University of Nottingham
UK | CHINA | MALAYSIA

UNIVERSITY OF NOTTINGHAM, SCHOOL OF COMPUTER SCIENCE

A 3APL Interpreter

Dissertation

Submitted in partial fulfillment of
the conditions for the award of the degree **BSc Computer Science.**

AUTHOR

Student ID: 4336486/16522115

SUPERVISED BY

Brian Logan

pszbsl@nottingham.ac.uk

I hereby declare that this dissertation is all my own work, except as indicated in the text.

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive.

April 23, 2021

Abstract

Agent-oriented programming is a programming paradigm that generally treats software as agents and focuses on developing the interaction and deliberation procedure of these agents. It has been used in different academic areas and industries such as robotics and manufacturing. This project aims to implement a new interpreter for an agent-oriented programming language called 3APL. Such an interpreter is able to compile codes and construct multi-agent systems easily while providing functionality to put the agents in a virtual environment. In addition, a way to monitor the communication and deliberation of agents in runtime is also implemented. It can be seen that this interpreter can facilitate the development of agent based applications significantly.

Acknowledgements

I would like to express my sincerest thanks to my supervisor, Dr Brian Logan, for not only his guidance and advice but also his patience and encouragement throughout the entire year. I would like to thank all my friends for their inspiration and my parents for their unwavering support. I would also like to extend my thanks to everyone who have encouraged and helped me along this road.

Contents

1 Introduction

According to Wooldridge and Jenning's [1][2] definition, an intelligent agent is "a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives". By this definition, agents are able to run in a given environment autonomously in order to achieve their goals while sensing and reacting to changes made in the environment without outside guidance or intervention. The concept of intelligent agent is used to abstract various entities which can be either software or hardware. For example, an autonomous car can be viewed as an agent whose objectives is to approach its destination safely and a mail server can be viewed as an agent whose objective is to transfer emails properly.

In real world scenarios, the problems to be solved are often large scale or even distributed. Therefore, it becomes hard to use the abstraction of single agent to capture the whole pattern of such complex problems. However, the decomposition performed by multi-agent systems "allow each agent to use the most appropriate paradigm for solving its particular problem" in a complex, large or unpredictable problem domain according to Sycara[6]. In a multi-agent system, multiple agents can exist in the same environment, and given the ability to interact with each other. For example, Autonomous cars can interact in order to avoid collisions and perform better routes planning.

The concept of agent and multi-agent system has been viewed as a compact and efficient approach of describing and therefore solving problems. Moreover, as Shoham, Dennett, McCarthy [3][4][5] illustrated, *anything* can be described by the pattern of agent (cognitive agent) "whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments". These two facts have motivated the proposition of a new programming paradigm called agent-oriented programming. As Shoham [3] described, a complete AOP (agent-oriented programming) framework should consists a restricted formal language to describe mental state of agents, an interpreted programming language to define and program agents and an "agentifier" to convert neutral devices into programmable agents.

Many of the agent-oriented programming languages are proposed a decade ago such as 3APL(1999)[7], GOAL(2000)[8], 2APL(2008)[9], AgentSpeak(L) and JASON(1996,2007)[10][11]. However, they weren't widely used for practical purposes until recently due to the fact that the development of techniques in fields such as computer vision, sensor and low-level control system have improved the ability of robots and other physical devices to function accurately enough on simple tasks. Agent-oriented systems can usually be used as the high-level decision-making system of these devices. Due to this development, agent-oriented programming is performing greater impact on the study of Robotics and Artificial Intelligence.

3APL (3APL) is short for An **Abstract Agent Programming Language** which is an agent-oriented programming language proposed in 1999 [7]. It is considered to be an implementation of a common approach for designing intelligent agents called Belief-Desire-Intention (BDI) architecture. The representation of a BDI-described agent can be separated in three parts: knowledge that the agent has about the world, the objectives that agent tries to achieve and the selected objective that is currently being achieved (belief, desire and intention respectively). Besides, the strategy that 3APL agents applied to commit to intentions are considered to be blind commitment which means that once an objective (intention) is selected to be achieved, it is maintained as long as it is not achieved [12]. As previously mentioned, there are multiple agent-oriented programming languages existing, however, 3APL provides a unique feature that allows agents to modify the compiled code (plans, more specifically) in runtime which means that even though blind commitment is applied to agents, they are still reactive to the environment after intentions are selected.

In this project, a new 3APL interpreter is implemented with a graphical user interface, this interpreter can be used to compile 3APL code into agents and construct multi-agent systems. The agents in multi-agent systems will be able to communicate with each other under a set of protocols. In addition, a set of standards for implementing environments of multi-agent systems is established.

2 3APL language

In this project, the 3APL language being used is the latest version (2003) [13] with some minor tweaks on the syntax to facilitate the implementation. However, these tweaks does not influence any actual functionality so that they will only be highlighted in the appendix regarding detailed syntax of 3APL.

2.1 Agent

A typical 3APL agent has six mental components which are capability base, belief base(belief in BDI), goal base (desire in BDI), plan base (intention in BDI), plan revision rule base and goal planning rule base [13]. In detail:

- Goals are the objectives that agent tends to achieve. They can be pre-programed in code or added in runtime by receiving messages.
- Beliefs represent knowledge that agent has. They can be pre-programed in code and updated during runtime.
- Capabilities are the collection of basic actions that agent can perform. Each actions can change agent's beliefs and achieve goals.
- Plans are serials of capabilities and other special actions that agent will perform which could result in goals achieved.
- Goal planning rules are used by the agent to determine which plans to execute according to its goals and beliefs.
- Plan revision rules are reactive rules which are used to adjust the plans according to current beliefs.

Each cognitive component is coded by a language which uses part of Prolog logic programming language. The Prolog features are used to represent the knowledge that agent has and search these knowledge when needed. A deliberation cycle which specifies how agent use their cognition to fulfil goals is used by each agent to act accordingly.

More details about the syntax of agent can be found in Appendix A

2.2 Deliberation Cycle

In this section, the deliberation cycle will be introduced with an example to help the understanding of the concept.

1. Initial State.
2. Check messages.
3. Select and perform plan revision rule.
4. Perform one step of each plan.
5. Select and perform goal planning rule.
6. Jump back to step 2 and repeat.

After initializing, the first step will be taken in the deliberation cycle is to receive messages from other agents and environment to update its beliefs which is also the only way for and agent to get outside information. It will then try to see if any revision can be done to its plans before actually executing any plans. The revision procedure has the highest priority to maintain the reactive characteristics of agents. After the actual execution of plans, the agent will use goal planning rule to see if there is any other plans can need to be committed and end one cycle of deliberation.

An example of agent will be given briefly to help the understanding of this language. This example will be extended in detail in the example section.

The scenario being used here is a garbage collecting task. An agent is used to control a cleaner which is able to command the cleaner and retrieve information from it through a special kind of messages. The cleaner is used to collect garbage from garbage bins, carry them and throw them into garbage stations. In addition, each step the cleaner moves will cost it one unit of power and it will need to recharge itself at recharge station regularly. The two limits on the cleaner that need to be considered are: limited capacity of garbage and limited size of battery.

The agent has one goal which is to clean up all the garbage bins. It also has the belief which describes the surrounding environment and its own status (battery and garbages being carried). There will be only one initial plan which is to connect itself to cleaner. There can be two goal planning rules which represent the two mode the agent can be in. One mode is to gather garbages and throw them, this rule will be performed when the surrounding environment has garbages uncollected or the agent has already carried some garbages. This rule will generate a plan which is to go to the location with highest score (determined by the amount of garbages, battery and the distance between cleaner and that location) and perform a certain action to either collect or throw. The other action is to explore the environment, it can only be performed when all the calculated scores are under a threshold. The agent will randomly move towards a direction to explore the environment for a short period of time in this mode. There are also mechanisms to ensure that the agent can only be in one mode at a time. Finally, there is a plan revision rule which is a special emergency mode which will be used when the battery is running too low to reach a critical state. Due to the fact that revision rules are reactive with highest priority, it will overwrite all other modes and command the cleaner to move to the nearest recharge station immediately.

To summarize, the specific deliberation cycle of this cleaner controlling agent is described as:

1. Initial State: retrieve information about itself and about the environment from cleaner.
2. Check messages: check if one action has already been completed by the cleaner and therefore another one can be taken.
3. Select and perform plan revision rule: check if the battery is running critically and therefore emergency mode should be used.
4. Perform one step of each plan: perform the current mode for one step if the previous step in that mode is completed by the cleaner.
5. Select and perform goal planning rule: if the agent just completed one mode and has not yet started any modes, choose one mode to commit to.
6. Jump back to step 2 and repeat.

More details on the transition rules and semantics of 3APL such as plan matching and guard satisfying can be found in Appendix B

3 Related Work

3APL was first proposed in 1999, in its first version, agent consists of only beliefs and intentions (plans) [7]. In this version, beliefs are used to represent the knowledge that agents hold and intentions are the reasoning procedures that will be taken by the agent. Compare to the later version, intentions can be considered as combination of plans and rules while goals are represented by beliefs that the agent does not hold.

In 2003, the language was extended by adding goals, planning rules, plan revision rules, goal revision rules as agents' components [13]. The beliefs of the agent are also modified to be similar to a logic language which is constructed by predicates and Horn clauses. This is the latest version of this language and is also the version being used in this project.

The latest and most comprehensive implementation of 3APL interpreter is called 3APL Platform. It was developed in 2003 and maintained by a team at the computer science department of Utrecht University [14]. The latest update on this interpreter was performed on 19 November, 2007. This interpreter has a graphical user interface which contains useful features such as sniffer to monitor runtime status, syntax highlighting editor and dashboard to control the execution of programs. This implementation also included multi-agent environment and mechanism for communication between agents. The agents' components supported in 3APL Platform are the same as the specification in this project.

Other implementations include a prototype of 3APL interpreter in Haskell, and student thesis-based projects [15].

4 Description of the Project

Due to fact that agent-oriented programming is a highly dynamic procedure meaning that agents can be deleted from or added to a system in runtime while containers and environment can also be modified, it would be difficult to develop a compiler for it. In addition, as mentioned before, the existing 3APL interpreter has not been updated since 2007 and works only with Java 1.6 which is out-of-date. This project aims to develop a new 3APL interpreter on newer version of Java with additional features and new approaches. The interpreter being developed differs from the old one mainly in two aspects: a new approach of formalizing communication between agents and environment, a new approach for programming environment. However, due to the limitations of time, part of features of the old 3APL platform such as syntax highlighting editor are not supported in the interpreter developed in this project.

Some specifications of this project are given below:

1. A compiler for the 3APL language will be developed, it should support all the features of the language.
2. A new approach of constructing multi-agent system will be introduced which includes server, containers and agents.
3. The containers of agents will be implemented as part of the multi-agent system. They're used to contain agents and handle the communication between agents and help the agents to interact with environments.
4. A mechanism will be implemented for communications between agents.
5. A mechanism will be implemented for interactions between environment and agents.
6. A graphical user interface will be implemented. Users will be able to stop any container, edit its agents, recompile and resume the previous procedure. Moreover, the user will be provided visualized information of how containers and agents work and the condition of current environment.

In addition, for illustration and evaluation purpose, an example of multi-agent system will be designed and developed for both evaluation and illustration. The example should contain multiple agents and a non-trivial environment in order to illustrate most of the platform's features.

5 Design

In this section, the design concerns will be introduced. Generally, design and implementation of the interpreter can be separated into three parts, the multi-agent system which is the product of interpreter, the interpreter itself and user interface of the software. In order to fully understand the design of the interpreter, it would be beneficial to introduce the design of multi-agent systems first so that some concerns become clearer by knowing the requirements made by the product.

However, due to its complexity and the fact that great deal of effort was investigated to design and implement it, agents will be introduced first in a separate sub-section following the rest parts of multi-agent system.

5.1 Agent

As discussed in 3APL language section, agents are the most basic and fundamental components of a multi-agent system. The structure of agents has already been introduced in the language section, while the detailed syntax of agents is introduced in Appendix A. The design of the agent follows the same structure which means that each component of the implementation is corresponding to a component of the agent in the language. The structure of implementation that construct those components is also corresponding to the syntax structure of those components.

Some examples of this corresponding relationship will be shown below so that the idea can be illustrated clearly.

- Prolog clause: implemented by *PredClause* class
- While loop: implemented by *WhileAction* class
- Goal: implemented by *Goal* class
- Goal planning rule base: implemented by *GoalPlanningRuleBase* class

In addition to these necessary components that have already been introduced in the 3APL language section, there are also some other features that need to be implemented by agents. The first one is an identifier that can be used by containers and server to identify the agent. A second feature is that agents should be able to send messages to other agents, they should also send request to environment and receive response from environment.

After introducing how an agent can be constructed and the functionality it provides, it is also important to discuss about the Prolog reasoning procedure in agents which is the core of execution procedure in an agent. The Prolog reasoning procedure in this project is provided by an open source library call tuProlog. tuProlog provides a set of Java interface that will be used in this project. The core functionality of tuProlog is provided by a Prolog engine class which can manage theories (facts) and respond to queries. A detailed explanation about how tuProlog works and the usage of it will be provided in implementation section.

5.2 Multi-agent System

Each multi-agent system in this software is hosted by one and only one server. The server contains multiple containers which are considered as clusters of agents. In addition, a server can connect to an environment which contains controllable entities that are controlled by the agents. To simplify this model, we separate the server and environment as two part of a multi-agent system which are connected by environment interface.

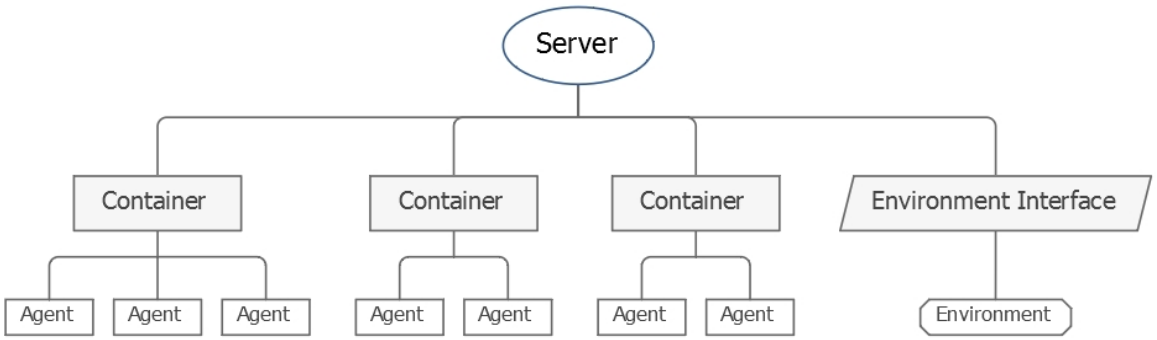

Figure 1: Multi-agent System in 3APL

5.2.1 Server and Container

Servers are used to control the whole system. A server is considered to be the only component of an multi-agent system that users can directly interact with. The server receives all messages from the containers and forward them to destination containers. The server is also used as intermediary for agents and environment, it transfers action request from agent to environment and response from environment to agent.

Each container is considered to be a clusters of agents, agents belonging to the same container are considered to be closely related. Containers are also used to contain information about functionality of agents, meaning that agent can offer a certain service and notify its container which is responsible for providing information of that service when other agents need the service. Since we have made the assumption that agents in the same container are related, when an agent tries to find a service, the container will first check if other agents in it offer such service before send the request to other containers. Further explanation about message handling and service management will be done in the design section of agent messages. In addition to representing the relationship between agents, the other reason of introducing containers is to prevent server from holding information of all agents and therefore slow down the procedure of message handling. Such design can also facilitate this procedure by running each containers and servers on different threads. In addition, each containers and servers are assigned a unique identifier.

5.2.2 Agent Messages

The messages within the system are meant to carry out all the communication between agents and containers meaning that the messages are not only sent and received by agents, they can also be sent and received directly by the AMS (agent management system). The agent management system is simply the set of all the containers. All messages sent or received directly by containers are considered to be sent or received by AMS. All the messages between the agents are sent by their send actions while executing their plans. For each send action, the receiver, performative, content and additional information will be stated. Variables are allowed in all of these components which normally are expected to be grounded by the environment. However, there are no specific rule for receiving messages, messages will be added to the belief base of an agent when received and should be handled specifically by the programmer of agents.

To discuss about the type of messages can be sent, we first introduce the different performative and how they influence the procedure. There are four different performative. The expected behaviors of them are specified as following:

- Query: Ask for service or information from other agents.
- Inform: Inform other agents the service it provides or believes it holds.
- SendGoal: Ask other agents to commit to a specific goal.

- RequestGoal: Request a goal from other agents to commit to.

There are no strict constraints on the behaviors of a performative except several special cases involving querying and informing services which influence the procedure of handling messages.

Type of messages

Even though there are four different performative, the types of messages are not only determined by the performative but also sender and receiver of the message.

