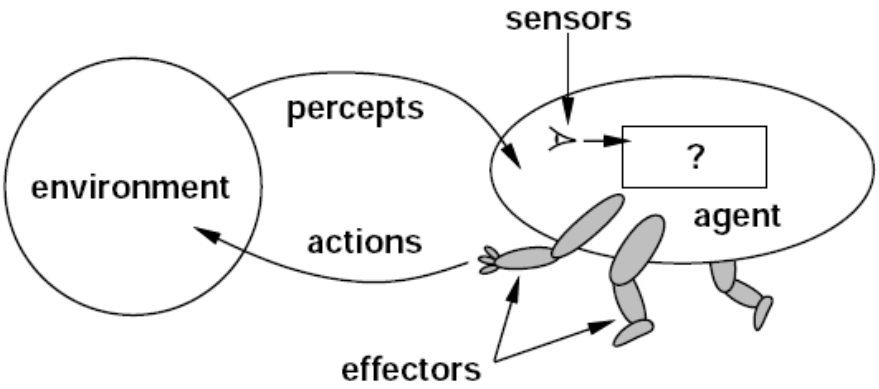

Figure 1.1: Structure of agent, adopted from[1]

Agents are often applied to multi-agent systems. A computer system consisting of multiple interacting intelligent agents is a multi-agent system[5] where multiple agents work together to achieve a common goal or solve complex problems. Because in real-life scenarios, the problems to be solved are too complex, and it is difficult to use a single agent to solve complex problems. In a multi-agent system, multiple agents can exist in the same environment and are given the ability to interact with each other, for example, in a

traffic management system, traffic lights and road detectors can be considered as multiple agents, these Agents can work together to reduce congestion.
BDI is an agent-oriented programming paradigm that drives the behavior of agents by setting beliefs, desires, and intentions for agents, simulating human behavior. BDI consists of three core components: Belief, Desire, and Intention. Beliefs represent the agent's observations and understanding of the world, desires represent the goals the agent wishes to achieve, and intentions represent the specific actions the agent takes to achieve the goals[6]. The second chapter will explain in detail the operation process of the BDI agent. BDI agents are able to adaptively respond to changing environments. This makes the BDI programming paradigm has been applied to many fields, including artificial intelligence and robotics.

3APL (3APL) is an agent-oriented programming language proposed in 1999[7]. It is conceived as an APL for intelligent agents built around belief-desire-intention (BDI).

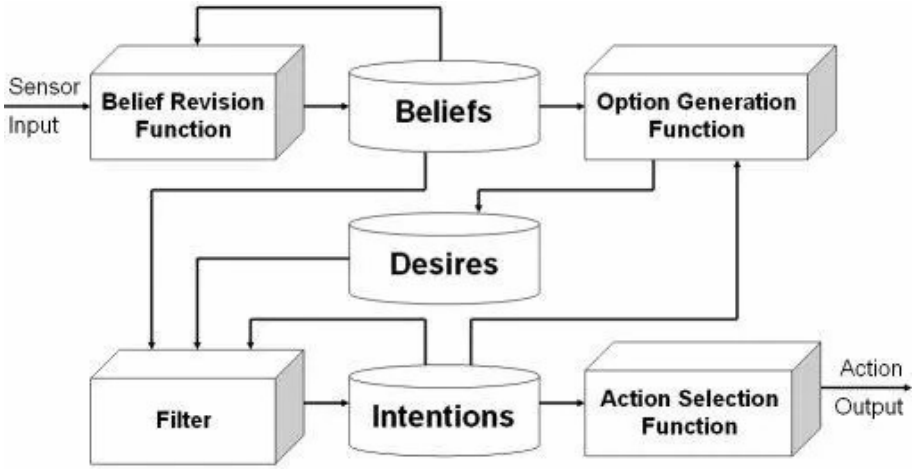

Figure 1.2: BDI model structure, adopted from[1]

APL can be used to model complex systems, simulate intelligent agents, and effectively solve complex problems in reality. These are achieved through cognitive agents. In general, a cognitive subject is considered to have a mental state consisting of mental attitudes such as beliefs, goals, plans, and rules of inference [6][8][9]. Furthermore, the actions performed by a cognitive subject, i.e. the actions and plans it chooses and performs, are said to be determined by taking into account its own mental attitudes [10][11][12][13]. Therefore, agents can be written through the new programming paradigm. Agent-oriented Programming Language (APL) is a programming language through which people can write intelligent Agents (agent programs). Agents simulate the behavior of intelligent agents, enabling them to complete specific tasks. Existing APLs include AgentSpeak, 3APL, Jason, etc.
By applying the BDI model to the Agent, the updating of the Agent after running is re-

alized. Furthermore, once a 3APL agent chooses a goal to achieve, it remains unchanged until the goal is achieved[14]. Unlike other APL, 3APL allows agents to modify compiled code(its own base) at run-time.
This project will develop a new 3APL interpreter, which can compile 3APL code into an agent. A graphical user interface will display the different bases and running processes of intelligent agents, and agents will be able to communicate with each other in a multi-agent system.

1.2 Motivation

3APL is an agent-oriented programming language for developing intelligent agents, whose core features are supporting the autonomy, reactivity, sociability, and rationality of agents[1]. However, the existing 3APL interpreter (APL platform) mentioned below, last updated in 2007, supports an outdated version of Java (Java 1.6), which may cause several issues such as:

1. Security issues: Outdated versions of Java may have vulnerabilities that can lead to security issues, including attacks on the agent's runtime environment.
2. Compatibility issues: Outdated versions of Java may not be compatible with the latest Java applications and libraries, which can cause agents to be unable to interact with other applications.

Therefore, it is necessary to develop a new 3APL interpreter to solve these issues and provide a better experience for users. The new interpreter will try to use newer versions of the Java platform and related technologies to improve interpreter security and compatibility. In addition, a new interpreter can provide better debugging tools to help users debug and test agent files more easily.

1.3 Aim and Objective

The main aim of this project is to develop a new 3APL interpreter on a new version of Java, implement the basic functionality of the 3APL platform, and develop some new functionality. Due to the lack of time remaining in the project, some of the original features of the 3APL platform were not implemented.

1. Develop an interpreter for 3APL that supports all features of the language.

2. Develop and implement the multi-agent system.

3. Implement communication between different agents.

4. Implement a way to interact between the environment and the agent.

5. A graphical user interface will be developed and users will be able to load and stop any agent.

6. The operation's information of the agent will be visualized in the graphical user interface.

In addition, to demonstrate the 3APL interpreter, the project will design and develop several examples of 3APL agent for illustration. The example should include interactions and communications between multiple agents and show how agents react to the environment to illustrate what the interpreter implements.

1.4 Contributions

The main contribution of the project can be summarized in two aspects.

First, the new interpreter makes the 3APL compatible with newer versions of Java, providing the benefits of security and compatibility. The new interpreter can help avoid potential security vulnerabilities associated with outdated Java versions. By updating the interpreter to match the updated Java version, the project ensures that 3APL users can take advantage of the updated Java features and technologies to improve their 3APL agent applications.

Second, the project provides a simplified graphical user interface(GUI) to facilitate the management and testing of 3APL agents. With the new GUI, users can easily create and test 3APL agents without requiring a lot of agent-oriented programming knowledge. The GUI provides intuitive visual tools to manage agents, monitor their behavior, and debug potential problems.

Overall, these two major contributions help improve the 3APL interpreter experience for new users.

1.5 Dissertation Outline

This dissertation mainly includes three parts:

1. The first two chapters address the issues with existing 3APL interpreters and provide technical background and related work on this project.

2. In Chapters 3 and 4, the article delves into the design and implementation of the new 3APL interpreter. These sections provide a detailed overview of the new interpreter architecture, highlighting its various components and capabilities. In addition, these chapters detail the parsing and lexical analysis of 3APL code, which is a key part of the interpretation process.

3. In the last two chapters(Chapter 5,6), several concrete 3APL examples are introduced to demonstrate all the capabilities of the compiler. The reader will know what the project has accomplished, what went wrong during the development of the compiler, what features are not currently implemented, and what the project plans for the future.

Chapter 2

Background & Related Work

This chapter will provide an overview of the background knowledge and related products of 3APL, as well as the technologies applied in this project. Specifically, we will discuss the theoretical foundations of agent programming, such as the BDI model and 3APL. it will also introduce the existing interpreter for 3APL and its limitations of it. Additionally, we will discuss the specific technologies used in the development of the 3APL interpreter's GUI, including programming languages, software libraries, and development tools.