- General message: all the messages sent from agent to agent regardless of the performative, they are the most common message type.
- Inform about service: the messages sent from agent to AMS using a $Inform$ performative, these messages will log a service in the container of the sender agent for further enquiry.
- Query about service: the messages sent from agent to AMS using a $Query$ performative, the container will react properly regarding if it holds information about this service.
- Reply for query about service: the messages sent from AMS to agent who queried about a service using a $Inform$ performative. The container which has information about the service will send this reply to the querying agent on behalf of the service providing agent.

General Procedure of Transferring Messages
As we have described above, agents are contained by containers and containers are contained by servers. This hierarchy also applies for transferring messages which means that messages from an agent can only be directly transferred to its container and its container will determine how to handle that message next. Similarly, messages from an agent can only be directly transferred to the server. During this procedure, content of the message normally remain unchanged which means that this procedure is highly dependent on the decision making of each of the components (agents, containers and server) in the multi-agent system. Such decision making procedure and the specific flow of messages will be explained in detail in the implementation section later.

Service Management
Service Management functionality is also done by Prolog engines, they consider provided services as facts and query of services as Prolog queries. Each container has the highest priority regarding providing services for its own agents while the information about what services are provided by an agent is only held in the container of that agent. Meaning that agents in the same container will first try to use the services provided by the each other. When a query about service is made by an agent, the container of query sending agent will first check if the service is available in it. If the container of sender agent failed to provide the service, it will forward the query to server and let it spread it to all other containers. Other containers will receive the query with equal priority meaning that each of them will decided if they provide that service themselves and send a response to the agent regardless of what decision is made by other containers. As a result, in this case, multiple responses may be sent to the query sending agent and it needs to handle all those responses.

5.2.3 Environment

On the environment side of the multi-agent system, the Environment Interface Standard (EIS) approach is applied, for which an environment is used to represent the world where agents act in and it should have no knowledge of how agents and the whole multi-agent system are implemented [12]. Environments can be self-updating and can contain multiple controllable entities. Controllable entities are the virtual or physical objects that agents control and therefore, all impacts caused by the agents to the environment should be done by these entities. Each entity can only be controlled by one agent and an agent can only control no more than one entity. In this way, the environment and other parts of the multi-agent system are separated clearly, environment should not contain any information about agents, and server should not contain any information about controllable entities. Therefore, the additional component called environment interface is introduced which is used to map between agents and controllable entities. The environments are made so that they commit to the same APIs, and therefore it is easier for programmers to program.

Environment Actions and Environment Responses
As introduced before, there is a special type of action in agents' plan called environment action. These actions are used to send action request to the environment instead of performing the action themselves. In order to send action requests, the agent must be connected to an entity. The environment of an system can reject or accept actions, but a response will be sent back to the agent in both cases. In addition, responses can also be sent to agents who didn't make any requests.

The behavior of agents (controlled entities) can be classified as three types according to the environment actions and responses:

- Passive Sensing: representing the scenario where information of the environment are passively fed to the agent, meaning that the agent doesn't send any action requests but responses from environment are produced frequently.
- Active Sensing: representing the scenario where information of the environment are fed to the agent when it is asked. The responses are given by the environment according the request made by the agent. However, the action requests from the agent does not influence the behavior of the entities controlled by it.
- Action taking: representing the scenario where agent asks actions to be taken by the controlled entity.

Transferring Environment Action Requests and Environment Responses
Similar to messages between agents, the request is sent indirectly to environment interface through container and server. The environment interface is responsible for mapping the agent's identifier to entity's identifier and forwarding the request to environment with the corresponding entity's identifier.

The response are also generated with entity's identifier and such identifier will be used by the environment interface to map to agent's identifier. With the agent's identifier retrieved, the procedure is simply to the procedure of sending messages from server to a specific agent.

5.3 Interpreter

The interpreter is considered as the logical part of this software. It handles input from graphical user interface and processes them.

5.3.1 Interpreter Structure

For the interpreter, Factory pattern is used in this interpreter in order to facilitate the construction of each components of the multi-agent system. There are only three possible external inputs for the interpreter which are the 3APL code, environment class and parameters for creating controllable entities. The 3APL code will be given to the compiler and Java agent object will be generated and returned to the runtime manager. Similarly, container factory and server factory will be used to create containers and servers and all containers and servers are also stored in the runtime manager. After an environment is loaded in a server, controllable entities can be created by the environment using function provided by that environment. However, the functionality of creating controllable entities can also be used to modify settings of the environment itself.

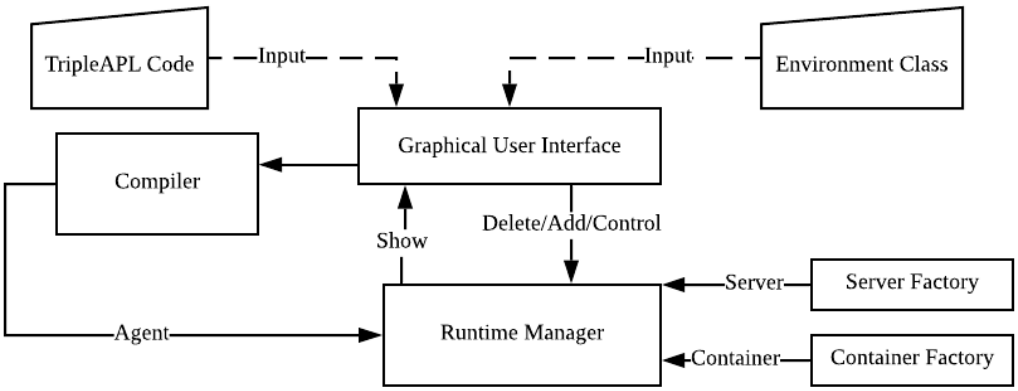

Figure 2: Interpreter Structure

5.3.2 Server factory and Container factory

Due to the fact that there are very limited parameters for setting a server or container and the graphical user interface requires them to be constructed without contains anything, the only benefits of using factories are that they can generate unique identifier automatically.

5.3.3 Compiler

The compiler are developed using JavaCC which is short for Java Compiler Compiler. Due to the nature of this tool being used, the compiler is designed so that each node in the abstract syntax tree is converted into an object of a specific Java class which means that the parser, checker and code generator are hard to be distinguished. The structure of the abstract syntax tree is also reasonably similar to the one being described in language section with only few tweaks to avoid conflicts such as left-recursion. More detailed explanation on the development of compiler will be explained in the implementation section.

5.4 Graphical User Interface

The graphical user interface of this interpreter is designed as three parts:

- Main Page: the main page of interface, containing all the operations need to be taken including creating and linking servers/containers/agents/controllable entities/environments. It also allows user to control the servers and open other pages.
- Status Monitor: this page will display information about the running of selected server, it includes three sections: messages in the multi-agent system, action requests and responses of environment and customable information in the environment.
- Graphical Environment: for each environment, a graphical representation of the environment can be programmed and it will be shown when the server of that environment is started.

The usage of this interface will be shown in the evaluation section and details about key points of implementation of each parts will be explained in the implementation section.

6 Implementation

6.1 Agent

The implementation of agents will be described in this section. To make the structure clearer, each component of an agent is defined to be a class, which are *GoalBase, BeliefBase, CapabilityBase, PlanBase, GoalPlanningRuleBase, PlanRevisionRuleBase*. The agent also contains a Prolog engine which will be used to represent the runtime beliefs of the agent so that instead of re-implementing techniques such as

backtracking, that Prolog engine is used to reason logic clauses. In this project, the Prolog engine used is called tuProlog. As shown before, each terminal and non-terminal in the abstract syntax tree of the compiler is compiled into a Java object and these objects are used to construct those components of an agent in the same way as their corresponding terminals and non-terminals construct the abstract syntax tree. In such a way, the hierarchy of each class is clearly defined and unnecessary details are hidden from higher levels of the abstraction.

In this section, the architecture of how an agent is constructed will be first described in a graph. The graph contains the information about how each level of abstraction is made and which classes are implemented for that abstraction.

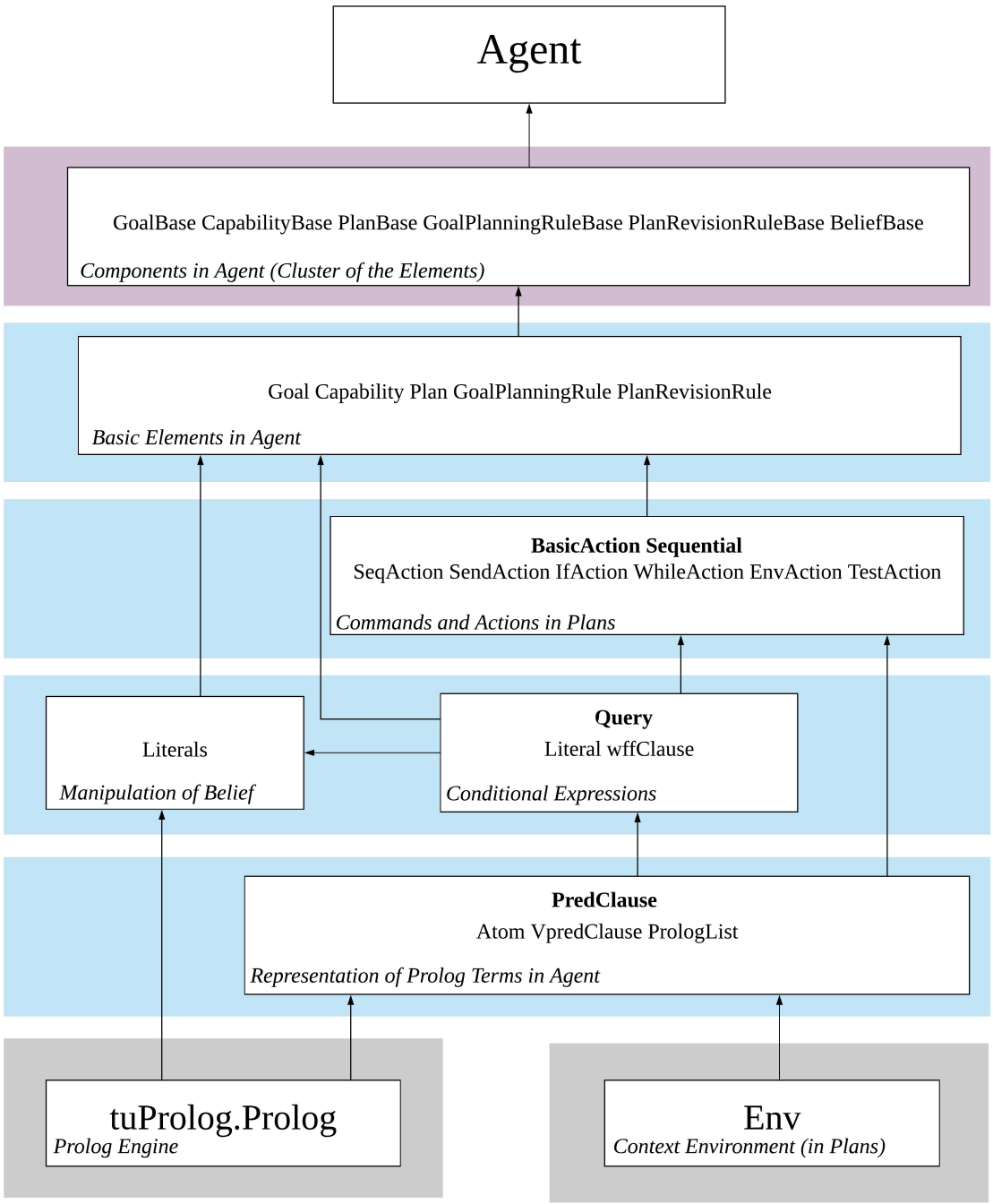

Figure 3: Agent structure

The full implementation of an agent will be explained next in a down-top order which means that the most basic components will be explained first and then we will move on to higher level components and explain how they are constructed.

6.1.1 tuProlog and Prolog Engine

The Prolog reasoning mechanism in this project is completed by an external library called tuProlog [21]. Before introducing any implemented classes, the Java interface of tuProlog will be introduced first, including $Theory$, $TheoryManager$, $SolveInfo$, Var and $Prolog$ classes;

The $Prolog$ class is the implementation of Prolog engine, it can store theories and perform queries on its theory base. It provides the following methods which are used in this project:

- addTheory(Theory): this method add a theory to the engine's theory library for querying.
- getTheoryManager(): this method is used to get the theory manager of the engine which will be used to retract theories.
- solve(String): this method takes a valid Prolog clause and returns a $SolveInfo$ class which contains boolean value indicating if the query is successful or not and information about potential substitutions of variables.

The $Theory$ class represents theories (facts) in Prolog which will be queried. It has a constructor with a string parameter.

The $TheoryManager$ class has a method $retract$ which takes a Prolog clause in string form as parameter and retract all the theories in the engine's theory library.

The $SolveInfo$ class contains all the information after a query is made. It has following methods which are used in this project:

- isSuccess(): it returns a boolean value indicating whether the query is successful or not.
- getBindingVars(): it returns a set of Var objects which contains variable and its binding if the query is successful.
- getVarValue(String): it takes a variable's name and return the binding value of it.
- hasOpenAlternatives(): there are many cases where there are not only one set of bindings available. The method will return true if that's the case.
- solveNext(): when there are more sets of bindings available, this method will solve and get another set of bindings. After this method is called, the $getBindingVars()$ will return the new set of bindings.

The Var class represents the variables in Prolog and their binding values if its available. The name of the variable can be got by calling $getName()$ method while the binding can be got by $getTerm()$ method.

As explained above, the retraction and addition of theories and querying can all be done by using Prolog clause in string form which is the approach taken in this project. In addition, all the other Prolog terms such as values, variables and lists are all represented by strings. However, there is another approach which requires Prolog clause to be constructed by $Number$, Val, Var and other appropriate classes systematically. This approach is more efficient since it does not require parsing the string but it is not being used due to the its complexity in this project. However, it can be implemented by changing a reasonable amount of implementation in some classes including $PredClause$ and $Atom$.

After explaining the tuProlog library and its usage in this project, the classes implemented in this project will be introduced.

6.1.2 Context Environment

Each of the capabilities, plans, rules and some of the actions in them have their own context environment just like functions, programs and block in C programming language. This context environment is represented by *Env* class and it will be applied in Prolog term level, meaning that the representations of Prolog terms in this project will be used to apply and manipulate this context environment. Therefore, before moving to any higher level of implementations, the *Env* class will be introduced. The *Env* class has a subclass $VVPair$ which is simple data structure storing variable and its binding value. The two methods provided by the *Env* class are used to manipulate the bindings.

- changeVal(String,String): the method takes the variable and its bindings. If the variable is unknown in the environment, that variable will be added to the environment with its binding. If the variable already exists, the binding of it will be changed.
- getVal(String): the method takes the variable and return the binding of it if it is available. If the variable doesn't exist, the variable itself will be returned.

6.1.3 Representation of Prolog Terms in Agent

PredClause and VpredClause
The *PredClause* class is used to represent not only predicate clauses (compound terms), but also the pairs. In addition, it is designed as the only class to connect Prolog engine and other components in the agent, therefore, it is inherited by classes that represent all other kinds of terms in Prolog except Horn clauses including *Atom*, $VpredClause$ and *PrologList*. In the compiler, the *PredClause* object can be generated by non-terminal *GpredClause* and *GsubpredClause* which are used to represent the clauses that contains no variables. The $VpredClause$ is designed to be the same as *PredClause*, it is used by non-terminal $VpredClause$ and $VsubpredClause$ to represent clauses with variables. The constraints of having variables or not is made in the parser in order to forbid *belief* clauses (except Horn clauses) containing variables.

There are three attributes in a *PredClause* object which are $functor$, *arguments* and *operators*. As the name suggested, $functor$ represent the functor of a clause. Therefore, it must be a symbolic value with first character in lower case. It is also possible for the functor to be *null*, which is used to represent pairs instead of clauses. In the meantime, the *operators* attribute is set to be *null* by default when the object is used to represent clauses. The *operators* attribute is used to represent different operators in a pair in order which could be $and(,)$,$or(;)$,$not(\backslash+)$. For example, a pair of $(a, b; c)$ will be represented as: functor = null; arguments = ["a","b","c"]; operators = [",",";"]. Due to the fact that terms in Prolog can be nested, the *arguments* attribute is implemented as a list of *PredClause* objects. This recursive implementation allows unboundedly nested terms and the recursion can be stopped by *Atom* object which represents a simple term.

The methods of the *PredClause* has same signature and functionality as those of *Atom* and the details of how *Env* class substitutes the variables will be explained later.

Atom
In this implementation, the most basic components is called *Atom*:

As shown in the design of compiler, the *Atom* class is used to represent variables and values in the Prolog which include both numeric ones and symbolic ones. The *Atom* class is implemented as a simple wrapper for *String* which provide basic functionality that if the string is representing a variable, it can be replaced by a value if that substitution is available in the context environment.

The variables in Prolog is defined as symbolic values with first character in upper case which means that it is easy to distinguish between values and variables by checking the first character. In addition, all the numeric values are also stored as strings and will be dealt by the Prolog engine during reasoning.

The $toString(Env)$ method is used to generate the string form of that *Atom* while context environment is provided while the $applyEnv(Env)$ method is used to construct a new *Atom* preserving the context information even if the context environment is removed which is essential when sending messages to other agents or the agent management system.

PrologList
The *PrologList* class is used to represent list terms in Prolog which have the form: $[a, b|Rest]$. Therefore, the there are two attributes of the class: *restPart* which could be either a Prolog list or variable representing the *Rest* part of the list and *arguments* which represents the elements in the head of the list.

6.1.4 Conditional Expressions

Query
The *Query* class is an abstract class used to represent the conditions in an agent. It is inherited by *wffBinary*, *Literal* and *TrueQuery* which implements the actual expressions. It can be considered as another form of Prolog clauses following the conventions of many structured programming languages which is the programming style used in the plans and rules of agents. The *Query* class uses "and", "or", "not" operators and parentheses to construct well-formed formula which is used in guards of plan revision rules and goal planning rules, if/while expressions in plans and pre-condition of capabilities.

Instead of returning simply true or false, the *Query* object will be handled by the Prolog engine when $PerformQuery$ method is called and it will return information about potential substitutions if there are any variables involves. Normally, the first possible set of substitutions will be saved to the context environment for later use.

TrueQuery
As name suggested, *TrueQuery* is considered to be a special case for condition expressions which simply return true whenever it is executed. The implementation is therefore simply return string "true" when *toString* method is called with or without context environment. Therefore, it will also return true and no substitutions when $PerformQuery$ method is called.

Literal
Literalclass is used to represent a simple boolean expression which is constructed by a Prolog clause and a potential negation operator. The *toString* method works the same way as the methods of *PredClause* except that if the additional *isNeg* attribute is set to be true, the additional negation operator will be added in Prolog form: "$\backslash +$". As explained before, the $PerformQuery$ make use of functionality of the tuProlog engine which accept a string as input and return a boolean value with potential substitutions which are wrapped by a $SolveInfo$ object. The implementation is simply to use *toString* method with context environment and use the returned value as input for Prolog engine and solve it.

wffBinary
As name suggested, *wffBinary* class is used to represent the well formed formulas in the condition expression. There are two operators available which are *and* and *or*, they are distinguished by the attribute *isOr*. When method *toString* is called, the formula is converted into a Prolog pair of two terms with proper operator in string. In this way, the order of applying operators are fixed like having parentheses in well formed formulas.

6.1.5 Manipulation of Beliefs

As discussed in Section 2.1 and Appendix B, the only components of an agent that can directly manipulate its belief base is the capabilities. Such manipulation is carried out by the list of post-conditions of such a capability which is implemented as *Literals*, a wrapper class for a cluster of *Literal*. The *Literals* class contains one field called *literls* which is simply a list of all *Literal* it contains. The $performChange$ method will be used the capabilities which takes the Prolog engine of agent as parameter and manipulate the theories (also called beliefs or facts) of it. The $performChange$ method will iterate over the all *Literal* and perform the changes they proposed one by one. If the *Literal* is negative, *retract* method from $TheoryManager$ of the Prolog engine will be called to retract all the theories that can match the clause of *Literal*. If the *Literal* is not negative, the clause of *Literal* will be converted to a string as we discussed before and that string will be used to construct a theory. That theory will then be added to

the Prolog engine using its *addTheory* method.

6.1.6 Actions (Commands) in Plans

Similar to commands in many structured programming languages, actions are the only elements that are used to construct plans, they can be classified into two categories: single step action and sequential action.

Single step actions include $EnvAction$, $SendAction$, $CapAction$, $TestAction$.
Sequential actions include $SeqAction$, $WhileAction$, $IfAction$.

Both types of actions are inherited from a class called *BasicAction* which is an abstract class and will be introduced first.

BasicAction
For each action (*BasicAction* object) there are five necessary methods:

- init: the initialization method which will be called before the first time the agent is started, it takes capability base of the agent and connects the capabilities to the actions that require them. It is the same as linking function call to the function's body.
- oneStep: this method is called when the action is executed. It takes a context environment, a Prolog engine and the agent object itself as parameters. An integer value will be returned representing the status of the execution.
- ModifyEnv: this method is used to modify the context environment of an action which is only useful for some actions and only when plans are being revised.
- clone: this method is used to create a deep copy of the action by creating new object with same arguments which will be used in planning rules and revision rules where plans and actions can be produced multiple times.
- toString: this method will constructed a string for the action which will be used for matching different plans in plan revision rules. This method is also a useful feature for debugging the software.

The output signal of *oneStep* method could be

- 1: Action executed successfully and finished.
- 0: Action executed successfully but not finished, will be continued in next step.
- -1: Action execution failed, this plan will be suspended until changes have been made to the belief base.
- -2: Action execution failed, trying to execute an abstract action, the agent is illy programmed.

These signals will be return to the agent and will be handled properly in agent level.

CapAction
The capability actions (*CapAction* objects) are the actions that are used to call the capabilities like function call. Each of such actions contains a *predicate* which is the name of the capability and *arguments* being used. In addition, the *cap* represents the capability being linked to the action which is set to be *null* as default. When the *init*(*CapabilityBase*) method of a *CapAction* is called, the *CapabilityBase* of the agent will be searched, if one of the capabilities has the same name and parameter number, it will be set as the *cap* field of this action. In addition, all the capability actions that does not match with any capabilities are considered to be abstract actions. They are expected to be used as placeholders by plan revision rules and will be replaced by other actions.

When the *oneStep* of a concrete capability action is called, if the arguments include any variables, they will be replaced by values according to the context environment provided. A new context environment will then be created with substitution information of the substituted arguments in action and parameters in capability. The capability will be performed with the created new context environment.

If the pre-condition of capability is evaluated as true, the post-condition will be used to change belief base of the agent and return with signal 1. If the pre-condition does not hold true, signal -1 will be returned.

TestAction
The *TestAction* class represent the test action whose implementation is straightforward. The only field contained is *query* of the action. When the *oneStep* method is called, the query is performed in Prolog engine provided. The returned signal is 1 or -1 according to the result of Prolog reasoning. In addition, if the result of query is true, then any substitutions need to be done for that query is added to the context environment.

SendAction
The message sending actions are represented by the *SendAction* class. Similar to messages, this type of actions contain four components: performative, receiver, content and additional information while the sender is implicitly set as the agent itself. The *oneStep* method will simply construct a new message (*Message* object) using these components and send it to the container of this agent through agent's *sendMessage* method.

EnvAction
The environment actions are represented by the *EnvAction* class. Since there is no limitations on the number and meaning of arguments, they are stored in a list while preserving the order. Each of the actions is assigned a unique ID so which will be used in the requests to the environment and responses from environment. This ID can help the agent to identify which environment action is responded and therefore determine whether resend the request or continue with the next action. Moreover, the *requestSent* boolean value is used to indicate whether there has been a request sent to the environment but have not been responded so that the request will not be sent repeatedly.

When the *oneStep* method is called, the value of the *requestSent* will be checked first. If the *requestSent* is false, then a new environment action request (*EnvironmentAction* object) will be constructed and sent to the container of this agent using agent's *sendActionRequest* method. The returned signal will be -1 which can prevent this plan from being called constantly and the agent will be able to commit to other plans. If the *requestSent* is true, the received environment responses will be checked. If there is no response for this action's request, nothing will be done and signal -1 will be returned. If there is a response for this action's request and the request is successful, then the post-conditions in the response will be added to agent's belief base and action will return signal 1 with *requestSent* field set to false. If the request is failed, the *requestSent* field will still be set to false but the returned signal will be -1 which means that if the agent's belief base is changed by other plans or messages, this plan will be waked up and the request will be sent again unless it is revised by plan revision rules.

Sequential
The *Sequential* class basically represents all the actions that can not be performed by a single step. All of them can potentially contain multiple other actions which makes them the only actions that can produce signal 0 (successful but incomplete) after execution.

The additional field *components* represents the sub-actions it contains and *currentPos* indicates which sub-action will be executed next.

There are also two additional methods implemented for the *Sequential* class. However, in order to implement and explain them properly, a helper class *CodePosition* is introduced first. This class is simply a recursive implementation of single linked list while the *Sequential* actions can also be implemented recursively. It is used to represent the position of an action in a potentially nested *Sequential* action. Each object of *CodePosition* class contains an integer representing the position of the code in current *Sequential* action and another *CodePosition* representing the position of the code in next *Sequential* action. The *nextLevel* will be set to *null* when the action is in the current *Sequential* action.

The *singleMatch* method is used by the plan revision rules tp try to find a sequence of sub-actions in this *Sequential* action that is the same as another sequence of actions. The method takes a *SeqAction*

which is simply a sequence of actions and compare it with this *Sequential* action. A *CodePosition* parameter is used (set as not *null*) if this *Sequential* action belongs to another *Sequential* action, and that *CodePosition* contains information about how to find this *Sequential* action from the top level of *Sequential* action. A list of *CodePosition* will be returned which contains the starting position (how to find the sequence of actions from the top level *Sequential* action) of all possible matches. The logic of this method is simple: use the *toString* method provided by *BasicAction* class, convert each components to string systematically and check if there is a match between components in new *SeqAction* and this *Sequential* action. If some components in this *Sequential* action are also *Sequential*, call the *singleMatch* method of it with properly constructed *CodePosition* and add the results to the final returned list.

The *replace* method is used to replace a sequence of actions with a new sequence which will typically be called after matches has been found by plan revision rules. This method takes three arguments: a *CodePosition* representing the start of the old sequence, an integer value representing the length of the old sequence, and a *SeqAction* representing the new sequence.

SeqAction
The *SeqAction* class is the most basic *Sequential* action which simply represents a sequence of actions. Therefore, it is also used as the top-level action in a plan (a single *SeqAction* is used to contain all the actions in a plan).

The only method needs to be explained is *oneStep*. When *oneStep* method of *SeqAction* is called, the action at *currentPos* in the components will be executed. If that action returns signal 0, then the *currentPos* will not be increased since that action is not completed and a signal 0 will be returned by this method. If that action is not the last action in components and it returns signal 1, then the *currentPos* will be increased by 1 and a signal 0 will be returned by this method indicating that the whole sequence is not yet completed. If that action is the last action and signal 1 is returned, the *currentPos* is set to 0 and a signal 1 will be returned by this method indicating that the whole sequence is completed and is ready to be executed again.

IfAction
The $IfAction$ class is constructed by one *Query* condition expression and two *SeqAction* objects which represents the sequences of actions in then-branch and else-branch. An additional boolean value *currCond* is used to indicate which branch will be executed. Due to the fact that the variables introduced in the two branches should not be introduced to the upper level of context environment, an *insideEnv* field is introduced to store the context environment being used in the two branches.

The two method specifically implemented by the $IfAction$ is *oneStep* and $ModifyEnv$. When *oneStep* method of *SeqAction* is called, the action at *currentPos* in the components will be executed. If that action returns signal 0, then the *currentPos* will not be increased since that action is not completed and a signal 0 will be returned by this method. If that action is not the last action in components and it returns signal 1, then the *currentPos* will be increased by 1 and a signal 0 will be returned by this method indicating that the whole sequence is not yet completed. If that action is the last action and signal 1 is returned, the *currentPos* is set to 0 and a signal 1 will be returned by this method indicating that the whole sequence is completed and is ready to be executed again.

The $ModifyEnv$ method of $IfAction$ also needs to be explained specifically. This method is used when plan revision happens during the execution of a $IfAction$. This method takes a new contextual environment and simply iterate over all the bindings in the new environment and modify the *insideEnv* of this $IfAction$ (by adding new bindings or change existing bindings) according to it.

WhileAction
The $WhileAction$ class is constructed similar to the *SeqAction* with an additional condition expression. For the *oneStep* method, it will execute the action at *currentPos* of the *components* of this $WhileAction$ which is similar to the *SeqAction*. However, different from the *SeqAction*, the condition expression is also considered to be an action and it is considered to be the first action of the sequence followed by other actions in the *components*. When the condition expression is queried, if the result is *true*, then the *currentPos* will be added by one and a new context environment will be copied from the context environ-

ment passed to this *whileAction* as *insideEnv*. The return signal will be 0. If the condition expression is queried and result is $false$, then the return signal will be 1 and nothing else will be done. When the condition expression is already queried and an action in the *components* is executed, it will always be executed using *insideEnv* instead of directly using context environment provided to the $WhileAction$. The return signal will always be 0. In addition, if the action being executed is the last action and is finished, the *currentPos* will be set to 0 so that the condition expression will be queried again and the *insideEnv* will be reset to *null*.

Similar to $IfAction$, the $ModifyEnv$ will also be used to modify the *insideEnv* when the action is executing and plan revision happens.

The two method specifically implemented by the $IfAction$ is *oneStep* and $ModifyEnv$. The only method needs to be explained is *oneStep*. When *oneStep* method of *SeqAction* is called, the action at *currentPos* in the components will be executed. If that action returns signal 0, then the *currentPos* will not be increased since that action is not completed and a signal 0 will be returned by this method. If that action is not the last action in components and it returns signal 1, then the *currentPos* will be increased by 1 and a signal 0 will be returned by this method indicating that the whole sequence is not yet completed. If that action is the last action and signal 1 is returned, the *currentPos* is set to 0 and a signal 1 will be returned by this method indicating that the whole sequence is completed and is ready to be executed again.

6.1.7 Basic Elements in Agent

As described in design section, the basic elements of an agent are those who construct the bases of the agent, which are belief, goal, capability, plan, goal planning rule and plan revision rule. In this implementation, as the graph shown, each element is represented by a Java class except the beliefs. Beliefs can be represented by *PredClause* class and no special methods are required.

Goal
As discussed in the full syntax of 3APL , each goals can contain multiple sub-goals that need to be completed which are represented by the *subgoals* field. Another field called *locked* is introduced in order to store the information about which sub-goals have already been used by goal planning rules and have a committed plan linked to it.

The goal completion in an agent is implemented that once a goal is completed, it will be added to the Prolog engine of the agent. The *checkGoals* method is used to check that if the Prolog engine contains facts that matches one of the sub-goals. If only part of the sub-goals can be matched, the whole goal is not considered to be completed and therefore it will return false and nothing will be done. However, if all the sub-goals can be matched, then the returned will be true indicating this goal is completed and all those facts matching sub-goals will be removed from the theory base of Prolog engine.

The *checkExist* method will check if a goal exist as a sub-goal in this *Goal* and return true or false accordingly.

Similarly, the *blockGoal* method will also check if a goal exist as a sub-goal in this *Goal* first, however, it will further check if that sub-goal is blocked. If it is block, false will be returned indicating that the goal either does not exist or a corresponding plan has already been committed for this goal. If it is not blocked, true will be returned and it will be blocked, so that a plan corresponding to it will be committed only once.

Capability
Similar to the design of capability in the 3APL language, the implementation of capability also contains four elements: a $functor$ and its *arguments*, a *precondition* and a *postcondition*.

When the $perform$ method is called (by the a $CapAction$ object), a context environment will be provided containing bindings to its arguments and the Prolog engine of the agent will be provided and manipulated by this capability.

The *perform* method will first apply all substitutions to its arguments and then query the Prolog engine regarding its pre-conditions. If the query failed, false will be returned indicating that this capability can not be performed. If the query succeeded, all bindings generated by that query will be store in the context environment which will be used for post-conditions. Then the post-condition will be applied to the Prolog engine in order using *addTheory* or *retract* method according to if a clause is positive or negative.

Plan
The *Plan* class represents a plan in the agent, it can be considered as a wrapper class for a *SeqAction* object which represents a sequence of actions and provides special functionalities needed by a plan.

It contains several fields, including:

- plan: a *SeqAction* object representing the actual sequence of actions need to be executed in this plan.
- insideEnv: a *Env* object representing the context environment (top level environment) will be used by the actions in this plan.
- associatedGoal: a *Goal* object representing the goals linked to this plans which will be considered as completed after this plan is finished.
- createdBy: a *GoalPlanningRule* object representing the goal planning rule that creates this plan, it will be set to *null* if this plan is not created by goal planning rules.
- firingCondition: a *String* object representing a Prolog clause which is the condition under which this plan was created by the goal planning rule. This field and the *createBy* field are used to prevent the same goal plan rule being fired multiple times by the same condition.

The *oneStep* method is called when this plan is executed. It will trigger the *oneStep* method of *SeqAction* (*plan* field) using the *insideEnv* as the context environment for the actions. This method will return whatever signal the *SeqAction*'s *oneStep* method returns.

The *complete* method is called when a plan is completed, and it will add the facts that the *associatedGoal* of it is completed to the agent's Prolog engine.

The *match* method is a helper method used by the *revisePlan* method which will call *singleMatch* method of the *SeqAction* and return the code positions that a match has been found.