2.1 Belief-desire–intention software model (BDI)

The philosophical view of the concept of Belief-Desire-Intention (BDI) was proposed by Bratman[15]. BDI software model is a representation method for the composition of an agent, which divides the composition of an agent (agent) into three components: belief, desire, and intention. An agent modeled on the BDI selects and executes specific actions based on current beliefs, desires, and intentions to achieve its goals. Among them, belief represents the agent's observation and understanding of the world (knowledge), desire represents the goal that the agent hopes to achieve, and intention represents the specific action taken by the agent.

When the agent observes new information (in the environment), its beliefs are updated accordingly; when the agent's desires change, its intentions are adjusted accordingly. After updating the belief base, the agent chooses a goal from the desires according to its expectations on itself. Then, according to the goal selected from the desire, a plan is selected from the intention to implement. The output is the execution plan (action). Finally, the agent's beliefs, wishes, and intentions are all updated after the actions are implemented [1]. In this way, BDI agents can adaptively respond to environmental changes and task demands.

2.2 Prolog

3APL is a multi-agent system programming language based on Prolog. Prolog is a programming language whose name comes from the abbreviation of "Programming in Logic". The main feature of Prolog is its logic-based programming approach, where problems and knowledge are represented using rules and facts, and then a backtracking search algorithm is used to solve the problem.

In Prolog, programmers can define facts and rules, which can be used to deduce answers to problems. Prolog uses a method called "backward chaining" to perform inference. This algorithm allows Prolog to reason from a goal, and search for a solution by continuously applying rules and querying facts.

2.3 3APL language

In this project, the 3APL language version used is the same as the paper[14] by Mehdi Dastani et al. The two most important parts of 3APL are the agent's base and the deliberation cycle. These two parts will be briefly introduced below.

2.3.1 Agent

3APL agents consist of languages expressing beliefs, basic actions, goals, and practical inference rules [16][17]. It takes the BDI model as the core structure. A standard 3APL agent has six components, which are capability base, belief base (Belief in BDI), goal base (Desire in BDI model), plan base (Intention in BDI), plan revision rule(PR-rule) base, and goal planning rule(PG-rule) base. These six parts are described in detail below:

- capability base: The capability base is a collection of basic operations that agents can perform.
- belief base: Beliefs represent knowledge obtained by an intelligent agent and describe the agent's situation and environment. It can contain information about the surrounding environment that the agent considers, as well as information inside the agent.
- goal base: The goal base is a collection of goals to be achieved by agents.
- plan base: The plan base is a collection of actions that an agent will perform that

will help the agent achieve a goal.

- plan revision rule base: the plan revision rule base is a collection of rules that agents applied to adjust the plan base based on current beliefs.
- goal planning rule base: the goal planning rule base is a collection of rules that agents applied to determine which plan to execute based on its goal base and belief base.

In addition, the agent usually has a prolog file, which is a supplement to the agent's base, and the agent will search for this part at runtime.

2.3.2 Deliberation Cycle

The operation of the 3APL agent can be realized through the deliberation cycle, which is the expression of deliberation language[18]. In this section, the deliberation cycle will be introduced to help understand the operation of 3APL agents.

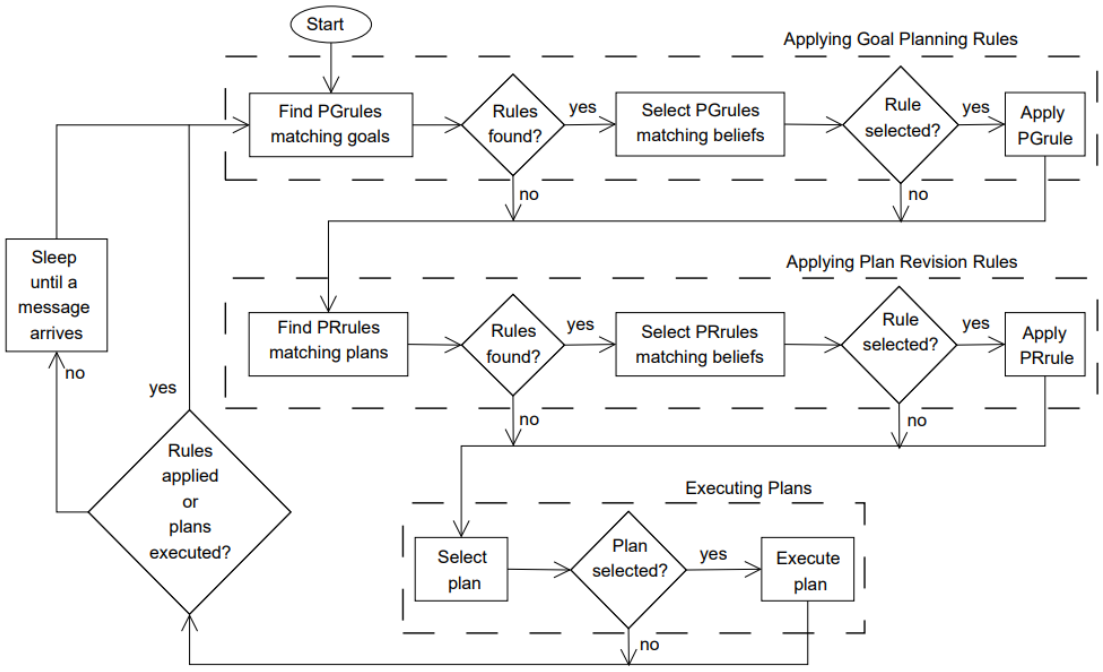

Figure 2.1: The cyclic interpreter (deliberation cycle) for 3APL agents

The deliberation cycle mainly consists of five steps The first step is the initialization of the agent. After the code is compiled, each base of the agent is successfully compiled. The second step is to apply the goal revision rule (PG rule). In this step, the compiler

needs to search for the goal and belief that match the PG rule in the goal base and belief base successively. If the goal and belief that match the PG rule are selected, Then the plan base is updated, if not, go to the third step.
The third step is to apply the plan revision rule (PR rule). In this step, the compiler needs to search for the plan and belief that match the PR rule in the plan base and belief base successively. If the matching plan and belief are selected, Then the plan base is updated, if not, go to the fourth step.
The fourth step is to apply the plan. In this step, the compiler needs to search for a plan in the plan base successively. If the plan is selected, each base of the agent will be updated (belief base, goal base, plan base). If the plan base is empty, then go to the fifth step.
The fifth step is to check whether there is a plan or PG/PR rule applied, if so, jump to the second step, otherwise stop the review cycle.

2.4 Agent interpreter

An agent interpreter is an interpreter designed to execute agent programs. Unlike other interpreters that execute programs written in a specific language, agent interpreters are designed to execute programs written in agent-oriented programming languages.

The main advantage of the Agent Interpreter is that it provides built-in support for agent-specific concepts such as beliefs, wishes, and intentions (BDI). This makes it easier to develop and execute agents compared to normal interpreters. Furthermore, agent interpreters are designed to support agent communication and interaction, a key aspect of multi-agent systems.

Some examples of proxy interpreters include JACK and 3APL Interpreter. JACK is a popular agent interpreter used to develop and execute BDI agent programs. 3APL Interpreter is an interpreter specially designed to execute programs written in the 3APL agent programming language.

2.5 Java swing

In this project, we will be using Java Swing to develop a new GUI interface for the 3APL interpreter. Java Swing is a powerful GUI toolkit for designing desktop applications in Java[19]. One of its major advantages is it can create GUIs that run on different

operating systems. Additionally, Swing provides a wider range of GUI components and layout managers than the earlier Abstract Window Toolkit (AWT)[19], making it easier for developers to create better interfaces. The use of Swing in this project will enable us to provide a more simplified and intuitive user experience for the 3APL interpreter, making it easier for users to manage and test their 3APL agents.

2.6 Related Work

3APL was first proposed in 1999. there are only two base cases(belief and plan) in 3APL agents. But in 2003, 3APL was updated, A standard 3PAL agent is composed of six components: belief, goal, capability, planning rule, plan revision rule, and goal revision rule[14]. This is also the version used in this project.

Currently, the 3APL interpreter currently implemented is best referred to as the 3APL platform[20]. It was developed in 2003 by a team at Utrecht University. It provides a graphical interface through which users can develop and execute 3APL agents using tools (syntax-colored editor and several debugging tools), and the 3APL platform implements a multi-agent environment that allows agents to communicate with each other.

Chapter 3

Design

In this section, we will describe how to design a new 3APL interpreter for the project. Typically, the design of an interpreter can be divided into three parts: the multi-agent system, the interpreter itself, and the software user interface. In this project, we will implement the multi-agent system using multithreading and design and implement the 3APL interpreter in the Model-View-Controller(MVC) architecture.

3.1 Agent

In a multi-agent system, the most fundamental component is the agent. A standard agent structure consists of six components, and its detailed syntax is in the appendix. In the code of this project, the design of the applied agent follows the same structure, which means that each component of the implementation corresponds to a component of the agent in the language. Therefore, the grammatical structure of these components must be implemented in the code. For example,

- if statement: implemented by the ifAction class
- Belief: implemented by belief class
- Plan revision rule base: implemented by the PlanRevisionRule class

ANTLR4 is a tool for building language parsers, and compilers. It generates parse trees by recognizing tokens and syntax structures in the code and can be used to build custom programming languages and parsers. Advantages of ANTLR4 include ease of use and learning, high performance and flexibility, and support for Java. In this project, ANTLR4 is used to build the 3APL interpreter, which generates parse trees from 3APL code and is used for the implementation of the interpreter.

In addition to these necessary components in 3APL. The interpreter should recognize

the identifier of the agent file and also enable the agent to be able to interact with other agents. Only in this way can we successfully build a multi-agent system and make multiple intelligent agents in it run normally. Finally, the Prolog reasoning process in the agent is the core of the execution process in the agent. The Prolog inference process in this project is provided by an open-source library called JIProlog. JIProlog is a Prolog interpreter based on the Java platform, which supports mixed programming between Java and Prolog, provides standard features of Prolog, and also provides some advanced features, such as multi-threading support and integration with the Java Swing GUI framework Integration makes JIProlog very suitable for developing intelligent agent-oriented applications.

3.2 Multi-agent system

Through the application of JIProlog, the compiler supports multi-threading function, so in order to realize the multi-agent system, this project regards each agent in the software as a thread to realize, and simulate the multi-agent system by running multiple threads at the same time, and then Realize the interaction between agents. Simultaneously running threads all exist in the agent controller poll, which can be regarded as MAS. Additionally, threads can be attached to environments containing controllable entities controlled by agents. To simplify this model, we separate the server and the environment into two parts of the multi-agent system, which are connected by the environment interface.

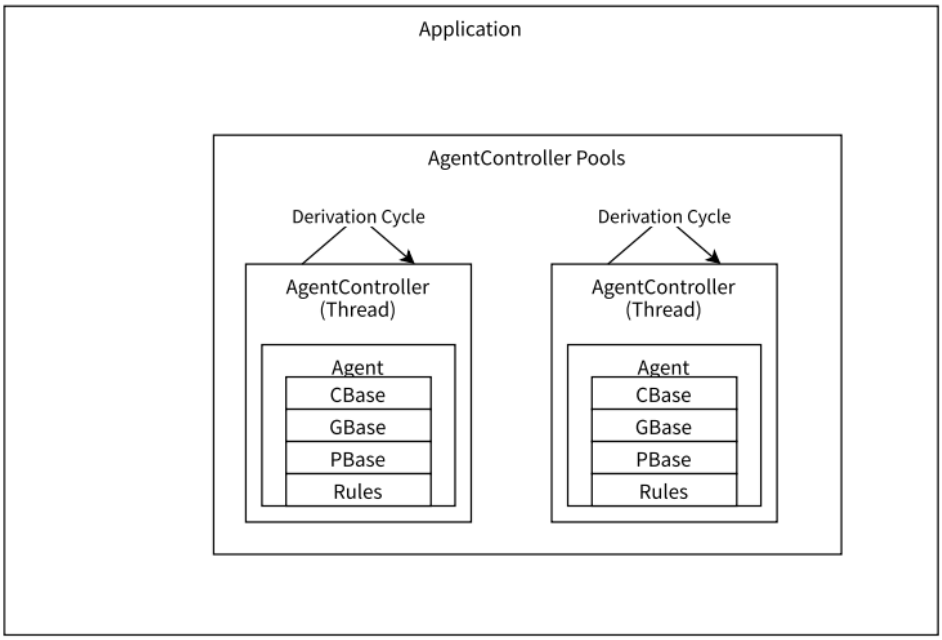

Figure 3.1: Threading model of multi-agent system

3.2.1 Environment

The environment is used to represent the world in which an agent in a multi-agent system lives. Environments can be updated to contain multiple objects. These objects are mutable, therefore, all actions the agent causes to the environment should be performed by these objects. The environment should not contain any information about proxies, and the agent should not contain any information about objects. Therefore, add-ons to the environment are introduced, thereby making it easier for programmers to write code.

3.3 Interpreter

3.3.1 Interpreter structure

The interpreter is the logical part of the software. It takes input files or instructions from the GUI and processes them. For the interpreter, this interpreter adopts the MVC mode to facilitate the normal operation of 3APL. MVC (Model-View-Controller) is a software architecture design pattern.

- Model (Model): Represents the data and business logic of the application. Models contain code for data storage, data validation, business rules, and data access, but not for user interface or user interaction.
- View: Displays the user interface of the application. A view can be a form, a page, or a control. A view typically takes data from a model and presents that data to the user. it can be regarded as GUI.
- Controller (Controller): Receives user input from the view, and then calls methods in the model to handle user requests. The controller is also responsible for updating the view to reflect changes in the data in the model.

In this project, the user should input information to the controller through buttons on the view interface as events. The interpreter only accepts one possible external input, which is the 3APL code. Two controllers are designed in this project, one mainly corresponds to the events the user uses with the GUI, and the other is responsible for executing the agent's deliberation cycle. The 3APL code will be provided to the controller, and ANTLR4 will be used to parse the input 3APL code. Inputs that do not conform to the syntax will not work normally, and then a separate class will handle syntax errors.

In this project, the Model stores the various elements and operation rules of 3APL agent. After obtaining the 3APL code, the controller should retrieve data from the Model and execute the relevant logic to run the 3APL agent, and then update the view. Finally, the update of the view is visualized on the GUI, and the user obtains information on the operation of the agent.

3.3.2 Graphical user interface

The GUI design of the interpreter is divided into three parts:

- Main page: This is the main interface of the 3APL interpreter, including all operations that need to be performed, including loading agent/running agent/stopping agent operation/removing agent environment. It also allows users to control the 3APL agent and open other pages.
- Agent status page: This page will display the running information of the selected 3APL agent, including three parts: each base of the 3APL agent, the specific process of the deliberation cycle execution, and the information involved.
- Environment page: Part of the agent operation will change the environment in which it is located. This page will visualize the operating environment of the selected 3APL agent and show the changes in the operating environment.

Chapter 4

Implementation

In this section, because the project is written with the MVC architecture as the core, we will explain how to implement the 3APL interpreter according to the MVC structure. Model is the entity and state that the program runs, View is mainly the GUI of 3APL, Controller is the control logic of the program, and application is the entry of the program.

4.1 Syntax parser/Antrl4

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text. ANTLR generates a parser from the grammar definition, and the parser can build and traverse parse trees. Avoiding that tedious lexical analysis, the main work becomes describing the grammar using Extended Basco-Naur Form (EBNF).

In this project, IntelliJ IDEA will be used as a development tool. We can search for antlr4 in Preference-Plugins and install it.

First of all, we need to define the grammar and lexical structure of 3APL, which requires creating a .g4 file to define the lexical analyzer (lexer) and parser (Parser). For the specific grammar of the g4 file, please refer to the official document [21]. In this project, we created APLGrammar.g4. After creating and writing this file, right-click the .g4 file in IDEA and select Generate ANTLR Recognizer. The plug-in will automatically generate a bunch of Java codes in the gen directory, and we need to move them to the corresponding package. If @header is defined, IDEA will also automatically generate package information.

APLGrammar.g4 defines a grammar called APLGrammar. The grammar rules included in the grammar will be introduced below:

- program: Defines the overall structure of the program, including identifiers, capa-

bilities, beliefs, goals, plans, and other elements.