The $ModifyEnv$ method is also a helper method used by the *revisePlan* method which will modify the *insideEnv* and all lower level context environments being used in the plan according to a new context environment (add new bindings and modify existing bindings but keep all other bindings that are not involved).

The *revisePlan* method will take two *SeqAction* parameters which one of them represents the pattern of that sequence of actions that need to be found and replaced with the new sequence of actions which is the second *SeqAction* parameter. It will also takes a condition expression (*Query* object) representing the condition it needs to satisfy in order to revise this plan. The plan can only be revised if the condition expression is true and the pattern can be found in this plan. If the plan is revised, a new context environment will be created which contains all the substitutions of variables that need to be done in order to satisfy the condition expression. That new context environment will be used to modify the environment of the old plan by calling $ModifyEnv$ method. The return value of this method is a boolean value indicating whether the plan is revised or not.

The *init* method is used to connect all the capability actions with capabilities which will call the *init* method of *SeqAction*.

GoalPlanningRule
As the design of goal planning rule in the 3APL language shows, three fields are contained by a *GoalPlanningRule* object which are a goal, a condition expression representing the guard and a *SeqAction*

representing the plan that connected to the goal. An additional field called *appliedCondition* is used to store all the conditions that have been used to fire an active plan, so that this *GoalPlanningRule* will not be fired by the same condition again until that active plan using the same condition is finished.

The *deleteFireCondition* method will be used by the plans that are created by this goal planning rule, it will called after a plan is finished and the condition that creating that plan will be deleted from the list of applied conditions.

Same as the *init* method in plans, the *init* method of the goal planning rule is also be used to connect capability actions with capabilities when the agent is initialized.

The *execute* will be called to check if a plan can be generated by this goal planning rule. It will first execute the guard (condition expression) and if the result is true, it will save all the bindings to a new context environment and check upon the goal base to see if all sub-goals of *associatedGoal* exist in the goal base of agent and are not locked. If they are, they will be locked and a plan will be created and add to the plan base of agent using the new context environment as *insideEnv* and *SeqAction* plan as the actual sequence of actions in that plan. The return value is also a boolean value indicating if a new plan is created or not.

PlanRevisionRule
As the design of plan revision rule in the 3APL language shows, three fields are contained by a *PlanRevisionRule* object which are a condition expression representing the guard, one *SeqAction* representing the plan pattern need to be matched and another *SeqAction* representing the new sequence of actions that should be replaced with.

The *PlanRevisionRule* has only one method that need to be implemented which is *execute*. This method will simply iterate over the plan base of the agent and try to revise as many plans as possible, it will call the *revisePlan* method of each plans with the *PlanRevisionRule* 's components and return a boolean value indicating if any the plans are revised.

6.1.8 Components of Agent

In this sub-section, the components which are directly used to construct an agent is introduced. This implementation strictly follows the pattern introduced in the description of language section so that each *Agent* object contains six components: *BeliefBase*, *GoalBase*, *PlanBase*, *CapabilityBase*, *PlanRevisionRuleBase* and *GoalPlanningRuleBase*.

BeliefBase
As discussed in the beginning of implementation section of agent, the runtime belief base of an agent is represented by the Prolog engine of it instead of the *BeliefBase* object. Due to the fact that all the Prolog reasoning procedure is done by the Prolog engine, such implementation will significantly reduce the effort needed. The *BeliefBase* is used to represent the initial belief base of an agent. It is used to stored all the beliefs an agent should contain before it get executed which are explicitly defined by the programmer of the agent in code. It has two fields storing two kinds of initial beliefs: Horn clauses and other Prolog clauses which does not contain any variables. There is no specific class implemented for Horn clauses, they are simply store in plain text by string which can be interpreted by the Prolog engine.

The only method implemented is *initial*. This method will simply add all the beliefs of the agent to its Prolog engine by converting *PredClause* objects to strings and use *addTheory* method. After the agent is initialized and the *initial* method is called, the *BeliefBase* object of an agent is no longer needed.

GoalBase
The only field of *GoalBase* is an list containing all the goals of the agent. And the methods implemented by the *GoalBase* have similar functionalities to those implemented by *Goal*. The only difference is that the methods in *GoalBase* will iterate through all the goals instead of a single rule. The returned value will be set to true if any of the method calls of the goals return true.

The only additional method is *isEmpty* which will return a boolean value indicating if goal base of an agent is empty or not. This method is one of the methods that will be used to determine if an agent should be put into $FINISHED$ state.

CapabilityBase
The *CapabilityBase* class simply servers as an cluster of capabilities and the only method is provided is to search the capability base by functor and argument numbers.

PlanBase
The *PlanBase* class has a field *plans* containing all the plans of an agent. The *oneStep* method will be called in the deliberation cycle of agents which will execute each plans in the agent for one step. It will also return a integer value of 1 or 0 indicating if any plans are executed.

The *addPlan* method will add a plan to the plan base which is essentially used by goal planning rules. The *revisePlans* method is used by plan revision rules which will iterate over all the plans in the plan base and revise them if it is possible using the given pattern and condition.

The *initial* method will simply call the *init* methods of all plans and link the capability actions in them to the capabilities in capability base.

The *isEmpty* method simply returns if there are any plans in the plan base or not. It is also one of the methods to determine whether an agent should be transferred to $FINISHED$ state or not.

GoalPlanningRuleBase and PlanRevisionRuleBase
Both *GoalPlanningRuleBase* and *PlanRevisionRuleBase* have a field containing all the corresponding rules in the agent. They also provide a *initial* method which will also iterate over rules in the agent and link their capability actions to capabilities. The *execute* method will iterate over all rules and try to execute by calling their *execute* method.

6.1.9 Agent class

In addition to the six elements and Prolog engine as we discussed above, an *Agent* object also contains a unique identifier, a *Container* object which is the container containing it, a name, a runtime state, a integer *clock* representing how many deliberation cycle the agent has completed and a boolean value *terminate* indicating whether the agent has received a terminating signal.

When an agent is created, added to a container and the container is started for the first time. Agents' *initial* methods will be called which will create an empty Prolog engine and use *initial* method of its elements (bases) with proper arguments to make sure that the agent is ready to run and all initial beliefs have been added to its Prolog engine. In addition to all these expected behaviors, the *initial* method will also load a Java implemented Prolog library into the Prolog engine which provides basic arithmetic operators since tuProlog does not provide these operators in a convenient way. The provided operators include: $sum(arg1, arg2, result)$, $sub(arg1, arg2, result)$, $div(arg1, arg2, result)$, $mul(arg1, arg2, result)$, $pow(arg1, arg2, result)$ (power), $root(arg1, arg2, result)$, $toInt(arg1, result)$, $toFloat(arg1, result)$, $eql(arg1, arg2)$ (equal), $grt(arg1, arg2)$ (greater), $les(arg1, arg2)$ (less), $geq(arg1, arg2)$ (greater or equal), $leq(arg1, arg2)$ (less or equal) and $abs(arg1, arg2)$ (absolute value).

In addition, as the names suggested, the *sendMessage* and *sendActionRequest* methods are used by the *SendAction* and *EnvAction* to send all kinds of messages. The *receiveMessage* and *receiveResponse* methods are used to receive agent messages and environment responses which will be used directly by the *deliberation* method. They will return boolean value true if any messages are received.

In order to run different components of the multi-agent system in different threads, *Agent* class is extended from *Runnable*. The *run* method of agents will call *deliberation* within a while-loop to perform the deliberation cycle constantly until a terminating signal is received or the agent has finished all the goals and plans and put itself into $FINISHED$ state.

Finally, the *deliberation* method which is used to represent a full deliberation cycle will be explained. It is implemented relatively straightforward since all the necessary methods that need to be executed during the deliberation cycle have already been implemented by either elements (bases) of the agent or the agent itself. Each of these methods will be explained in the order they will be executed along with the actual functionality it provides to make the explanation clearer.

- 1. $Agent.receiveMessage$: try to receive messages from the container.
- 2. $Agent.receiveResponse$: try to receive environment responses from the container. (In addition, if the agent is in $SUSPEND$ state and no messages or responses are received, skip the rest of deliberation cycle. If any messages or responses are received, the agent will be put into $ACTIVE$ state regardless of its current state.)
- 3. $PlanRevisionRuleBase.execute$: try to apply plan revision rules to plans. (try every combination of rules and plans)
- 4. $PlanBase.oneStep$: try to execute one step for each plans.
- 5. $GoalPlanningRuleBase.execute$: try to generate new plans using goal planning rules. (In addition, if no rules and plans are applied or executed, the agent will be put into $SUSPEND$ state.)
- 6. $GoalBase.checkGoals$: check if there are any goals completed, if there are, delete them from goal base.
- 7. $GoalBase.isEmpty$ and $PlanBase.isEmpty$: check if the plan base and goal base are empty. (If both of them are empty, put the agent into $FINISHED$ state.)

The whole class diagram of this agent architecture will be provided in Appendix D.1.

6.2 Multi-agent System

Generally, most components of the multi-agent system are implemented as a runnable object and can therefore be run in separated threads. However, the environment interface which is considered as an object having APIs to connect environment and server while containing information about the relationship between agents and controllable entities is implemented as a un-runnable object that being used by both the server and environment instead of having a separate thread.

6.2.1 Server and Container

As discussed before, servers and containers are implemented so that they run on separated threads. Consequently, synchronization issues are essential, especially regarding the synchronization of messages.

Both containers and servers are implemented as objects of Runnable objects in Java, which have one run() method. When a runnable object is started in a new thread, the run() method will be called and that thread will be kept until the method is finished. The run() methods of containers and servers are implemented the same way as deliberation cycle of agents, which means that the method repeats a sequence of procedures constantly until a terminating or interrupting signal is received. For containers, a processing cycle involves the following procedure:

- If an agent is in READY status, change the status to RUNNING and start a new thread for it.
- Receive environment responses from server.
- Forward the responses to target agent.
- Receive messages from server.
- Proceed the received messages, which involves sending them to their destination.
- Forward the environment actions to server.
- Check if a terminating signal is received. If signal is received, forward it to all agents in the container and terminate.

For servers, a processing cycle involves the following procedure:

- If there is an environment connected to environment interface and it is in READY status, change the status to RUNNING and start a new thread for it.
- If a container is in READY status, change the status to RUNNING and start a new thread for it.
- Proceed the received messages, which involves sending them to their destination.
- Receive responses from environment.
- Forward the responses to target container.
- Forward the environment actions to environment interface.

These two processing cycle are different due to the fact that the procedures of transferring messages between different components of the system are different and will be explained in next section.

In addition to these basic functionalities, servers also have an additional feature which is to update the status monitoring page of graphical user interface. This feature is implemented by *updateTextArea* method which will be called approximately every second. This method will use update all the customized environment information, environment requests and responses and messages inside the multi-agent system to the *TextArea*s in the GUI of software.

6.2.2 Agent Messages

In order to carry out the communication properly, each messages in the multi-agent system is represented by *Message* class which contains five fields:

- Sender ID: the identifier of the sender, is constructed by both container ID and agent ID. This ID is constructed during the procedure of sending messages, meaning that when an agent sends an message out, the sender ID only contains its own agent ID. When the message is being handled by the container of the sender, the container will append its own ID to the sender ID.
- Receiver ID: the identifier of the receiver, is also constructed by both container ID and agent ID. However, this ID should be fully constructed when a message is sent. The receiver ID can also be "AMS", which means the agent management system. Messages with receiver ID "AMS" will be sent and handled by the container of the sender. This kind of message is
- Performative: this component identify what kind of information a message is carrying. It also specifies what response is expected by the agent and container.
- Content: the actual content being sent, it is represented by a Prolog predicate clause.
- Additional Information: the additional information could be used for assigning dedicated channel for the communication. It could also be simply used as supplement for main content.

The *Message* class is only a data structure to contain these five pieces of information and provides no special functionalities. In addition, once a message is received by the agent, there is no special representation for the received messages, they will be added as facts to the Prolog engine of agent in a format as following: *received(Sender ID, Performative, Content, Additional Information)*.

Furthermore, as shown above in the container and server section, containers and servers are implemented as runnable objects which run in different threads, in order to prevent race conditions, the Java *synchronized* block is used. The *synchronized* block takes an object as argument and all the blocks synchronized on the same object will not be able to run in parallel. This feature is used so that each message receiving and sending methods are wrapped by a *synchronized* block synchronizing on the message list objects they are about to manipulate. In addition, in order to prevent containers and servers from waiting for completion of the handling procedures for messages before sending new messages, a message buffer is introduced for each containers, agents and servers. The handling procedure will first copy messages from the received to that buffer and clear the received message list in a *synchronized* block, then start to handle each messages one by one in its own thread without synchronization.

Another notable implementation consideration is that the received message lists of agents are actually stored in their containers and such lists for containers are stored in the server. Those lists will only be copied and reset to empty when the agents or containers call their *receiveMessage* method. In this way, after messages are handled and their destinations are set, they will not be distributed until they are required.

Message Processing in Containers

To further clarify the procedure of handling messages, a detailed description of the process being taken by containers when they receive messages is provided as a flowchart with necessary explanation below.

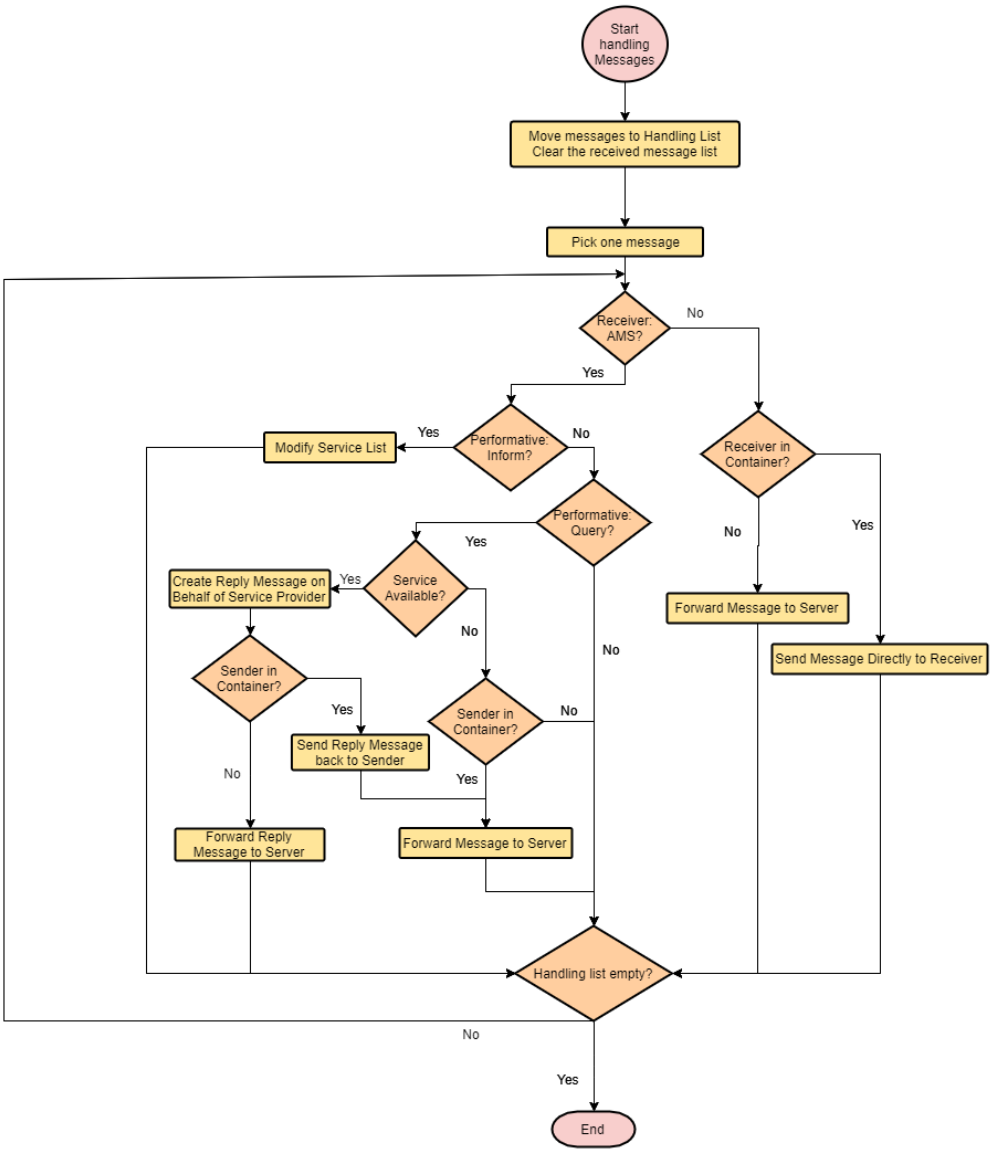

Figure 4: Procedure of handling messages within containers

As shown in the diagram, once a message is being handled, the containers will first distinguish messages about services from other messages by checking if the receiver of those messages are identified as agent management system. In addition, the only two valid performative for messages to agent management system are $Query$ and $Inform$. Therefore, messages to AMS with other performative are ignored. If the requested service is available in the container, a reply message will be created using the same content and additional information as the original message which informs this service. In stead of the container itself, the sender identifier of that reply message will also be assigned as the agent who provided the service.