- capabilities: Defines the syntax rules for capabilities, including capability queries and capability execution.
- belief: Defines the syntax rules for beliefs, including atomic and literal types.
- goal: Defines the syntax rules for goals, including single and multiple goals.
- plan: Defines the syntax rules for plans, including basic and composite actions.
- p_rule: Defines the syntax rules for PG-RULES, which specify how plans are executed.
- r_rule: Defines the syntax rules for PR-RULES, which specify how plans respond.
- literal: Defines the syntax rules for literals, including atomic and non-atomic types.
- wff: Defines the syntax rules for logical formulas, including atomic formulas, conjunctions, and disjunctions.
- query: Defines the syntax rules for queries, including logical formulas and TRUE.
- iv: Defines the syntax rules for variables, including identifiers and plan variables.
- ident: Defines the syntax rules for identifiers.
- var: Defines the syntax rules for variable names.
- atom: Defines the syntax rules for atoms, including atom names and parameter lists.
- argument: Defines the syntax rules for parameters, including plan variables and atoms.
- aatom: Defines the syntax rules for atomic actions, including plan variables and parameter lists.
- pvarval: Defines the syntax rules for plan variables and plan constants.
- pname: Defines the syntax rules for atom names.
- pval: Defines the syntax rules for plan constants.

- Keywords: Defines the keywords and symbols of the language, including PROGRAM, LOAD, CBASE, BBASE, GBASE, PBASE, PGRULE, PRRULE, QUOTE, SEMI, LBRACE, RBRACE, DOT, BARROW, COMMA, AND, OR, LPAREN, RPAREN, SEND, JAVA, QUESTION, IF, THEN, ELSE, WHILE, DO, SEMICOLON, PARROW, ORS, NOT, TRUE, IDENT, CAPVAR, NAMEVAR, ALPHANUMERIC, ALPHA, DIGIT, and WS.

4.2 Model

Model, which is the core of the 3APL interpreter, is responsible for processing data. In this project, the files in the Model correspond to each element in the grammar, and they are parsed into each element. The following is a brief introduction to how to implement the model by introducing several model files:

4.2.1 Atom

This Java class represents an atomic formula, also known as an atom, in a logic program. It has three fields: name, which is a string representing the predicate name of the atom; args, which is a list of Argument objects representing the arguments of the atom; and substitute, which is a method that takes a Hashtable object representing variable bindings and substitutes any variables in the arguments of the atom with their corresponding values in the Hashtable.

The toString method of the Aatom class returns a string representation of the atom in the form "name(arg1,arg2,...)". The args list is converted to a comma-separated string using the map and collect methods.

4.2.2 PlanBase

The Java class PlanBase, which implements the Serializable interface, enables instances of this class to be serialized and transmitted. This class has a member variable plans of List¡Plan¿ type, which is used to represent the plan list. In the class's constructor, initialize the list to an empty ArrayList. This class uses the toString method to convert the plan list into a string form, and uses the String.join method to join the strings in the list.

4.3 Controller

In this section, we will introduce how to implement the controller. In MVC, Controller is the component responsible for processing user input and converting it into operations on models and views. The controller processes user events from View and forwards these events to Model for processing. This project's controller consists of five parts, which will be introduced below.

4.3.1 Main Controller

The first thing to implement is the MainController. MainController class is implemented by Java Swing. There are many different operations contained in MainController to manage and monitor the behavior of the agent. MainController creates a MainView object that represents the Main Page of the application's GUI. In addition, MainController also creates a hash table (Map(String, AgentController)) containing all agent controllers, which can be indexed by the name of the agent.
In the MainController class, start() will cause each operation to be associated with a component in the MainView object, be it a JButton, JMenu, or JTable, respectively. When the user performs related operations (such as clicking these buttons) in the GUI/View, the corresponding method in MainController will be called. They are briefly introduced below:

- onCreateAgent(): The user triggers this action by clicking the "Create Agent" button or selecting the "Create" menu item from the File menu. Opens a JFileChooser dialog, allowing the user to choose an agent file. After the user selects the file, the file is parsed and the agent objects it contains are added to the hash table and a new agent controller object is created. If the file is parsed incorrectly or cannot be opened, a pop-up window will display the wrong type of information. This agent controller will run in its own thread and manage all activities of the agent.
- onCheckStatus(): The user selects an agent in the agent list and clicks the "Check Status" button to view the information of the currently selected agent. This action creates an AgentStatusView object to display the agent's name, capabilities, beliefs, goals, plans, and deliberation log. If no agent file is selected or loaded, an alert window will pop up.
- onStart(): The user selects an agent in the agent list and clicks the "Start Agent" button to start running the agent. Change the state of the agent to running via

setStatus(). If no agent is selected, showAlert() will be called to pop up a dialog box to warn. Finally, call refreshAgentList() to update the list of agents on the main page.

- onStop(): The user selects an agent in the agent list and clicks the "Stop Agent" button to stop running the agent. Change the state of the agent to stopped via setStatus(). If no agent is selected, showAlert() will be called to pop up a dialog box to warn. Finally, call refreshAgentList() to update the list of agents on the main page.

- showAlert(): Use the JOptionPane.showMessageDialog() method in the Swing class library to pop up an information prompt box in the GUI, prompting the user to select an agent, and input the parameter view.getMainFrame(), which is the JFrame object (the main page of the application). The prompt dialog will be displayed in the center of the main page of the application.

- onRemove(): The user selects an agent in the agent list and clicks the "Remove Agent" button to remove the agent from the hash table. If no agent is selected, showAlert() will be called to pop up a dialog box to warn. Finally, call refreshAgentList() to update the list of agents on the main page.

- refreshAgentList(): used to refresh the agent list. In the method, first, obtain the agentControllers collection in the view object, then traverse each element in the collection, and insert it into the agentListView of the view object. This function first gets the table model tableModel and removes all the rows in it. Then, for each agent controller iter, its key-value pair will be added to a new row of the table model, use iter.getKey() to get the name of the agent, use iter.getValue().getStatus() to get the agent status. Finally, the content of the table model 'tableModel' is updated and the interface is refreshed to display the new data.

- onExit(): It is an ActionEvent event listener method. When the user clicks the "Exit" menu, this method will be called. Its function is to close the main window, that is, to call the mainFrame.dispose() method to release the window and all associated resources, thus ending the running of the application.

- onSendMsg(): Send message: The user selects an agent in the agent list and selects the "SendMsg" menu item to send a message to the selected agent. This function implements an event-handling method that is bound to an ActionListener of a JMenuItem that "SendMsg". This method is called when the user clicks on the "Send Message" menu item. The method first builds an array of strings as options in the

drop-down menu by getting the names of all agents. It then pops up a JOptionPane dialog asking the user to select a proxy name from the drop-down menu. Next, pops up another JOptionPane dialog box, allowing the user to enter the content of the message. Afterward, this method creates a Message object according to the user's selection and input, specifies the target agent for sending the message and the message content, and adds the Message object to the input message queue of the selected agent.

- onAddGoal(): When the method is triggered, an input box will pop up first to ask the user to enter a goal, and then the goal entered by the user will be parsed and added to the goals list of the currently selected agent. This method first uses JOptionPane.showInputDialog to pop up a dialog box, asking the user to input a goal, and save the string entered by the user in the input variable. Then, this method parses the input goal string by calling the parseGoal method of the ParseDriver class, and saves the parsed Goal object in the goal variable. Next, the method calls the view.getAgentListView().getAgentTable().getSelectedRow() method to obtain the row number of the currently selected agent and save it in the row variable. If no agent is currently selected, this method will call the showAlert() method to pop up a prompt box. Otherwise, this method will obtain the name of the currently selected agent by calling the view.getAgentListView().getAgentTable().getValueAt(row, 0) method, and save it in the agentName variable. Finally, this method adds the parsed goal object to the goals list of the currently selected agent by calling the agentControllers.get(agentName).getAgent().getGbase().getGoals().add(goal) method. In addition, addgoal() can only be triggered in the agent status window.

4.3.2 Agent Controller

The AgentController class is responsible for managing an Agent's deliberation cycle. This interpreter implements the interaction and operation of agents through the AgentController class. This class contains an Agent object, as well as attributes such as input and output message queues, status flags, etc. The input message of AgentController is received by the retrieveMessage() method, and the message is recorded in the log. Through the findPGRulesMatchedGoals() and findPGRulesMatchedBeliefs() methods, find the PRule rules that match Goal and Belief, and then apply the PRule rules through the applyPGRule() method to change the state of the Agent. When the user clicks "Start Agent" in GUI/View, the corresponding method in AgentController will be called. Here's how to implement these methods:

- AgentController():This is a constructor for a class named AgentController. The constructor initializes the instance variables and sets the initial state of the AgentController object. The constructor takes two arguments: an Agent object and a MainController object. Inside the constructor, a LinkedBlockingQueue object is created and assigned to the instance variable inputMessages. An empty ArrayList object is also created and assigned to the instance variable agentLog.This will be used to store log messages related to the Agent object. The instance variable 'status' is set to the value STOPPED, which is a constant indicating the current state of the agent.

- retrieveMessage(): This method retrieves a message from the 'inputMessages' queue and records it in the agent's belief base. First, the method retrieves the next message in the queue using the take() method of LinkedBlockingQueue. Then, the method adds a log entry to the agentLog list, indicating that a message was received. Next, the method creates an Argument object for the sender of the message (from), the message level (level), and the message content (content). It creates an Atom object with the name "received" and the arguments that were just created. Finally, the method creates a Belief object with the Atom object and adds it to the agent's belief base using the add() method of the belief base.

- findPGRulesMatchedGoals():This method searches for the plan goal rules in the agent's PG-rule set that match the agent's current goals in its goal base (Gbase). It creates a new list and iterates through the PRules in the agent's PG-rule set, and for each PRule, it checks if its atom is null.

- findPGRulesMatchedBeliefs(): This method searches for PGRules (Procedural Goal Rules) that match beliefs. It takes a list of PRules as input and returns a list of Map.Entry objects, where each entry contains a Hashtable of variable bindings and the corresponding PRule object that matches the beliefs.

- applyPGRule(): This method applies a production rule matched with a goal and a set of beliefs. The method first removes the goal that matched the rule from the agent's goal base, then substitutes the variables in the plan with the values found in the matched beliefs. Finally, the method adds the new plan to the agent's plan base.

- findPRRulesMatchedPlans(): The method iterates through all the RRules defined in the PRRule of the agent. If the RRule does not contain a Plan1 it is added to the result. Otherwise, it checks if any plan in the PBase matches with the Plan1

defined in the RRule. If there is a match, the RRule is added to the result. Finally, the method returns a list of RRules that match the plans in the PBase.

- findPRRulesMatchedBeliefs(): This method finds a PRrule from a list of PRrules that match the beliefs of the agent's belief base. First, creates a new JIPEngine instance and load the agent's belief base as a Prolog program. Then it iterates through each RRule in the input list and evaluates its associated query using the JIPEngine. If the query returns a non-null list of JIPTerm objects with at least one term, it means the rule matches the current beliefs, and the method returns the matching rule. If none of the rules match the current beliefs, the method returns null.

- applyPRRule(): This method adds the new plan to the agent's plan base, obtained from (rule.getPlan2()).

- executePlan(): This method takes a Plan object as a parameter and then iterates through each Action in the Plan, executing them one by one.

- run(): This is the main loop of the AgentController class, which implements the agent's deliberation cycle process and also calls the methods above. The loop keeps running until the agent is stopped. During each iteration of the loop, the agent performs a series of tasks, which include:

  - Waiting for incoming messages
  - Searching for goal planning rules that match the agent's goals
  - Selecting a goal-planning rule that also matches the agent's beliefs
  - Applying the selected goal-planning rule
  - Searching for plan revision rules that match the agent's current plans
  - Searching for plan revision rules that match the agent's plans
  - Selecting a plan revision rule that also matches the agent's beliefs
  - Applying the selected plan revision rule
  - Selecting a plan
  - Executing the plan

  If any of these tasks are completed, 'rulesAppliedOrPlanExecuted' is set to true, indicating that progress has been made during this iteration of the loop. Otherwise, the flag remains false. The agent's actions and decisions during each iteration of the loop are logged in the 'agentLog'. This log can be useful for debugging and understanding the agent's deliberation cycle.

4.3.3 Parse Driver

In the ParseDriver class, it parses the input string into an object using the ANTLR parser generated by the APL grammar. It consists of three methods:

- parseAgent(): In this method, it takes an ANTLRInputStream as input and returns an Agent object, which is built by visiting the nodes of the ANTLR parse tree generated by parsing the input stream with the AgentBuilder visitor.
- parseGoal(): In this method, it takes a string as input and returns a Goal object, which is built in the same way as the Agent object using the Goal rule of the APL grammar.
- parseAtom(): In this method, it takes a string as input and returns an Atom object, which is built in the same way as the Agent and Goal objects using the Atom rule of the APL grammar.

ANTLRInputStream is created from an input string, using a lexer, and parser to generate a parse tree. Finally, the AgentBuilder visitor is used to walk the parse tree and build the appropriate object. Any syntax errors encountered during parsing are handled by the GrammarErrorListener instance described below.

4.3.4 Agent Builder

Java class AgentBuilder, extended from APLGrammarBaseVisitor class. This class is a part of the parser, which parses the statement according to the provided grammatical rules and is used to construct the Agent object. The following are the main methods of this class.

- visitProgram(): Visit the root node of the parse tree and return the Agent object, which contains all the elements defined in the program.
- visitCapabilities(): Visit the Capabilities node defined in the program and return the CapabilityBase object.
- visitBeliefs(): Visit the Beliefs node defined in the program and return the BeliefBase object.
- visitGoals(): Visit the Goals node defined in the program and return the GoalBase object.

- visitPlans(): Visit the Plans node defined in the program and return the PlanBase object.

In this class, there are also many visit methods of other node types, such as visitLiteralWff(), visitOrWff(),visitWhileAction(), visitAtom(), etc. These methods access different nodes and return corresponding objects. For example, in visitPlan(), use the stream() method to convert the ctx.p_rule() node list into a stream, then use the map() method to convert each PRule object into a PlanGoalRule or PlanRevisionRule object, and use the collect() method Collect the results into a List.

4.3.5 Grammar Error Listener

This Java class implements the ANTLR4 BaseErrorListener interface to handle errors that occur during the parsing of a programming language.

The class provides the syntaxError() method of the BaseErrorListener interface. This method is called by ANTLR4 when a syntax error occurs during parsing. The method throws a ParseCancellationException with an error message that includes the line number, and character position.

4.4 View/Graphical User Interface

This interpreter's graphical user interface is implemented using JavaSwing. The GUI consists of two interfaces(Main View and Agent Status View). In this section, we will introduce how to implement the GUI of the 3APL interpreter

4.4.1 Main View

The first one introduced in this section is MainView, which is mainly used to create the GUI interface of the program. The properties of this class include (private final JFrame mainFrame) main window object, (private final JMenuItem exit) exit menu item object, (private final Font font) font object, and others. The MainView class is mainly composed of a constructor, MainView(), which initializes the main window object and other GUI components, including agent list views, agent control views, menu bars, and menu items. The main window is titled "3APL Interpreter" and closing it will exit the program. In this constructor, the menu bar and menu items are created and fonts are set. The menu