Message Processing in Servers

In servers, the procedure of handling messages are relatively simple since that most work are done by the containers. The simple flowchart below shows this whole procedure. Note: if the receiver identifier of a received message is AMS, it also means that the performative of that message is $Query$ which is the only valid performative filtered by container. Such a message will be broadcasted to all other containers except the one that contains the sender.

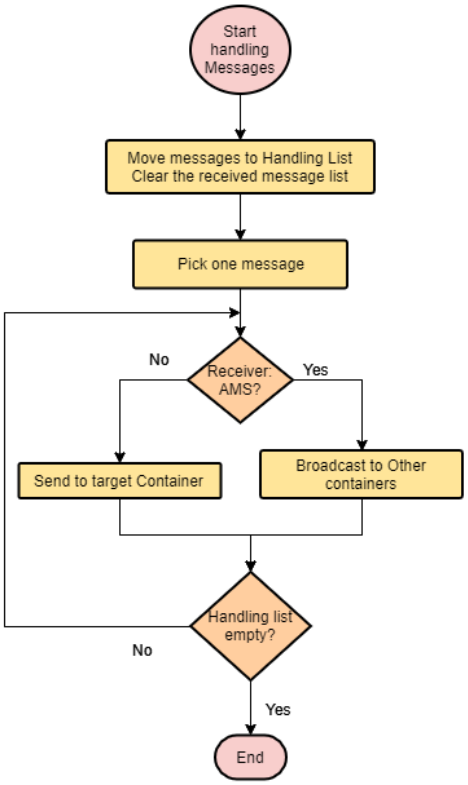

Figure 5: Procedure of handling messages within servers

6.2.3 Environment

Environment Interface class
The $EnvironmentInterface$ class is used to represent the interface between environment and the server. As discussed in the design section, the environment interface is used to map agents to entities and entities to agents so that the server does not have to contain any information about environment and environment does not have to contain any information about server. This mapping procedure is implemented by two $HashMap$ objects, the first one uses agent identifier as key and entity identifier as value, the second one uses entity identifier as key and agent identifier as value. In this way, adding, deleting and mapping from both side can be done in constant time. The environment interface also provides methods to send requests and responses between server and environment. It simply forward these messages using mapping to find the correct destinations and no other side effects will be taken.

In addition, the environment interface also provides a $updateEnvironmentInfo$ method which takes an integer value indicating the current tick of the environment and a string which is the information to be shown in the status monitoring page in graphical user interface.

Environment class
The *Environment* class is a Java abstract class which implements *Runnable* interface. All environments should extend this *Environment* class. The essential elements of an environment are a list of entities, a list of action requests, a list of action requests that are being handled, a list of responses that need to be sent and a runtime state.

The methods need to be implemented by programmers include:

- creatableEntities: this method will be called by the graphical user interface to show which entities

can be created and the arguments for creating them.

- createEntity: this method takes a string that represents the type of entity to create and a list of arguments. This method will return an integer which normally is supposed to be the identifier of created entity. However, if a negative value is returned, it will be considered that no entities are created. Therefore, this method can also be used to perform internal changes to the environment.
- proceedAction: this method take an *EnvironmentAction*, process it and potentially send the response.
- showGUI: this method will be called by the graphical user interface when the server is started, it is meant to show a graphical representation of the environment.
- run: this method is needed by any *Runnable* classes, normally it would contain a loop in which the environment keep receiving and handling the action requests and keep updating its own internal state.

Apart from these four methods, programmers should also define their own controllable entities by implement the *ControllableEntity* interface. The only method in the *ControllableEntity* needs to be implemented is *passiveSensing* which defines what information should be given to agent that connected to this entity without any request.

Note that there is no constraints about how to handle action requests and even whether to handle them or not, it is totally up to the programmers of the multi-agent system.

The environment classes should be compiled into *.class* files and will be dynamically loaded into the software in runtime using Java's reflection mechanism. ***Environment Action Requests and Environment Responses***
The environment actions and responses works similar to the agent messages. They are transferred through $agent - container - server - environmentInterface - environment$. Each of them contains three common components: agent ID, entity ID and action ID. As we have already discussed while introducing *EnvAction*, the action ID is used by the agent to link responses to actions. In addition, only one of entity ID and action ID is set when an action request or response is created, the other one will be filled by the environment interface using its mapping mechanic during the transferring procedure.

The environment action request has two other arguments which are the header of the request and the list of arguments being used for the request, both of them are set by the *EnvAction* in agent.

The environment action respond also has two other arguments which are a boolean value indicating if the action is successful or not and a list of *PredClause* objects to represent the post-condition of the action or information obtained by passive sensing.

6.3 Interpreter

6.3.1 Interpreter Structure

As discussed in the design section, the runtime manager of this interpreter contains all the servers that are currently in the software. These information will given to the graphical user interface to show to the user. This mechanic is implemented by wrappers classes of servers, containers and agents. They are called view classes:$ServerView$, $ContainerView$ and $AgentView$.

These view classes contain a user editable name and the an object of the original class. In addition, servers and containers' view class also contains a list of view objects of their components. In this way, when a server view object is selected to be shown by the user interface, the user interface will take the container view objects contained by it and display them in a different table. It is notable that this implementation require extra operations to be taken when containers and agents are added or deleted from containers and servers. The consistency of view objects and original objects must be maintained. To control the servers from user interface, the original objects will be fetched from the view objects and their methods will be used.

In order to create a server, container or agent, the corresponding factory or compiler will be used while a view object will also be created to contain it.

6.3.2 Server factory and Container factory

As described in the design section, the factories will use the constructor of the corresponding classes and a unique identifier will be generated and assigned. The implementation of this identifier generation procedure will be introduced below.

Each of the factories class contains static integer value which is set to be 0 called $currentID$ by default. When an object is being generated by the factory, its identifier will be assigned the same as $currentID$ and the $currentID$ will be incremented by 1 after that.

6.3.3 Compiler

As described the design section, the parser/code generator of this compiler is developed together. The way that compiler generate agents is the same as how the *Agent* objects are constructed by objects of its components as described above. The whole idea is that each terminals in the abstract syntax tree will be generated as a component and the parser/code generator will determine how these components can construct another higher level component.

The full abstract syntax tree is shown in the graph in Appendix D.2. The objects that will be generated by each terminals are indicated by the name of that terminal or shown in a pair of parentheses under its name. An additional note is that the unique ID of agents are also generated and assigned the same way as factories.

6.4 Graphical User Interface

The graphical user interface of this software is implemented using JavaFx with SceneBuilder.

6.4.1 Main Page

The main page of the software contains three tables. They are used to display servers, containers and agents separately. The tables will fetch information from the view objects in runtime manager. In addition, the container table depends on server table, meaning that the container table will be displayed as empty by default, and if a server is selected in the server table, container table will be updated with the container list of that server. The same rule is also apply to be agent table and container table.

In addition, as described in design section, there are several buttons in the main page providing different functionalities. There will be two buttons for adding and deleting each of the components (servers/containers/agents), one button for creating controllable entities, one button for start/stop selected server, one button for linking environments to servers and one button for opening status monitor.

- add and delete: they will simply make use of methods in factories and compiler and manipulate the list of view objects. It is also important that their behavior depends on the which server and container is selected, all adding and deleting will take place in them.
- start/stop: this button can only be clicked when a server is selected, the server will be started or stopped depending on which state it is in. Starting a server will also call the $showGUI$ method of the environment of the server if there is one.
- set environment: this button will invoke a JavaFx file selector, the selected file should be a compiled *.class* file of an environment class. This method will load that class into JVM and call its default constructor with no arguments to create a new environment and use the *setEnvironment* method of selected server to connect them.

- create entity: this button is only useful when there is an environment connected to the selected server and there is an agent selected. It will first call the *creatableEntity* method of the server's environment to check what entities are available and provide the name and parameters needed to be set in a new window. It will then use the *createEntity* method of the environment with the arguments entered by uses to create entities.
- check status: this button will simply open a new window of the status monitor for the selected server.

6.4.2 Status Monitoring

This page will display three type of status/information: communication in the multi-agent system, environment requests and responses and programmer defined information from the environment. These three text fields will be updated approximately every second.

The status monitoring page can be opened at any time by the button in main page. However, only one page can be active at the same time and this page will not memorize previous information. All the previous information will not be preserved once it is closed.

7 Examples

In this section, two examples of multi-agent system will be explained to illustrate to strengths of this interpreter.

7.1 Example: multiple agents with communication

In this example, a certain scenario will be simulated which is that one agent sales while the other buys. The seller agent has certain number of goods which are pens and pencils and try to sell them to others. It also provides prices of them. The buyer agent is willing to buy these items with the money in its pocket.

This scenario is chosen due to the fact that it has certain generality while both agents are reasonably simple. It demonstrates a services providing and querying procedure that is involved in many types of multi-agent systems. In addition, it also contains a handshaking procedure that buyers and sellers need to confirm each other's availability before a contract is made. Such a procedure is also commonly used for communication between different software entities and therefore illustrates the power of this interpreter. This multi-agent system will run in the following procedure:

1. Seller notifies the agent manager system about its selling service.
2. Buyer asks the agent manager system for sellers.
3. Buyer receives seller's identifier from the agent manager system and ask seller for item list.
4. Seller sends item list and their prices to buyer.
5. Multiple copies of the following procedure will run in parallel:
    - Buyer check if it has enough money to buy an item.
        - if yes, sends message to seller expressing the willing of buying an item and takes money from its pocket to its hand.
        - if not, do nothing.
    - Seller check if it has the item buyer requested.
        - if yes,
            (a) replies to the buyer to confirm this sale.
            (b) Seller will add the proper amount of money to its account.
            (c) Buyer will subtract proper amount of money from its hand.
        - if not,
            (a) replies to the buyer to cancel this sale.
            (b) Buyer will put the money back in its pocket.
6. Terminating:
    - Seller runs out of stock, it will send messages to cancel all contracts by sending messages to those buyers.
    - Buyer runs out of money and stops making query.

As shown above, in this procedure, buyer proposes contracts and seller passively process these contracts. In addition, this system can also handle the case of multiple buyers and sellers with limited modifications needed.

7.1.1 Seller

The seller agent starts with the follow beliefs: $money(0), price(pen, 2), price(pencil, 1), storage(pen, 12)$, $storage(pencil, 45)$ meaning that it starts with information about the amount and prices of items but no money while the only goal it has is to $sell()$ items.

There is only one plan in the plan base when it is created and that plan contains only one action which is $Send(inform, ams, is(seller), channel(1))$. This action will inform the container that it is a seller and it will be queried by buyer agents. The $channel(1)$ additional information is not useful in this case and can be replaced with others that actually contain some information.

The goal planning rules of the seller are mainly reactive meaning that they can be performed without any goals being committed. The first rule can be applied with guard $received(query, Buyer, price(), channel(1))$ which simply wait for buyers to query about the prices of items. The plan connected to this rule will send messages to the buyer containing the pricing information of the items. It is notable that in order to prevent from sending these messages repeatedly, the received query from buyer should be deleted once the replies have been sent. The core functionality of the seller is also programmed as a goal planning rule which will be fired after receiving requests from buyer to buy an item. The plan connected to this rule will confirm this request, send a reply and add proper amount of money to seller's account:

```
<- received(query, Buyer, buy(X), channel(1)) |
{
    sell(X, Buyer);
    Send(inform, Buyer, sell(X), channel(1));
}
```

The $sell(Item, Buyer)$ capability being used will remove one item from storage, adding the price to account and delete the query message from buyer:

```
{price(X,Price) and storage(X,Count) and sub(Count,1,NCount) and money(Money) and sum(Money, Price, NMoney) and not eql(Count,0)}
sell(X,Buyer)
{not money(Money), not storage(X,Count), not received(query, Buyer, buy(X), channel(1)), money(NMoney), storage(X,NCount)}
```

However, it is obvious that this plan will failed when there is no items left. Therefore, a plan revision rule is introduced, which will basically delete the request and send the reply to the buyer indicating that there are no such items left.

```
{
    sell(Item, Buyer);
    Send(inform, Buyer, sell(Item), channel(1));
}
<- storage(Item,0) |
{
    deleteOrder(Item);
    Send(inform, Buyer, no(Item), channel(1));
}
```

7.1.2 Buyer

The buyer agent is initialized with beliefs about the amount of money in its pocket and hand. The belief base also contains a special belief which is a Horn clause which is used as an helper function that check if the buyer has enough money to buy an item:

```
enough(Item) :- price(Item, Price, Seller), pocket(Pocket), geq(Pocket, Price).
```

Similar to the seller, the buyer also has only one plan in the plan base when it is created which is to find out an agent that is seller. This method will repeatedly send querying messages to the agent manager system until receives a reply. After retrieving the identifier of seller, it will send a message to the seller querying about the list of items and their prices. It will then delete the message about identifier of the seller since it is no longer useful.

```
{
    while(not received(inform, Seller, is(seller), channel(1))){
        Send(query, ams, is(seller), channel(1));
    };
    received(inform, Seller, is(seller), channel(1))?;
    Send(query, Seller, price(), channel(1));
}
```

Notably, there is a test action at line 4: $received(inform, Seller, is(seller), channel(1))?$ which seems to be redundant. This action is not meant to check if this message is received since the *while* action before it has already done so. Instead, it is used to retrieve the identifier of the seller which will be put into the context environment as variables $Seller$ for later use.

The goal planning rules of buyer are also reactive, one of them is used to handle the received messages:

```
<- true |
{
    getPrice();
    if(price(Item,Price,Seller) and not stock(Item,Amount)){
        createStock(Item);
    }else{
        true?;
    };
}
```

The first rule is used to extract price information from messages, the $getPrice()$ capability has $received(inform, Seller, price(Item, X), channel(1))$ as its pre-condition, and $price(Item, X, Seller)$ as its post-condition which will store both the seller identifier of the item and the price it provides. This procedure is actually not necessary, but it can make the rest part of this program more readable. In addition, using this structure to store the pricing information can also be used in multiple seller cases where each seller provides different price of the same item. The following actions will then create a stock for the item if there is not one. This if action also illustrates a use case that *test* action *true*? is similar to *pass* command in some other languages that occupies a line but does nothing. Since this rule has *true* as its guard, it will be run constantly. However, the first action in this sequence is a *capability* action, the pre-condition of that action can be treated as the guard of the rule which is why the guard is not necessary.

The other goal planning rule of the seller will be introduced together with its plan revision rule:

```
Goal Planning Rule:
    <- price(Item, X, Seller) and not received(inform, Seller, no(Item), channel(1)) and enough(Item) |
    {
        buy(Item,Seller);
        Send(query, Seller, buy(Item), channel(1));
        pay(Item,Seller);
    }

Plan Revision Rule:
    {
        pay(Item,Seller);
    }
    <- received(inform, Seller, no(Item), channel(1)) |
    {
        putBack(Item);
    }
```

The goal planning rule will try to buy an item when it has enough money. Instead of actually buying that item, the $buy(Item, Seller)$ capability will take the money from the pocket to its hand which meaning

that even though the money has not yet been spent, but it is committed to a purpose and can not be used again. Then it will send an query to the seller to buy that item. Referring back to the seller program, this query may result in two replies. The first one will be $received(inform, Seller, sell(Item), channel(1))$ where $Seller$ is the identifier of seller and $Item$ is the item being sold. This reply is the pre-condition of $pay(Item, Seller)$ capability, it will spends the money in its hand and add one item to stock. In this case, the contract is confirmed by both side and therefore completed. However, the reply message could also be $received(inform, Seller, no(Item), channel(1))$ which is guard of the plan revision rule. The rule will then be applied which revise the *pay* capability action to *putBack* capability, which will put the proper amount of money from its hand back into pocket. In this case, the plan revision rule serves similar to the *else* branch of an if command. But unlike the if command which will be executed immediately, the execution of this plan can be delayed for unbounded amount of time until a reply is actually received.

The full code of both agents can be found in Appendix E. It is important to point out that both implementations are around or less than 70 lines of code even being loosely formatted with multiple capabilities and rules that are only used for readability.

7.2 Example: multiple agents with environment

In this section more complicated agents and a graphical environment will be shown in order to demonstrate the ability of this interpreter to construct environments and complicated agents. However, there will be no communications between different agents in this example.

The scenario being implemented contains:

- Two types of garbage bins which will generate collection tasks: recyclable bin and non-recyclable bin.
- Two types of garbage station that gather garbage from bins: recyclable station and non-recyclable station.
- Cleaners: agents/controllable entities whose tasks is to collect and take garbages to corresponding stations.
- Recharge Station: stations that are used by the agent to regain power (the ability to move).