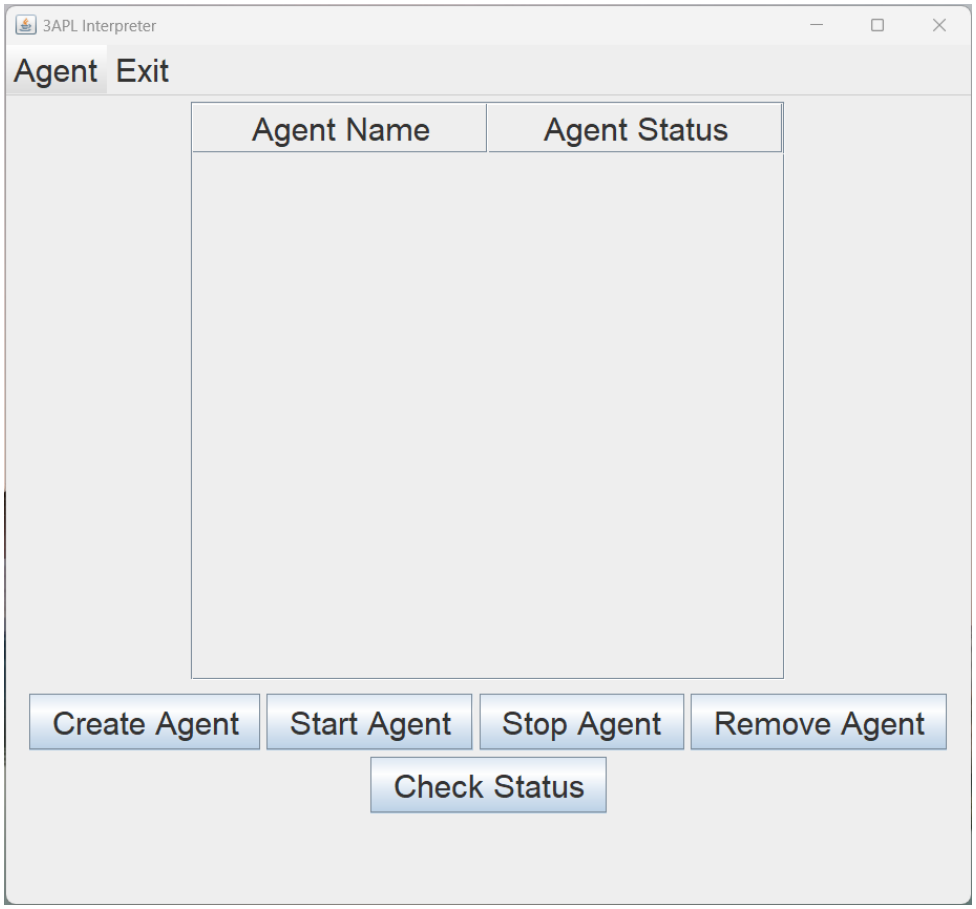

Figure 4.1: Main View

bar contains a menu called "Agent" with two menu items "Create" and "Send Msg". Among them, create is used to load the translated agent. In addition, there is an "Exit" menu item to exit the program. Finally, the constructor sets the main window's menu bar, showing the main window.

In addition, in this constructor, the MainView class calls the AgentControlView class and AgentListView class, agent list view, and agent control view, and then enriches the content of the GUI interface of the 3APL interpreter. These two parts will be described below

Agent List View

There is a list in the upper part of the software main page. Used to display a list of loaded agents and the status of each agent. In this project, the agent list is created through a class named AgentListView, which is a subclass of the JPanel type. The AgentListView class contains a constructor-AgentListView(), it will initialize the tableModel and agentTable objects, and add them to a JScrollPane in order to display scroll bars in the table. Then

add that scroll pane to the AgentListView panel.
In summary, the AgentListView class uses a JTable to present agent data. Moreover, the AgentListView class can easily add, delete and update data in the table by using the DefaultTableModel.

Agent Control View

There is a row of buttons on the bottom half of the software's main page. In this project, this part is implemented by a class named AgentControlView, which is also a subclass of type JPanel. The AgentControlView class creates createAgent, startAgent, stopAgent, removeAgent and checkStatus member variables of JButton type through the constructor (AgentControlView()), and creates multiple different buttons in the GUI to control the operation of different agents. The main purpose of the AgentControlView class is as a component in the GUI to perform user actions on the agent.

4.4.2 Agent Status View

After clicking the CheckStatus button, a new page will pop up, This part is implemented by a class called AgentStatusView, which inherits from the JFrame class. It shows specific information about an agent. In this interface, addGoal is set in the menu bar to add the agent's goal to realize the feature of 3APL's dynamic code change. The window contains various labels and text boxes for displaying the agent's name, belief, ability, goal, plan, Target rules, and revision rules. These labels and text boxes are placed on different panels using different layout managers.

Main member variables consist of the following:

- addGoal: JMenuItem type, used to add an "Add Goal" menu item in the menu bar.
- nameLabel: JLabel type, used to identify the name of the agent.
- agentName: JTextField type, used to display the name of the agent.
- beliefsLabel: JLabel type, used to identify the agent's beliefs.
- beliefs: JTextArea type, used to display the agent's beliefs.
- capabilitiesLabel: JLabel type, used to identify the capabilities of the agent.
- capabilities: JTextArea type, used to display the capabilities of the agent.

- goalsLabel: JLabel type, used to identify the goals of the agent.
- goals: JTextArea type, used to display the goals of the agent.
- revisionRulesLabel: JLabel type, used to identify the revision rules of the agent.
- revisionRules: JTextArea type, used to display the revision rules of the agent.
- deliberation: JTextArea type, used to display the inference process of the agent.

The AgentStatusView class is initialized through the constructor (AgentStatusView())

In the constructor, the size and layout of the interface will be set, a menu bar and various labels and text boxes will be added, and properties such as their fonts will be set.
This page can only be opened through the check Status of the main page, which can be opened normally before, during, and after stopping the operation. However, the operation of the agent can be analyzed by comparing the interface opened at different time points, but the information on the operation of the agent will not be stored in the interpreter. Once the window is closed when the agent is running, The agent specific before running will no longer be redisplayed.

4.5 Application

This part of the code is the entry point of the Java application and is mainly responsible for initializing and starting the 3APL interpreter.

In the above, we introduced how to implement the operation logic of the 3APL interpreter through the MVC architectural pattern. In order to refer to them, we need to define the package name of the Java class through the 'package cn.edu.nottingham.triaplinterpreter' and import the MainController class through "import cn.edu.nottingham.triaplinterpreter. controller.MainController" to use it in the Application class.

In the code of the Application class, a MainController object will be initialized, and the controller of the application will be started through this object. The entry point of the application is the main() function, which creates a new Application object and calls the start() function to start the application, which is 3APL interpreter

Chapter 5

Test Examples

In this section, two examples of the interpreter will be used to illustrate the function of the interpreter implementation and how to use it.

5.1 Interpreter initialization

The user needs to open the project using IntelliJ IDEA. Since jiprolog is used in this project, the SDK used in the project settings should be changed to JDK11. Then, set the language level to 9. Next, right-click on the "generated sources" directory and select "Mark as Source Root" to mark the directory as a source root. Finally, click on the "application.class" and run it to start the 3APL interpreter.

5.2 Communication of Multiple Agents

In this example, a scenario will be simulated where one agent cleans up the trash and another agent receives it. There are multiple rubbishes in the cleaner agent that is outside the rubbish bin and is trying to move them to the trash. It also provides their location. After the cleaner agent cleans up, it will send a message to the receiver agent, and the receiver agent will send a thank you message to the cleaner agent after receiving the information from the cleaner agent.
This example was chosen because it is common in reality, and both agents are fairly simple. It demonstrates the interaction between multiple agents involved in various types of MAS. The functionality of this interpreter is thus described. The MAS will operate as follows:

1. The cleaner agent moves to the location of the rubbish.
2. The cleaner agent grabs rubbish.

3. The cleaner agent moves the rubbish to the position of the bin.
4. The cleaner agent releases the rubbish to the bin.
5. Check whether rubbish is not in the bin, if so, jump to the first step.
6. The cleaner agent sends information to the receiver agent.
7. The receiver agent sends information to the cleaner agent.

As shown above, in this process, the cleaner agent cleans up the garbage and notifies the receiver agent, and the receiver agent passively receives the information and then sends the information to the cleaner agent. The specific code will be shown in the appendix.

5.2.1 Cleaner

The cleaner agent has the following beliefs before running: me(cleaner) you(receiver), positionAt(self,0,0), positionAt(rubbish,1,2), positionAt(rubbish,0,4) and positionAt(bin,5,6). This indicates the identity of oneself, the identity of the interacting agent, the position of the rubbish, the position of self, and the position of the bin.

```
BELIEFBASE {
    me(cleaner).
    you(receiver).

    positionAt(self,0,0).
    positionAt(rubbish,1,2).
    positionAt(rubbish,0,4).
    positionAt(bin,5,6).
    }
```

Figure 5.1: Cleaner's belief base

The cleaner agent has the following capabilities before running: Pick(), MoveTo(), and Throw(), which indicates the ability of the agent to select rubbish, move the position, and drop the rubbish.

```
CAPABILITIES {
    {positionAt(self,X,Y) AND positionAt(rubbish,X,Y) AND NOT occupied()}
    Pick(X,Y)
    {NOT positionAt(rubbish,X,Y), occupied(), cleaned(X, Y)},
    {positionAt(self,X0,Y0)}
    MoveTo(X1,Y1)
    {NOT positionAt(self,X0,Y0), positionAt(self,X1,Y1)},
    {positionAt(self,X,Y) AND positionAt(bin,X,Y) AND occupied()}
    Throw()
    {NOT occupied()}
}
```

Figure 5.2: Cleaner's capabilities base

The cleaner agent has two goals(clean(), and inform()) in the initial agent goal base to clean rubbish and inform another agent, and no plans in the plan base.