The garbage bins will corresponding garbages randomly if it does not contain any garbages and the limit of garbages in a bin is 100 units. The cleaners are able to contain 200 units of garbages, but the two types of garbages can not be carried together. Once the cleaner is on top of a garbage bin, it will be able to collect any amount of garbages without exceed its own capacity. It can also dump any amount of garbages to garbage stations. The cleaner also has a battery which can contain 500 units of power. The only action that takes power is moving, moving one step will cost 1 unit of power and that step can be taken in eight directions, meaning that cleaners can move in diagonal directions. The cleaners need to be on top of a recharge station to recharge, and it will be fully charged once it tries to recharge. It is also important that the cleaners only have the information of the environment within thirty steps around them.

In the graphical environment, the bins are shown as circle and the garbage stations are shown as squares with different colors. The non-recyclable ones are shown in black and recyclable are shown in green. The charge stations are big red circle and cleaners are represented by smaller brown squares. In addition, the graphical environment will only show the view of one cleaner at a time, that cleaner will always be in the center with the environment in the back moving. However, it is possible that the view of a cleaner contains other cleaners which will also be displayed as brown squares. There is a scrollable bar at the left bottom of the window which can be used to change which agent's view to show.

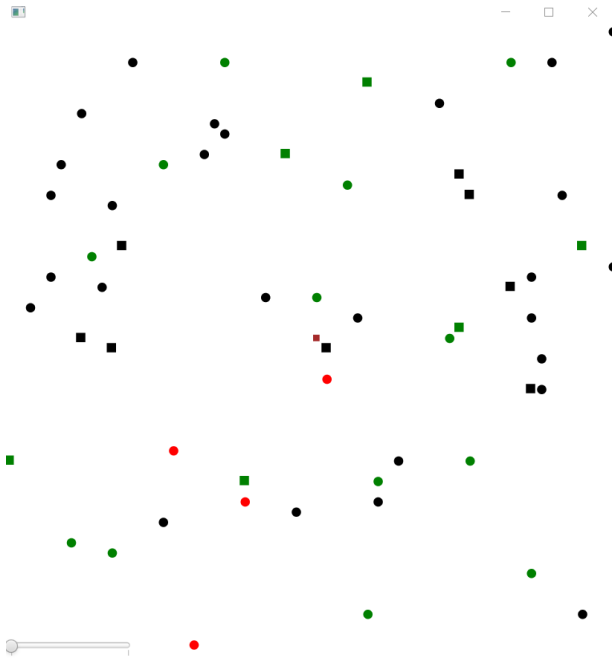

Figure 6: Graphical Environment of Cleaner

As we have introduced in design section, agents are connected to controllable entities in the environment which in this case are the cleaners. In this case, each agent will be connected to one and only one cleaner. The actions can be performed by the cleaners which are also the actions can be requested by agents include:

- init: initialize the agent by giving it the status of cleaner and its view (surrounding environment).
- pickup: collect a certain amount of garbages.
- move: move one step toward a specific direction.
- throw: dump all the carried garbages.
- recharge: recharge at recharge station.

Instead of focusing on the environment, the agents that controls cleaners will be introduced in detail. The belief base of cleaner agent contains a reasonably large amount of Horn clauses which are used for calculating and searching, they can be considered as helper functions for capabilities and rules. However, unlike seller and buyer agents, the cleaner agent created with information of the status of itself. When it is first created, the only plan in its plan base is to send an environment action request to get necessary information to start reasoning.

Once the agent is initialized, there are two goal planning rules available which represents two kind of modes the agent can be in. The first one is the normal collecting mode, which will calculate a score for every objects in the view of cleaner, rank them and move towards the location with highest score and try to perform corresponding actions. Note that after every environment actions, a new view of the environment will be returned as post-condition. The $saveView()$ capability is used to save these views using internal representation of agents.

```
<- not moving() and maxScore(X,Y,Max) |
{
    start();
    while(goto(X,Y,Direction)) {
        Env(move, Direction);
        saveView();
    };
    if(bins(A) and member(position(X,Y,Atr),A)){
        Env(pickup,200);
    }else{
        Env(throw);
    };
    saveView();
    finish();
}
```

For bins with same type as the garbages cleaner carrying, the score is calculated by the Horn clauses in belief base in the following way:

$$\frac{min(BinGarbage, CleanerCapability - CleanerGarbage)}{\sqrt{Distance}}$$

For stations with same type:

$$\frac{CleanerGarbage}{\sqrt{Distance}}$$

If the bins or stations do not have the same type as the cleaner, the score will be zero.

The other mode is exploring which means that either the highest score is too low or the battery is running low and there is no recharge station insight. In this mode, the cleaner will blindly move to up left for one step to explore more area.

```
<- not moving() and self(Xs,Ys,attribute(A,B,Battery)) and (not maxScore(X,Y,Max) or (not minRecharge(Xr,Yr,D) and leq(Battery, 150))) |
{
    start();
    Env(move, upleft);
    saveView();
    finish();
}
```

Note that both rules are wrapped by two capability actions *start*() and *finish*() conditioned on *not moving*(). The *moving*() condition serves as a *mutex* (or semaphore) while *start*() and *finish*() are used to lock and unlock to prevent these two mode being applied at the same time.

There is also a plan revision rule in the agent which can be considered as an emergency model. It will be used when the battery is running critically that it requires the cleaner to move to the nearest recharge at once in order to prevent from running out of power. This rule will overwrite other modes which can not be undone and command the cleaner to move to the nearest recharge station immediately. It is also a good use case for plan revision rules.

```
{
    while(goto(X,Y,Direction)) {
        Env(move, Direction);
        saveView();
    };
    if(bins(A) and member(position(X,Y,Atr),A)){
        Env(pickup,200);
    }else{
        Env(throw);
    };
}
<- self(Xs,Ys,attribute(A,B,Battery)) and minRecharge(Xr,Yr,D) and geq(D,Battery) |
{
    while(goto(Xr,Yr,Direction)) {
        Env(move, Direction);
        saveView();
    };
    Env(recharge);
}
```

The complete code of cleaner agents can also be found in Appendix E. There will also be a Java implementation of an agent applying the same logic but using different set of environment APIs available in the supplemental materials. To comparison, the Java version requires over 200 lines of code in different files while the 3APL version contains only around 100 lines of codes. In addition, there are certain functionalities that could be implemented more efficiently and effectively in Java rather than in 3APL, the environment all provides the opportunity for programmers to program in both languages and create hooks to facilitate the development of agents.

Note: the Java version of cleaner agents is implemented by Chonghan Chen using the package provided by Coursework 1 of G53DIA module. The cleaner example in this section follows the same environment setup as the CW1. More information can be found in those Java files.

8 Summary and Reflection

8.1 Project Management and Reflection

The whole interpreter is separated into different modules during the implementation. Development of each module in this interpreter is considered to be a sub-project. In this project, if the parallel development of each module is applied, placeholders that simulate the behavior of some modules will be required in order to test others which could be time-consuming since that each module is relatively complicated. Instead, the interpreter is developed from back-ends to front-ends, a linear approach is applied with minimum time overlap since both testing and development of each module highly depends on the completion of previous modules. For example, certain functionalities of containers such as services management highly depends on the message sending procedure of agents while these procedures also highly depend on the some internal implementation of the agent such as *SendAction* and *receiveMessage*. It would be easier to follow this linear dependency model to design and implement module by module instead of having to investigated large amount of effort purely into designing before do any implementation. In the meantime, examples of a 3APL multi-agent system are developed along with the interpreter in order to test and illustrate the interpreter's functionality. During the development of each sub-project, a typical waterfall model is used which follows a strict order of analyzing, designing, coding and testing due to the complexity of the sub-projects.

As the Gantt chart below shows, a great deal of time was spent on the development of compiler and agent classes which is also the part of the software that is most complicated. However, during the development of compiler, an issue appeared that due to lack of knowledge of agent-oriented programming before this project, the semantics of the language itself is hard to be understood. Besides, the language itself has multiple versions existing which makes it hard to determine which version to implement. At the start of this project, designing and coding began without enough research on the specification of language, and time was wasted. The rest of the project was reviewed and it had been seen that the same problem could appear multiple times especially in specifications of communication standard between agents, specifications of environment, protocols of communication between server and containers and communication between user interface and back-ends. Therefore, it was decided that a strict waterfall model is crucial in the development of these parts, and the plan was reviewed to capture this problem. Each procedure of development in each module was therefore listed in detail so that the order would be followed strictly. The failure of time management in the development of compiler is also showed in the updated Gantt chart.

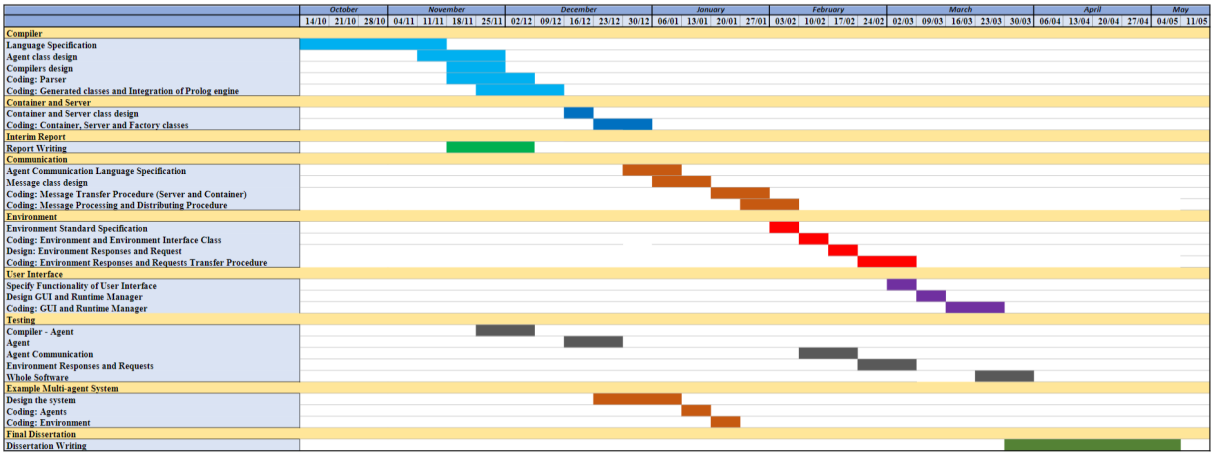

Figure 7: Gantt Chart

Besides the lesson learnt from the early project mismanagement, this project also introduced the area of agent programming to me which I was not even aware of its existence. Due to the complexity of this project and the effort I investigated into different parts, it helped me have better understanding of my own abilities. I can handle coding and implementing a reasonably large sized program on my own but

skills on other aspects such as technical writing and reading need lots of improvements.

8.2 Conclusion and Future Work

This project aims to implement an interpreter of 3APL agent programming languages. It has been seen that the implementation procedure for specific type of tasks can be significantly facilitated by using agent-oriented approach. In the example section, it is also shown that agent-oriented programming languages fit this kind of approach easily. The implemented interpreter successfully supports all the features proposed for 3APL language and therefore this project has achieved its aims and objectives successfully. This interpreter is also quite complete in many aspects, it can be used to construct a broad range type of multi-agent systems with reasonably complicated communications between agents. On top of that, the highly customizable implementation of environments can also easily adept to various kinds of tasks. This interpreter can also be used to display the messages being sent and behaviors of agents in runtime which can facilitate the development of multi-agent systems.

However, there are still many aspects can be improved or added. First, the compiler of this interpreter is highly depends on existing implementation of Prolog (namely tuProlog in this project), which requires duplicated procedure of parsing while providing multiple functionalities that are not used. It would improve the performance of this interpreter if an integrated Prolog implementation can be developed which stick to the specifications and conventions being made throughout other parts of the interpreter. Second, the parser/code generator of compiler can be refactored so that a complete checker can be implemented to check if the program is valid during compilation. Better error and exception generating and handling procedure can also be implemented to help programmers debugging the codes. Third, the service management system of containers can be improved including better syntax and potentially improved standard of communication so that agent programs involving messages sending and handling can be developed easier. One of the problem for current service management is that it requires a lot of repetitive coding work to properly delete services that has already been offered by agents.

9 Bibliography

[1] M. Wooldridge.An introduction to multi-agent systems. Wiley, 2002.

[2] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice.Knowledge engineering review, 10(2):115–152, 1995.

[3] Shoham, Y. (1990). Agent-Oriented Programming (Technical Report STAN-CS-90-1335). Stanford University: Computer Science Department.

[4] D.C. Dennett, The Intentional Stance (MIT Press, Cambridge, MA, 1987).

[5] J. McCarthy, Ascribing mental qualities to machines, Tech. Rept. Memo 326, Stanford AI Lab, Stanford, CA (1979).

[6] Multiagent Systems Katia P. Sycara AI magazine Volume 19, No.2 Intelligent Agents Summer 1998.

[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(4):357–401, 1999.

[8] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent Programming with Declarative Goals. In Cristiano Castelfranchi and Yves Lespérance,editors,Intelligent Agents VII Agent Theories Architectures and Languages, volume 1986 of Lecture Notes in Computer Science, pages 228–243. Springer, 2000. ISBN 3-540-42422-9.

[9] Mehdi Dastani. 2APL: a practical agent programming language.Autonomous Agents andMulti-Agent Systems, 16(3):214–248, 2008.

[10] Anand S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language.In Walter Van de Velde and John W. Perram, editors,MAAMAW, volume 1038 ofLectureNotes in Computer Science, pages 42–55. Springer, 1996. ISBN 3-540-60852-4.

[11] R.H. Bordini, J.F. Hübner, and M. Wooldridge.Programming multi-agent systems in AgentS-peak using Jason (Wiley Series in Agent Technology), 2007.

[12] Anand S. Rao and Michael P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors,KR, pages473–484. Morgan Kaufmann, 1991. ISBN 1-55860-165-1.

[13] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A Programming Language for Cognitive Agents Goal Directed 3APL. In Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors,PROMAS, volume 3067 ofLectureNotes in Computer Science, pages 111–130. Springer, 2003b. ISBN 3-540-22180-8.

[14] Eric ten Hoeve, Mehdi Dastani, Frank Dignum, John-Jules Meyer. 3APL Platform. Proceedings of the The 15th Belgian-Dutch Conference on Artificial Intelligence, 2003.

[15] Student theses on 3APL can be found at: `http://www.cs.uu.nl/3apl/studentthesis.html`

[16] Enrico Dent, tuProlog Manual, Alma Mater Studiorum–Universit'a di Bologna, Italy, 2015, available at: `http://amsacta.unibo.it/5450/7/tuprolog-guide.pdf`

[17] Omicini, A. Ricci, A. Viroli, M. Auton. (2008). Artifacts in the A&A meta-model for multi-agent systems. Agent Multi-Agent Systems, Vol 17, Issue 3, pp 432–456

[18] Behrens T. et al. (2012). An Interface for Agent-Environment Interaction. In: Collier R., Dix J., Novák P. (eds) Programming Multi-Agent Systems. ProMAS 2010. Lecture Notes in Computer Science, vol 6599. Springer, Berlin, Heidelberg

[19] Foundation for Intelligent Physical Agents. (2002). FIPA ACL Message Structure Specification, doc no.SC00061G, available at: `http://www.fipa.org/specs/fipa00061/SC00061G.html`

[20] JavaCC is available at: `https://javacc.org/`

[21] tuProlog is available at: `http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome`

A 3APL Syntax

Basic Components
To introduce syntax of 3APL, we first take a look at the basic components in it. Since a Prolog engine will be used to perform logic operations in this interpreter, we unify our term with normal Prolog terms for clearness.

- A **term** is any valid data structure in Prolog, it is the most general type in Prolog.
- An **atom** in Prolog is any valid sequence of characters which is used to represent value, variable and predicate.
- A **variable** is a sequence of characters that starts with a capital letter, note that underline is also considered to be a capital letter.
- A **string/name value** is a sequence of characters which starts with a lower case letter or wrapped by quote mark.
- A **number value** could be an integer or a floating point number.
- A **predicate** is a functor which can be performed on multiple elements (values or variables). It is represented by a sequence of letters that start with a lower case letter. Each **predicate** can take zero, one or multiple arguments.
- Now we can start to introduce new terms for convenience:
- We first introduce **predClause** which is a valid Prolog clause constructed by a predicate and its arguments.
- **ground predClause** is also defined to represent the predClause only contains values (not variables) as its arguments.
- **wff** is the well-formed formula constructed by predClauses.
- Prolog has also implemented **Horn clause** which defines the relationship of implication, therefore, we allow programmers to use it in 3APL. For example, $(a :- b)$ means b implies a, where a is a predicate term and b is also a term.