The PG-rule base of the cleaner agent includes PG rules related to inform() and clean(),

```
GOALBASE {
    clean(),
    inform()
    }

PLANBASE {
}
```

Figure 5.3: Cleaner's goal base

where clean() selects rubbish and bin, and updates the plan for subsequent movement to the bin to the plan base (when there is no rubbish bin location, move the rubbish to the bin location and place the rubbish). And inform() selects the location of the cleaned rubbish and the agent to send the message, and updates the plan to send the message to the plan base.

```
PG-RULES {
    clean() <- positionAt(rubbish, X, Y) AND positionAt(bin, Xb, Yb) |
    {
        WHILE positionAt(rubbish,X,Y) DO {
            MoveTo(X,Y);
            Pick(X,Y);
            MoveTo(Xb, Yb);
            Throw();
            }
    },
    inform() <- cleaned(X, Y) AND you(YOU) |
    {SEND(YOU, inform, clean(YOU) )}
    }
```

Figure 5.4: Cleaner's PG-rule base

The PR-rule base of the cleaner agent includes the PR rule of the plan to move the rubbish to the bin and select the matching plan and belief in the plan base and belief base to update the plan base

```
PR-RULES {
       WHILE positionAt(rubbish,X,Y) DO {
           MoveTo(X,Y);
           Pick(X,Y);
           MoveTo(Xb, Yb);
           Throw();
       }
   <- received(CUSTOM, inform, description_terminate) |
   {
        IF occupied() THEN
        {
            MoveTo(Xb,Yb);
            Throw();
        }
   }
```

Figure 5.5: Cleaner's PR-rule base

5.2.2 Receiver

The Receiver agent has the following beliefs before running: me(receiver), you(cleaner). This represents its own identity, the identity of the interactive agent, and there is only one goal of inform() in the goal base. The capabilities base and plan base of the Receiver agent is both empty.

```
CAPABILITIES {
}

BELIEFBASE {
me(receiver).
you(cleaner).
}

GOALBASE {
inform()
}

PLANBASE {
}
```

Figure 5.6: Receiver's capabilities base, goal base, belief base, and plan base

The Receiver agent has a PG rule that matches inform () in the PG rule base. After confirming the receipt of messages from other agents and not replying to them, apply this PGrule to update the plan base. And the PR rule base of the Receiver agent is empty.

```
PG-RULES {
inform() <- me(Me) AND received(You, inform, clean(Me)) AND NOT sent(You,inform,thanks(You)) |
                {SEND(You, inform, thanks(You) )}
}

PR-RULES{
}
```

Figure 5.7: Receiver's PG-rule base and PR-rule base

Chapter 6

Summary and Reflections

6.1 Project Management

The project starts in October 2022 and is developed on a weekly basis. The project management is shown in the Gantt chart below. When I first learned about the project, due to my lack of agent-oriented programming knowledge, it was difficult for me to understand the logic of the 3APL language itself and its syntax. At the beginning of this project, I tried to use JavaCC to parse 3APL, but I wasted a lot of time because 3APL's ENBF was not fully studied. Thus replacing the simpler Antrl4 for parsing. In addition, it took the most time to develop the compiler and agent class, and there were many mistakes in the syntax detection of the agent, which is also because 3APL's ENBF did not conduct a comprehensive study. Finally, due to the epidemic at the end of 2022 and more courses in the fall semester, the actual development time of the project was mainly in February and March, which also led to the failure of this project to be further improved. The issue of time management in interpreter development is also reflected in the Gantt chart below.

Figure 6.1: Gantt Chart

6.2 Reflection

This project also introduced me to the field of agent programming, a topic I had never heard of before getting this project. Due to the uniqueness and higher complexity of this project, helped me improve my ability to deal with coding better. ANTRL4 is used in this interpreter to parse the 3APL code, which made me understand the powerful plugin of ANTRL4. In addition, the interpreter is developed using the MVC software model, and the entire interpreter is divided into different modules during the implementation process. From back-end development to front-end development, the design and implementation of the interpreter will be easier. While developing the interpreter, an example of the 3APL multi-agent system was designed to test and illustrate the capabilities of the 3APL interpreter. In addition, failing to read the relevant materials of 3APL in detail led to the rush of the project schedule. This is a defect in my project management, and it also proves that my writing ability needs to be improved.

6.3 Conclusion

The project aims to develop an interpreter for the 3APL language using ANTLR4 and the MVC software pattern. The interpreter successfully supports all the features proposed for the 3APL language, and the example section demonstrates how to use and run the interpreter. Therefore, the project has successfully achieved its basic objectives. The interpreter is also quite comprehensive in many ways and can be used to build multi-agent systems while running multiple agents simultaneously. The interpreter can also display the agents' operation logs, which is convenient for users to test and develop 3APL agents.

6.4 Future Work

However, there are still many aspects of this project that can be improved and perfected. First of all, the function of the interpreter for user to send messages to the agent is not yet fully realized, and the SendMsg() function can be improved, so that the 3APL agent program can be developed more easily. Secondly, the parser of the interpreter can be refactored to check whether the agent conforms to the syntax when compiling the 3APL agent. This enables better error and exception generation procedures to help programmers debug code. Finally, the ability for an agent to make changes to the environment is not yet well implemented and could improve the visualization of the environment and thus the user experience with the 3APL interpreter.

References

[1] Topics in ai : Agents. https://www.doc.ic.ac.uk/project/examples/2005/163/g0516334/#contentalt.

[2] The ebnf specification of the 3apl language for individual agents. http://www.cs.uu.nl/3apl/bnf.pdf.

[3] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[4] Mehdi Dastani, Frank Dignum, and John-Jules Meyer. Autonomy and agent deliberation. In *International Workshop on Computational Autonomy*, pages 114–127. Springer, 2003.

[5] Junyan Hu, Parijat Bhowmick, Inmo Jang, Farshad Arvin, and Alexander Lanzon. A decentralized cluster formation containment framework for multirobot systems. *IEEE Transactions on Robotics*, 37(6):1936–1955, 2021.

[6] M. van Otterlo, M. Wiering, M. Dastani, and J.-J. Meyer. A characterization of sapient agents. In *IEMC '03 Proceedings. Managing Technologically Driven Organizations: The Human Side of Innovation and Change (IEEE Cat. No.03CH37502)*, pages 172–177, 2003.

[7] Koen V Hindriks, Frank S De Boer, Wiebe Van der Hoek, and John-Jules Ch Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2:357–401, 1999.

[8] Jan Broersen, Mehdi Dastani, Joris Hulstijn, and Leendert van der Torre. Goal generation in the boid architecture. *Cognitive Science Quarterly*, 2(3-4):428–447, 2002.

[9] Mehdi Dastani, Frank de Boer, Frank Dignum, and John-Jules Meyer. Programming agent deliberation: An approach illustrated using the 3apl language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, page 97–104, New York, NY, USA, 2003. Association for Computing Machinery.

[10] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dmars. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 155–176. Springer, 1997.

[11] Anand S Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *European workshop on modelling autonomous agents in a multi-agent world*, pages 42–55. Springer, 1996.

[12] Anand S Rao and Michael P Georgeff. Modeling rational agents within a bdi-architecture. *KR*, 91:473–484, 1991.

[13] Anand S Rao, Michael P Georgeff, et al. Bdi agents: from theory to practice. In *Icmas*, volume 95, pages 312–319, 1995.

[14] Mehdi Dastani, M Birna van Riemsdijk, Frank Dignum, and John-Jules Ch Meyer. A programming language for cognitive agents goal directed 3apl. In *Programming Multi-Agent Systems: First International Workshop, PROMAS 2003, Melbourne, Australia, July 15, 2003, Selected Revised and Invited papers 1*, pages 111–130. Springer, 2004.

[15] Michael Bratman. Intention, plans, and practical reason. 1987.

[16] Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 489–496, 2003.

[17] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules C. Meyer. *Semantics of Communicating Agents Based on Deduction and Abduction*, pages 63–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[18] Mehdi Dastani, Mv Riemsdijk, Frank Dignum, and John-Jules Ch Meyer. A programming language for cognitive agents goal directed 3apl. In *International Workshop on Programming Multi-Agent Systems*, pages 111–130. Springer, 2003.

[19] Robert Eckstein, Marc Loy, and Dave Wood. *Java swing*. O'Reilly & Associates, Inc., 1998.

[20] Mehdi Dastani. 3apl platform. *Utrecht University*, 2004.

[21] Bradley Steinbacher Tim McCormack and Terence Parr. Antrl4 grammar structure. `https://github.com/antlr/antlr4/blob/master/doc/grammars.md`, 2022.

Appendix A

EBNF of 3APL

⟨*Program*⟩ ::= "Program" ⟨*ident*⟩
("Load" ⟨*ident*⟩)?
"Capabilities :" (⟨*capabilities*⟩)?
"BeliefBase :" (⟨*beliefs*⟩)?
"GoalBase :" (⟨*goals*⟩)?
"PlanBase :" (⟨*plans*⟩)?
"PG − rules :" (⟨*p_rules*⟩)?
"PR − rules :" (⟨*r_rules*⟩)?

⟨*capabilities*⟩ ::= ⟨*capability*⟩ ("," ⟨*capability*⟩)*
⟨*capability*⟩ ::= "{" ⟨*query*⟩ "}" ⟨*Atom*⟩ "{" ⟨*literals*⟩ "}"
⟨*beliefs*⟩ ::= (⟨*belief*⟩)*
⟨*belief*⟩ ::= ⟨*ground_atom*⟩ "." | ⟨*atom*⟩ ": −" ⟨*literals*⟩"."
⟨*goals*⟩ ::= ⟨*goal*⟩ ("," ⟨*goal*⟩)*
⟨*goal*⟩ ::= ⟨*ground_atom*⟩ ("and" ⟨*ground_atom*⟩)*
⟨*plans*⟩ ::= ⟨*plan*⟩ ("," ⟨*plan*⟩)*
⟨*plan*⟩ ::= ⟨*basicaction*⟩ | ⟨*composedplan*⟩
⟨*basicaction*⟩ ::= "ϵ" | ⟨*Atom*⟩ | "Send("⟨*iv*⟩,⟨*iv*⟩,⟨*atom*⟩")" |
"Java("⟨*ident*⟩,⟨*atom*⟩,⟨*var*⟩")" | ⟨*wff*⟩"?" | ⟨*atom*⟩
⟨*composedplan*⟩ ::= "if" ⟨*wff*⟩ "then" ⟨*plan*⟩ ("else" ⟨*plan*⟩)? |
"while" ⟨*query*⟩ "do" ⟨*plan*⟩ |
⟨*plan*⟩ ";" ⟨*plan*⟩
⟨*p_rules*⟩ ::= ⟨*p_rule*⟩ ("," ⟨*p_rule*⟩)*
⟨*p_rule*⟩ ::= ⟨*atom*⟩ "<−" ⟨*query*⟩ "|" ⟨*plan*⟩
⟨*p_rule*⟩ ::= "<−" ⟨*query*⟩ "|" ⟨*plan*⟩
⟨*r_rules*⟩ ::= ⟨*r_rule*⟩ ("," ⟨*r_rule*⟩)*
⟨*r_rule*⟩ ::= ⟨*plan*⟩ "<−" ⟨*query*⟩ "|" ⟨*plan*⟩
⟨*literals*⟩ ::= ⟨*literal*⟩ ("," ⟨*literal*⟩)*
⟨*literal*⟩ ::= ⟨*atom*⟩ | "not("⟨*atom*⟩")"
⟨*wff*⟩ ::= ⟨*literal*⟩ | ⟨*wff*⟩ "and" ⟨*wff*⟩ | ⟨*wff*⟩ "or" ⟨*wff*⟩
⟨*query*⟩ ::= ⟨*wff*⟩ | "true"
⟨*iv*⟩ ::= ⟨*ident*⟩ | ⟨*var*⟩

Figure A.1: EBNF of 3APL, adopted from[2]

Appendix B

Figures

Figure B.1: Class diagram

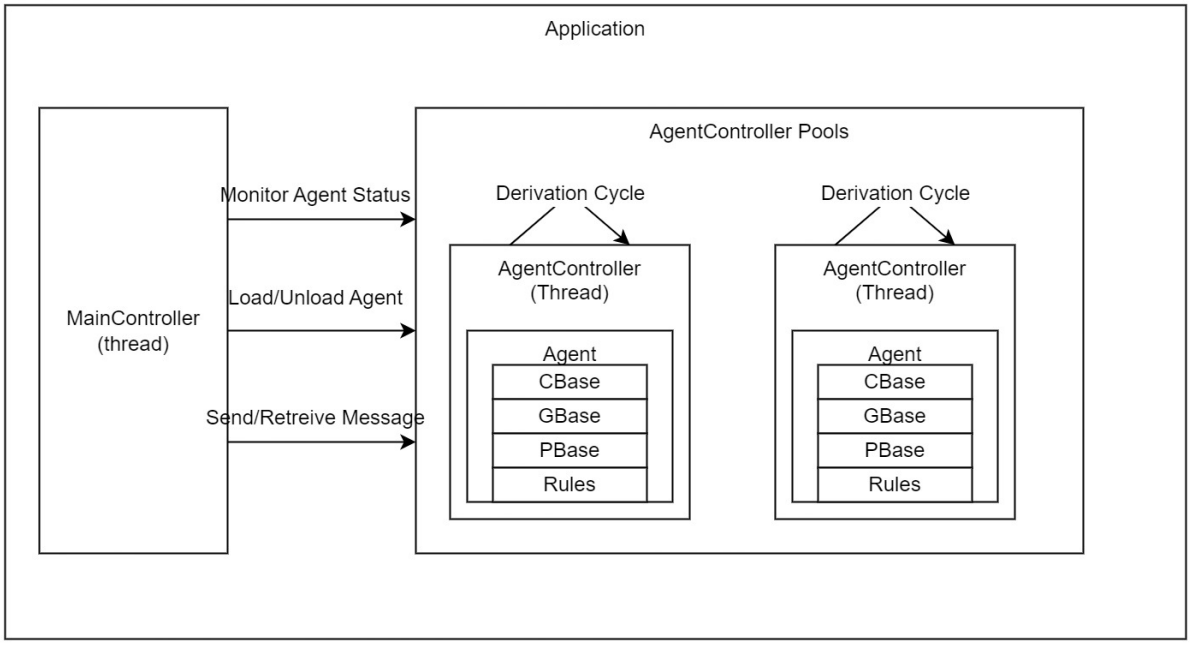

Figure B.2: Backend design

Appendix C

3APL Example Codes

C.1 cleaner.3apl

```
1
2  PROGRAM "cleaner"
3
4  CAPABILITIES {
5      {positionAt(self,X,Y) AND positionAt(rubbish,X,Y) AND NOT occupied()
       }
6      Pick(X,Y)
7      {NOT positionAt(rubbish,X,Y), occupied(), cleaned(X, Y)},
8      {positionAt(self,X0,Y0)}
9      MoveTo(X1,Y1)
10     {NOT positionAt(self,X0,Y0), positionAt(self,X1,Y1)},
11     {positionAt(self,X,Y) AND positionAt(bin,X,Y) AND occupied()}
12     Throw()
13     {NOT occupied()}
14 }
15
16 BELIEFBASE {
17     me(cleaner).
18     you(receiver).
19
20     positionAt(self,0,0).
21     positionAt(rubbish,1,2).
22     positionAt(bin,5,6).
23     }
24
25 GOALBASE {
26     clean(),
27     inform()
28     }
29
30 PLANBASE {
```

```
}

PG-RULES {
    clean() <- positionAt(rubbish, X, Y) AND positionAt(bin, Xb, Yb) |
    {
        WHILE positionAt(rubbish,X,Y) DO {
            MoveTo(X,Y);
            Pick(X,Y);
            MoveTo(Xb, Yb);
            Throw();
            }
    },
    inform() <- cleaned(X, Y) AND you(YOU) |
    {SEND(YOU, inform, clean(YOU) )}
    }

PR-RULES {
       WHILE positionAt(rubbish,X,Y) DO {
           MoveTo(X,Y);
           Pick(X,Y);
           MoveTo(Xb, Yb);
           Throw();
       }
   <- received(CUSTOM, inform, description_terminate) |
   {
        IF occupied() THEN
        {
            MoveTo(Xb,Yb);
            Throw();
        }
   }
   }
```

C.2 receiver.3apl

```
1  PROGRAM "receiver"
2
3  CAPABILITIES {
4  }
5
6  BELIEFBASE {
7  me(receiver).
8  you(cleaner).
9  }
10
11 GOALBASE {
12 inform()
13 }
14
15 PLANBASE {
16 }
17
18 PG-RULES {
19 inform() <- me(Me) AND received(You, inform, clean(Me)) AND NOT sent(You
       ,inform,thanks(You)) |
20                 {SEND(You, inform, thanks(You) )}
21 }
22
23 PR-RULES{
24 }
```