We can represent the syntax as:

$$
\begin{aligned}
val &::= num \mid str \\
atom &::= val \mid var \\
predClause &::= pred(atom^*) \\
gpredClause &::= pred(val^*) \\
hClause &::= predClause \ :- \ predClause^* \\
wff &::= predClause \ and \ predClause \mid predClause \ or \ predClause \mid not \ predClause
\end{aligned}
$$

Goal
In 3APL, goals will be used by the agent to check against beliefs. Even though they can contain variables, however, it could be confusing to define the scope of variables. Therefore, they are made to be deterministic and defined using ground predicate terms. Moreover, we implement the feature that guarantees two goals should be achieved simultaneously which is represented by: *gpredClause and gpredClause*.

$$goal ::= gpredClause \mid gpredClause \ and \ gpredClause$$

Belief

For beliefs, since they are used to define the knowledge that agents have, they should be represented as ground predicate terms. Also, we utilize the power of Horn clauses, they are also used to define beliefs.

$$bel \quad =:: \quad gpredClause \mid hClause$$

Capability
Capabilities are constructed by three components: signature, pre-conditions and post-conditions. Pre-condition is a **wff**, which will be used to check against the belief base of agent if the capability is allowed to perform. The signature contains name and arguments of the capability. Post-conditions determines what knowledge to add to or delete from the belief base, it is represented by a serial of predClause with a special notation *'not'* suggesting that the clause will be deleted from belief.

$$cap \quad =:: \quad \{wff\}\, name(atom^*)\, \{(predClause \mid not\ predClause)^*\}$$

Plan
A plan is considered to be a sequence of actions that need to be performed in order to achieve a goal. There are two type of actions, the first one would be **basic action** which is constructed by a single capability calling or special operation. The other one is called **sequential action** which represent a sequence of basic actions. The **sequential action** could be looping or conditional. The special operations include test action, message sending, environment action and abstract action which is a place holder and will be replaced by the plan revision rule.

$$\begin{aligned} bPlan \quad &=:: \quad capname(atom^*) \mid Send(x) \mid absPlan \mid wff? \\ comPlan \quad &=:: \quad bPlan \mid bPlan;\ comPlan \mid while(wff)\ comPlan \mid if(wff)\ comPlan\ else\ comPlan \end{aligned}$$

Goal Planning rule
Goal planning rules contain a query which is used to check against belief base, a goal to achieve and a plan to add to plan base of agent.

$$goalR \quad ::= \quad goal \quad <- \quad wff \mid comPlan$$

Plan revision rule
Plan revision rules contain a query which is used to check against belief base, an original plan to match and a new plan to replace the original one.

$$planR \quad ::= \quad comPlan \quad <- \quad wff \mid comPlan$$

Note: the full EBNF specification of 3APL can be found in Appendix C.

B 3APL Semantics

In this section, the semantics of 3APL is separated into two parts. A set of transition rules which are used to describe the change of agent's internal status during deliberating. An operational semantics will be used to describe how each parts of the agent are constructed and how they operate in more details.

B.1 Transition Rule of 3APL

Configuration of Agent:

The configuration shows the runtime status of agents, therefore, we introduce new component which is called *phase* to indicate the current deliberation phase the agent is in. Moreover, due to the fact that *goal planning rule base* and *plan revision rule base* can not be changed in runtime, to simplify the configuration, they are not included.

The configuration of agents is defined to be: $< bb, gb, pb, ph >$, where:

- bb represents *belief base*.
- gb represents *goal base*.
- pb represents *plan base*.
- ph represents *deliberation phase* which could be {*MessageHandling, GoalPlanningApply, PlanRevisionApply, PlanExecute* }.

Transition Rules:

After configuration is defined, it will be used to define the transition rules of the agent in deliberation cycle which indicate the changes of agent status after a certain phase of deliberation.

- *Check Message:* if messages are received, they will be added to the belief base.

$$< bb, gb, pb, MessageHandling > \rightarrow < bb, gb, pb, GoalPlanningApply > \text{ (No message received)}$$
$$< bb, gb, pb, MessageHandling > \rightarrow < bb', gb, pb, GoalPlanningApply > \text{ (New message received)}$$

- *Apply Goal Planning:* when certain conditions are satisfied by the *belief base* and that the goals associated with rule are not achieved, new plan will be added to the *plan base* and the goals will be removed temporarily from the *goal base*.

$$< bb, gb, pb, GoalPlanningApply > \rightarrow < bb, gb, pb, PlanRevisionApply > \text{ (No rule valid)}$$
$$< bb, gb, pb, GoalPlanningApply > \rightarrow < bb, gb', pb', PlanRevisionApply > \text{ (Rule applied)}$$

- *Apply Plan Revision:* when certain conditions are satisfied by the *belief base* and that some plans match the pattern in rule, those plans will be selected and revised by the rule.

$$< bb, gb, pb, PlanRevisionApply > \rightarrow < bb, gb, pb, PlanExecute > \text{ (No rule valid)}$$
$$< bb, gb, pb, PlanRevisionApply > \rightarrow < bb, gb, pb', PlanExecute > \text{ (Rule applied)}$$

- *Execute Plan:* if there are plans in *plan base*, one of them will be executed for one step. However, the execution may be failed and no changes should be made in this case.

$$< bb, gb, pb, PlanExecute > \rightarrow < bb, gb, pb, MessageHandling > \text{ (No plan executed successfully)}$$
$$< bb, gb, pb, PlanExecute > \rightarrow < bb', gb', pb', MessageHandling > \text{ (Plan executed successfully)}$$

B.2 Operational Semantics

After knowing how the language looks like, we now consider the semantics of it.

Basic Components
As we discussed before, 3APL makes use of first order logic language, which is Prolog in this project, therefore, the basic components follows the same rules as in Prolog.

In 3APL, the atoms and predicate functor are always constructed as a **predicate clause**, except in **capabilities** and **basic plans** where atoms are used for representing variables and values to assign to them.

Therefore, we can illustrate the behavior of these components by only showing how **predicate clauses** is executed.

If the **predicate clause** is ground by itself or is grounded by the environment, it is viewed as a normal boolean expression which returns boolean value and has no side effects.

If the **predicate clause** is not ground by itself and can't be ground by the environment, the variables in it will be unified. If unification is successful, the variable will be saved into the environment according to its scope and the clause itself will return boolean value *false*, otherwise, it returns *true* and no side effects will be performed.

Belief
Beliefs are the knowledge that agents have. They will be used in most of execution agents performed. There are two ways of changing the beliefs an agent hold, the messages received from the environment or other agents and the capabilities the agent performed.

Goal
Goals are the objectives that agents try to achieve. They are programmed by the programmer and there isn't a way to add new goals other than receiving messages from the environment or other agents. However, when goals are achieved, they will be removed from the agent's goal base.

There are two ways to achieve goals. When the agent's belief base is changed, the goals will be checked against the new belief base and removed accordingly. When a goal planning rule is performed, the goal and the plan will be associated, and when the plan is completed, the goal will be removed.

As we have showed in the *Syntax* section, goal can be constructed from multiple sub-goals in a form: *g1 and g2 and g3....* In this case, when each sub-goal is achieved, the rest of sub-goals will be checked, and the only way to remove the main goal is that all sub-goals are achieved. (Note that there exist situations where a sub-goal was achieved before but failed because of change of beliefs afterward)

Capability
As we discussed before, each capability is constructed by three parts, pre-condition, signature and post-condition.

When defining a capability, the signature contains the variables which are used in the conditions. And the post-condition should be able to be grounded by variables in signature and the unification of pre-condition, which means that all variables in post-condition should have already be seen in the pre-condition and signature.

When applying a capability in a plan, all variables in the signature will be assign to values according to the environment. Those values will be used to replace the according variables in conditions of the capability. Afterwards, the pre-condition is unified and if it success, the assignments of all variables will be used by post-condition to change agent's belief base, otherwise, no action is taken.

Note that the deletion of beliefs will be performed prior to the addition of them to prevent the case that result in deleting the added new beliefs.

Plan

Plan can also be considered as a **sequence action** which is simply an action that contains a sequence of other actions that need to be executed in order. In this part, the actions that can construct a plan will be introduced. Different type of actions will be introduced separately, but we will start with the two main types, **basic action** and **sequential action**.

- A **basic action** can be considered as a single step that agent execute except the **abstract action**. The **basic action**s are used to construct the sequential actions.
- A **sequential action** represents all actions that need to be executed for more than one steps.
- A **capability action** will call the capability with the same name when executed. The variables in the capability calling must be grounded by the environment.
- A **send action** is used to send messages to other agents or containers, the specification of messages is made in implementation section.
- A **environment action** is used to perform actions in the environment. However, instead of taking the action directly, this action will send an action request to the environment and wait for its respond. It is notable that after the request is sent, the whole plan will be suspended until a respond is received. The specification of environment action requests in implementation section.
- A **test action** can be considered as a guard within the action. When the guard holds *false* the action is stopped from further execution and this guard will be checked again during the next time this action is selected to execute. Unassigned variables are allowed a textbftest action, such variables will be unified and added to the environment.
- An **abstract action** can't not be executed, therefore the action starts with an abstract action can't be selected for execution. action revision rules will be applied to remove **abstract actions** and replace with concrete actions in runtime. *Note: any action that doesn't start with an abstract action is a concrete action.*
- A **sequence action** is a sequence of actions which that start with a **basic action**. They will be executed in order.
- A **conditional action** follows the normal *if-then-else* pattern. It consists a **wff** which is used to check for condition and two **sequence action**s as *then clause* and *else clause*. The boolean expression **wff** is allowed to contain unassigned variables, such variables will be unified and add to the environment if successful.
- A **looping action** follows the normal *while-do* pattern. It consists a **wff** for condition and a **sequence plan** as loop body. Again, the condition is allowed to have unassigned variables. However, the unassigned variables in condition will be unified instead of memory the unification of last loop every loop. More details of variable scoping will be provided in the next section.

Goal planning rule

A **goal planning rule** is constructed by a **wff** guard, a **goal** as its head and a **plan**. There are two conditions that a **goal planning rule** can be executed:

- First, the **goal** in its head exists in the **goal base** of the agent. Same as the definition of **goal**, we are allowing sequential **goal** (*g1 and g2 and g3 ...*), each sub-goals should exist in **goal base** when such **goal** is used. However, there is a different between the head and normal **goal** that the head **goal** allows variables which will be unified against the **goal base**.
- Second, the **wff** guard is satisfied by the **belief base** and its variables can be unified.

Once a **goal planning rule** is executed, its plan will be added to the **plan base** of agent for execution, the plan will keep unified variables in the rule's head and guard in its environment. And the goal will be associated with such plan.

Plan revision rule
A textbfplan revision rule contains a head which is a **plan**, a **wff** guard and a new **plan**. There are two conditions that a **plan revision rule** can be executed:

- First, the head **plan** exists in the **plan base**, the matching is performed on the **plan**s without considering any context which means that the clauses are required to have the same variable names and no assignments are made.
- Second, the guard **wff** can be unified by the **belief base** as in the **goal planning rules**.

Once a **plan revision rule** is executed, the old **plan** (matched by the head) will be replaced by the new **plan**. The new **plan** will inherit the environment of old **plan** and the unification of variables in guard **wff** will be added to that environment.

B.3 Scope of variables

During the previous sections, it is mentioned that there are multiple components that allows variables. Such components will be listed and explained.

- **Horn clauses** in **belief base**: the scope here is the same as the scopes in Prolog. The scope of a variable is the rest of the clause as soon as it first appears.
- **Capability**: the scope of variables in the signature is the entire clause of capability. After the variables in signature is assigned, during the execution of pre-condition, the variables in it have the scope of the rest of pre-condition and the entire post-condition. In post-condition, variables have the scope of the rest of post-condition.
- **Test action**: the scope of variables in a **test action** is rest of its **parent plan**.
- **Conditional plan**: the scope of variables in the **wff** conditional expression is the *then clause* in the plan.
- **Looping plan**: this is a special case that the conditional expression is checked and unified in every loop instead of only once when execution entire the loop for the first time. In such a way, variables are allowed to change their values during each loop and therefore more flexible. Such behaviour means that the scope of variables in the **wff** conditional expression is the looping body of the plan until the next iteration.
- **Goal planning rule**: the scope of variables in the head **goal** is the entire rule. The scope of variables in the **wff** guard is the **plan** of the rule.
- **Plan revision rule**: the variables in the **wff** will overwrite or be added to the environment of previous **plan** (the head) The new environment will be inherited by the new **plan**.

C EBNF of 3APL

⟨*Program*⟩ ::= **Name:** ⟨*Ident*⟩
Belief Base: (⟨*beliefs*⟩)?
Goal Base: (⟨*goals*⟩)?
Capabilities: (⟨*capabilities*⟩)?
Plan Base: (⟨*plan*⟩)?
Goal Planning Rules: (⟨*Grules*⟩)?
Plan Revision Rules: (⟨*Prules*⟩)?

⟨*capabilities*⟩ ::= (⟨*capability*⟩)*

⟨*capability*⟩ ::= `{` ⟨*query*⟩ `}` ⟨*vpredClause*⟩ `{` ⟨*literals*⟩ `}`

⟨*beliefs*⟩ ::= (⟨*belief*⟩)*

⟨*belief*⟩ ::= ⟨*gpredClause*⟩ `.` | ⟨*vpredClause*⟩ `:-` ⟨*pliterals*⟩ `.`

⟨*goals*⟩ ::= ⟨*goal*⟩ (`,` ⟨*goal*⟩)*

⟨*goal*⟩ ::= ⟨*gpredClause*⟩ (`and` ⟨*gpredClause*⟩)*

⟨*plans*⟩ ::= (`{` ⟨*plan*⟩ `}`)*

⟨*plan*⟩ ::= (⟨*basicplan*⟩)*

⟨*basicplan*⟩ ::= **if** (⟨*query*⟩) **then** `{` ⟨*plan*⟩ `}` (**else** `{` ⟨*plan*⟩ `}`)?
| **while** (⟨*query*⟩) **do** `{` ⟨*plan*⟩ `}`
| ⟨*basicaction*⟩ `;`

⟨*basicaction*⟩ ::= ⟨*vpredClause*⟩ | **Send**((⟨*atom*⟩ | ⟨*vpredClause*⟩)*)
| **Env**((⟨*atom*⟩)*) | ⟨*query*⟩`?`

⟨*Prules*⟩ ::= (⟨*Prule*⟩)*

⟨*Prule*⟩ ::= `{` (⟨*goal*⟩) `}` `<-` ⟨*query*⟩ `'|'` `{` ⟨*plan*⟩ `}`

⟨*Grules*⟩ ::= ⟨*Grule*⟩ (`;` ⟨*Grule*⟩)*

⟨*Grule*⟩ ::= `{`⟨*plan*⟩ `}` `<-` ⟨*query*⟩ `'|'` `{`⟨*plan*⟩ `}`

⟨*literals*⟩ ::= ⟨*literal*⟩ (`,` ⟨*literal*⟩)*

⟨*pliterals*⟩ ::= ⟨*literal*⟩ ((`,` | `;`) ⟨*literal*⟩)*

⟨*literal*⟩ ::= ⟨*vpredClause*⟩ | (**not** ⟨*vpredClause*⟩)*

⟨*pliteral*⟩ ::= ⟨*vpredClause*⟩ | (`\+` ⟨*vpredClause*⟩)*

⟨*wff*⟩ ::= ⟨*literal*⟩ | ⟨*wff*⟩ **and** ⟨*wff*⟩ | ⟨*wff*⟩ **or** ⟨*wff*⟩

⟨*query*⟩ ::= ⟨*wff*⟩ | **true**

⟨*gpredClause*⟩ ::= ⟨*val*⟩ `(` ⟨*val*⟩? (`,` ⟨*val*⟩)* `)`

⟨*vpredClause*⟩ ::= ⟨*val*⟩ `(` ⟨*atom*⟩? (`,` ⟨*atom*⟩)* `)`

⟨*atom*⟩ ::= ⟨*val*⟩ | ⟨*var*⟩

⟨*val*⟩ ::= ([`a-z`])+ ([`A-Z`] | [`a-z`] | `_` | [`0 - 9`])* | `'` (⟨*char*⟩)* `'` | `[` ⟨*val*⟩ (`,` ⟨*val*⟩)*`]`

⟨*var*⟩ ::= ([`A-Z`] | `_`) ([`A-Z`] | [`a-z`] | `_` | [`0 - 9`])* | `[` ⟨*atom*⟩ (`,` ⟨*atom*⟩)*`]`

D Figures

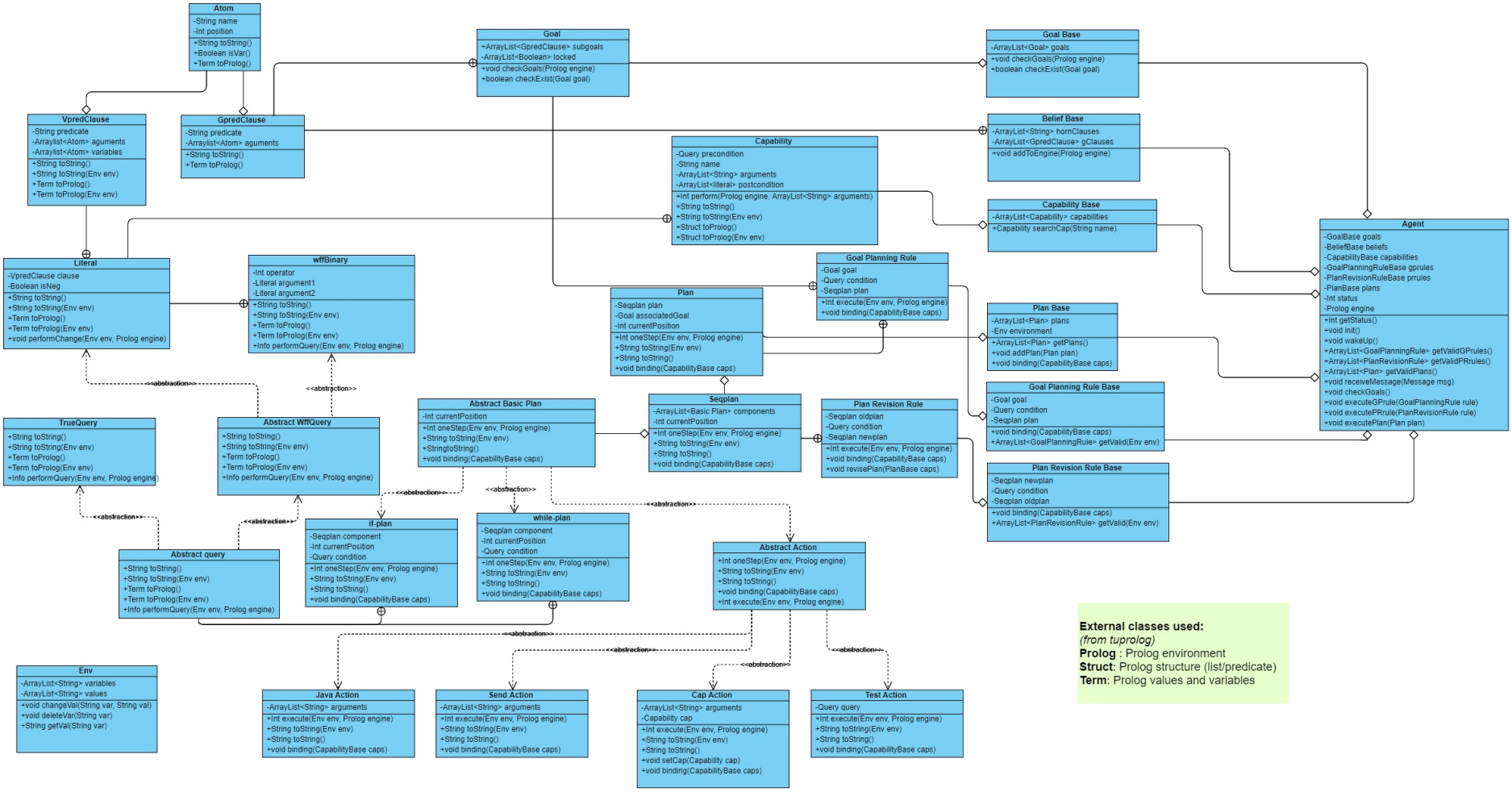

Figure D.1: Full Java Class Diagram

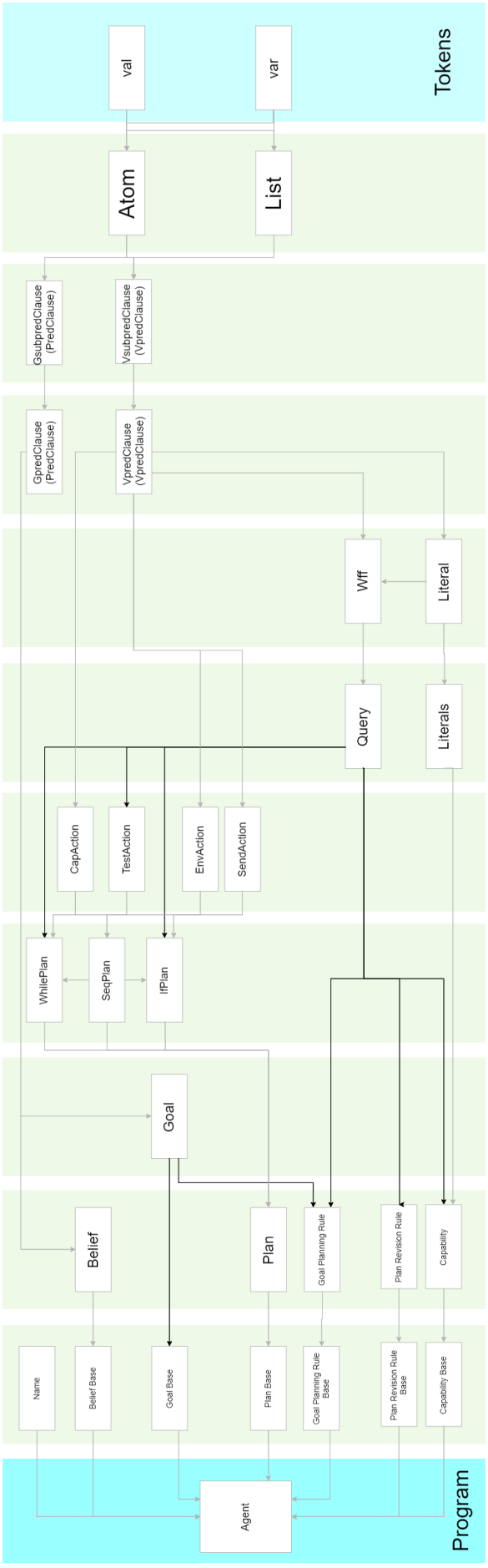

Figure D.2: Compiler Architecture (abstract syntax tree)

E 3APL Example Codes

Seller

```
Name: Seller
Belief Base:
    money(0).
    price(pen,2).
    price(pencil,1).
    storage(pen,2).
    storage(pencil,3).
Goal Base:
    sell()
Capability Base:
    {price(Item,Price) and storage(Item,Count) and sub(Count,1,NCount)
     and money(Money) and sum(Money, Price, NMoney) and not eql(Count,0)}
    sell(Item,Buyer)
    {not money(Money), not storage(Item,Count),
     not received(query, Buyer, buy(Item), channel(1)),
     money(NMoney), storage(Item,NCount)}

    {received(query, Buyer, buy(Item), channel(1))}
    deleteOrder(Item)
    {not received(query, Buyer, buy(Item), channel(1))}

    {received(query, Buyer, price(), channel(1))}
    receivePriceRequest(Buyer)
    {not received(query, Buyer, price(), channel(1))}
Plan Base:
{
    Send(inform, ams, is(seller),channel(1));
}

Goal Planning Rule Base:

<- received(query, Buyer, price(), channel(1)) |
{
    receivePriceRequest(Buyer);
    Send(inform, Buyer, price(pen,2),channel(1));
    Send(inform, Buyer, price(pencil,1),channel(1));
}

<- received(query, Buyer, buy(Item), channel(1)) |
{
    sell(Item, Buyer);
    Send(inform, Buyer, sell(Item), channel(1));
}

Plan Revision Rule Base:
{
    sell(Item, Buyer);
    Send(inform, Buyer, sell(Item), channel(1));
}
<- storage(Item,0) |
{
    deleteOrder(Item);
    Send(inform, Buyer, no(Item), channel(1));
}
```

Buyer

```
Name: Buyer
Belief Base:
    pocket(20).
    holding(0).
    enough(Item) :- price(Item, Price, Seller), pocket(Pocket),
                    geq(Pocket, Price).
Goal Base:
    buy()
Capability Base:
    { received(inform, Seller, price(Item, X), channel(1)) }
    getPrice()
    { not received(inform, Seller, price(Item, X),
      channel(1)), price(Item, X, Seller) }

    {true}
    createStock(Item)
    {stock(Item,0)}

    {price(X,A,Seller) and pocket(Pocket) and holding(Holding)
     and sub(Pocket, A, NPocket) and sum(Holding,A,NHolding)
     and geq(NPocket, 0)}
    buy(X,Seller)
    {not pocket(Pocket), not holding(Holding),
     pocket(NPocket), holding(NHolding)}

    {received(inform, Seller, sell(X), channel(1))
     and holding(Holding) and price(X, Price,Seller)
     and sub(Holding, Price, NHolding)
     and stock(X,Count) and sum(Count, 1, NCount)}
    pay(X,Seller)
    {not received(inform, Seller, sell(X), channel(1)),
     not holding(Holding), not stock(X,Count),
     holding(NHolding), stock(X,NCount)}

    {true}
    cleanMessage(Seller)
    {not received(inform, Seller, is(seller),channel(1))}

    {price(X,A,Seller) and pocket(Pocket) and holding(Holding)
     and sum(Pocket, A, NPocket) and sub(Holding,A,NHolding)}
    putBack(X,Seller)
    {not pocket(Pocket), not holding(Holding),
     pocket(NPocket), holding(NHolding)}

Plan Base:
    {
        while(not received(inform, Seller, is(seller), channel(1))){
            Send(query, ams, is(seller), channel(1));
        };
        received(inform, Seller, is(seller), channel(1))?;
        Send(query, Seller, price(), channel(1));
        cleanMessage(Seller);
    }
```

```
Goal Planning Rule Base:
    <- true |
    {
        getPrice();
        if(price(Item,Price,Seller) and not stock(Item,Amount)){
            createStock(Item);
        }else{
            true?;
        };
    }

    <- price(Item, X, Seller)
       and not received(inform, Seller, no(Item), channel(1))
       and enough(Item) |
    {
        buy(Item,Seller);
        Send(query, Seller, buy(Item), channel(1));
        pay(Item,Seller);
    }
Plan Revision Rule Base:
    {
        pay(Item,Seller);
    }
    <- received(inform, Seller, no(Item), channel(1)) |
    {
        putBack(Item,Seller);
    }
```

Cleaner

```
Name: Cleaner
Belief Base:
    takeBins(bins(A)) :- findall(position(X,Y,attribute(Type,Task)),
          (received_env(position(X,Y,bin,attribute(Type,Task)))),A).

    takeStations(stations(A)) :- findall(position(X,Y,Atr),
          received_env(position(X,Y,station,Atr)), A).

    takeRecharges(recharges(A)) :- findall(position(X,Y,Atr),
          received_env(position(X,Y,recharge,Atr)), A).

    takeSelf(self(X,Y,A)) :- received_env(position(X,Y,self,A)).

    maximum(X,Y,X) :- geq(X,Y).
    maximum(X,Y,Y) :- grt(Y,X).

    minimum(X,Y,Y) :- geq(X,Y).
    minimum(X,Y,X) :- grt(Y,X).

    maxDist((X1,Y1,Dist1),(X2,Y2,Dist2),(X1,Y1,Dist1)) :-
                                                geq(Dist1,Dist2).
    maxDist((X1,Y1,Dist1),(X2,Y2,Dist2),(X2,Y2,Dist2)) :-
                                                grt(Dist2,Dist1).

    minDist((X1,Y1,Dist1),(X2,Y2,Dist2),(X2,Y2,Dist2)) :-
                                                geq(Dist1,Dist2).
    minDist((X1,Y1,Dist1),(X2,Y2,Dist2),(X1,Y1,Dist1)) :-
                                                grt(Dist2,Dist1).

    dist(X1,X2,Y1,Y2,Dist) :- sub(X1,X2,X3), sub(Y1,Y2,Y3),
                              abs(X3,X), abs(Y3,Y),
                              maximum(X,Y,Dist).

    getScore(position(X,Y,attribute(Type)), score(X,Y,Score)) :-
                    (self(Xs,Ys,attribute(none, SCarry, SBattery));
                    self(Xs,Ys,attribute(Type, SCarry, SBattery))),
                    dist(X,Xs,Y,Ys,Dist), sum(SCarry, 0.0, Carry), \=(Dist,0),
                    root(Dist,0.5,DDist) ,div(Carry,DDist,Score).
    getScore(position(X,Y,attribute(Type,Task)), score(X,Y,Score)) :-
                    (self(Xs,Ys,attribute(none, SCarry, SBattery));
                    self(Xs,Ys,attribute(Type, SCarry, SBattery))),
                    dist(X,Xs,Y,Ys,Dist), sum(Task,0.0,Task1),\=(SCarry,200),
                    sub(200.0,SCarry,Left), minimum(Left,Task1,Take), \=(Dist,0),
                    root(Dist,0.5,DDist), div(Take,DDist,Score).

    getScores(Result) :- bins(A), stations(B),
                         append(A,B,List), scoresHelper(List,Result).
    scoresHelper([],[]).
    scoresHelper([A|List], [(X,Y,Score)|Result]) :-
                                    getScore(A,score(X,Y,Score)),
                                    scoresHelper(List,Result).
    scoresHelper([A|List], Result):- \+ getScore(A,B),
                                     scoresHelper(List,Result).

    rechargeDist(X,Y,[],[]).
    rechargeDist(X,Y,[position(X1,Y1,Atr)|List],[(X1,Y1,Dist)|Result]) :-
                            dist(X1,Y1,X,Y,Dist), rechargeDist(X,Y,List, Result).
```

```
    rechargeDist(Result) :- self(X,Y,Atr), recharges(List),
                            rechargeDist(X,Y,List, Result).

    mini([(X,Y,Min)], X, Y, Min).
    mini([(X1,Y1,Dist1),(X2,Y2,Dist2)|Rest],X3,Y3,Dist3) :-
                            mini([(X2,Y2,Dist2)|Rest],X4,Y4,Dist4),
                            minDist((X1,Y1,Dist1),(X4,Y4,Dist4),(X3,Y3,Dist3)).

    maxi([(X,Y,Max)], X, Y, Max).
    maxi([(X1,Y1,Dist1),(X2,Y2,Dist2)|Rest],X3,Y3,Dist3) :-
                            maxi([(X2,Y2,Dist2)|Rest],X4,Y4,Dist4),
                            maxDist((X1,Y1,Dist1),(X4,Y4,Dist4),(X3,Y3,Dist3)).

    maxScore(X,Y,Max) :- getScores(Result), maxi(Result,X,Y,Max).
    minRecharge(X,Y,Min) :- rechargeDist(Result), mini(Result, X, Y, Min).

    goto(X,Y,left):- self(Xs,Ys,Atr), grt(Xs,X), eql(Ys,Y).
    goto(X,Y,right):- self(Xs,Ys,Atr), grt(X,Xs), eql(Ys,Y).
    goto(X,Y,up):- self(Xs,Ys,Atr), grt(Y,Ys), eql(Xs,X).
    goto(X,Y,down):- self(Xs,Ys,Atr), grt(Ys,Y), eql(Xs,X).
    goto(X,Y,upleft):- self(Xs,Ys,Atr), grt(Y,Ys), grt(Xs,X).
    goto(X,Y,upright):- self(Xs,Ys,Atr), grt(Y,Ys), grt(X,Xs).
    goto(X,Y,downleft):- self(Xs,Ys,Atr), grt(Ys,Y), grt(Xs,X).
    goto(X,Y,downright):- self(Xs,Ys,Atr), grt(Ys,Y), grt(X,Xs).



Goal Base:
    clean()

Capability Base:
    {takeBins(bins(A)) and takeStations(stations(B))
     and takeRecharges(recharges(C)) and takeSelf(self(X,Y,Atr))}
    saveView()
    {not received_env(E), not bins(Pa), not stations(Pb), not recharges(Pc),
     not self(Px,Py,Patr), bins(A), stations(B), recharges(C), self(X,Y,Atr)}

    {true}
    start()
    {moving()}

    {true}
    finish()
    {not moving()}

Plan Base:
    {
        Env(init);
        saveView();
    }


Goal Planning Rule Base:

    <- not moving() and self(Xs,Ys,attribute(A,B,Battery))
       and (not maxScore(X,Y,Max) or (not minRecharge(Xr,Yr,D)
       and leq(Battery, 150))) |
    {
```

```
        start();
        Env(move, upleft);
        saveView();
        finish();
    }

    <- not moving() and maxScore(X,Y,Max) |
    {
        start();
        while(goto(X,Y,Direction)) {
            Env(move, Direction);
            saveView();
        };
        if(bins(A) and member(position(X,Y,Atr),A)){
            Env(pickup,200);
        }else{
            Env(throw);
        };
        saveView();
        finish();
    }

Plan Revision Rule Base:
{
    while(goto(X,Y,Direction)) {
        Env(move, Direction);
        saveView();
    };
    if(bins(A) and member(position(X,Y,Atr),A)){
        Env(pickup,200);
    }else{
        Env(throw);
    };
}
<- self(Xs,Ys,attribute(A,B,Battery))
   and minRecharge(Xr,Yr,D) and geq(D,Battery) |
{
    while(goto(Xr,Yr,Direction)) {
        Env(move, Direction);
        saveView();
    };
    Env(recharge);
}
```