3APL Platform

User Guide

Mehdi Dastani

16th November 2006

Contents

1 3APL Platform

The 3APL platform is an experimental tool, designed to support the development, implementation, and execution of 3APL agents [2]. It provides a graphical interface through which a user can develop and execute 3APL agents using several facilities, such as a syntax-colored editor and several debugging tools. The platform allows communication among agents. The platform can run on several machines connected in a network such that agents hosted on 3APL platforms can communicate with each other. More detailed information on the 3APL platform can be found in [5].

2 Software Requirements

3APL platform has been tested on Windows 98, Windows NT and Windows XP, as well as Linux and Unix (Solaris). 3APL is written in Java 2 JDK 1_5_0_06, and makes use of the Prolog engine of JIProlog, which is also written in java. The 3APL package consists of a .jar file that contains all the class files needed, as well as some examples of 3APL programs. The package needs approximately 850 KB.

3 Installation

In order to install 3APL Platform you need to follow the next procedure:

- Download the `3apl.zip` file from the following URL:

  `http://www.cs.uu.nl/3apl/download.html`

- Extract the contents of the ZIP-file into a directory. The content consists of a `3apl.jar` file and a directory called `example`. This directory contains a number of examples that will be discussed and explained in section 5 of this user guide.
- To start the 3APL platform on **Windows**, double click the file `3apl.jar`, or alternatively, type `java -jar 3apl.jar` in a Command Prompt window at the 3apl directory.
- To start the 3APL platform in **Mac OS X**, double click the file `3apl.jar`.
- To start the 3APL platform in **Unix** or **Linux**, type `java -jar 3apl.jar` to a prompt in the 3apl directory.

The implementation documentation of the 3APL platform can be found at:

`http://www.cs.uu.nl/3apl/docs/aplp-refman/index.html`

4 User Interface

When the 3APL platform is started, the user is presented with the window shown in figure 1. The user should select whether the application is intended to act as a server or as a client. The server option must be selected the first time the 3APL platform is run. If the user selects the server option, the application starts a new instance of the name server, which is a part of the JAS library. The client option can be selected only if the 3APL platform is running as a server already. When the user selects the client option, the IP of the server with which the (client) platform should connect must be filled in. Once the choice has been made the application continues and the user is presented with two windows. The first appearing window with the name `blockworld` is a shared environment in which agents can perform actions. This window will be discussed in details in section 5. The second window is the 3APL platform main window, shown in figure 2.

Figure 1: The startup dialog

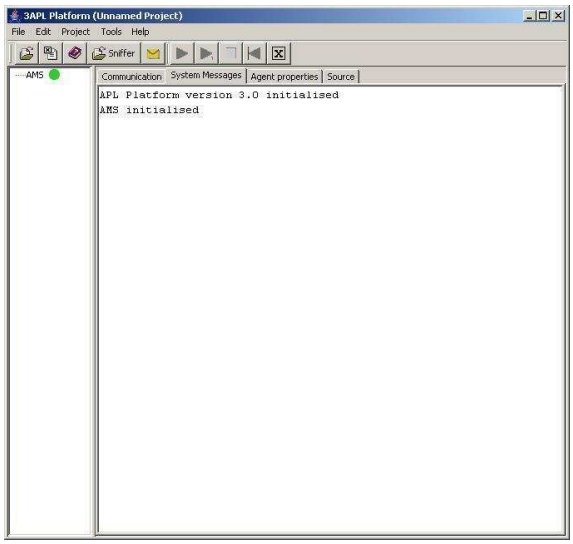

Figure 2: The main window

Left of the main window is a visual tree which represents the agents hosted on this platform. The leaves of the tree show the names of the agents that are running on the platform. The Agent Management System (AMS), which is responsible for registration of the hosted agents on the platform, is also listed. However, AMS is not a real agent, i.e. it cannot be controlled or inspected by the user. Initially, there are no agents on the platform (except the AMS). On the righthand side of the name of an agent is an icon displaying the status of this agent. The possible status of an agent and their representing icons are illustrated in figure 3.

Icon	*Meaning*
	`3APl program is compiled successfully and the agent is ready.`
	`Agent's deliberation process is started.`
	`Agent's deliberation process is stopped.`
	`Agent's deliberation process is ended.`
	`3APL program cannot be compiled because a syntax error has occurred.`

Figure 3: Status icons of agents.

An icon displaying the status of an agent may contain the description of the agent if the agent has reported its description to the AMS when it is executed. The agent can report its descriptions to AMS by sending an inform message to it. For instance, a seller agent may inform the AMS

that he is a seller by performing the following communication action (in the next sections, we will explain agent communication in more details):

```
Send(ams, inform, description(seller) );
```

The last icon in Figure 3 can only appear if the 3APL program of the agent is saved after a modification through which a syntax error is created. Note that the agent has already been loaded on the platform (see section 4.2 for modifying 3APL programs on the platform). The fact that the agent was loaded already implies that the program was correctly complied before modification. An erroneous agent program cannot be loaded on the 3APL platform. The System Messages tab will display the error message. This tab is generally used to display information about the platform.

4.1 Messages window

If an agent sends a message to another agent or to the AMS, the message will be displayed in the Communication tab. If the message is addressed to an agent which is also located on the platform, no HAP (Home Agent Platform) is displayed. The HAP is shown on the Communication tab whenever an agent receives a message from an agent on another platform, e.g. *seller*@10.0.0.12. The System Messages tab is shown in figure 4.

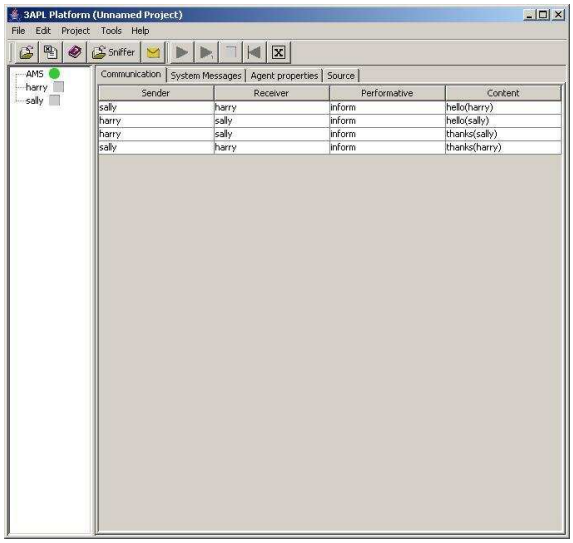

Figure 4: The Communication tab of the 3APL platform.

4.2 Agent properties

The user can select an agent by clicking on the agent's name (in the visual tree). The "Agent properties" tab shows the current belief base, goal base, plan base, capabilities, goal planning rules, and plan revision rules of the agent. If the agent is running, the inference log is filled with the reasoning steps. The reasoning step includes selection and applications of rules and selection and execution of plans. These steps form the so-called agent's deliberation cycle [3]. The Agent properties tab is shown in figure 5. On the left side of this panel, the control buttons for the selected agent are located. The functionalities of the control buttons are listed in figure 6.

If the user selects the first button, the agent's deliberation process commences. The second button is the same as the first except that the deliberation process commences only for one deliberation cycle. The user can select the third button to halt the deliberation process at any time. Editing 3APL programs on the platform can be done by selecting the fourth button. A syntax highlighting text editor is opened displaying the contents of the agent program. The user

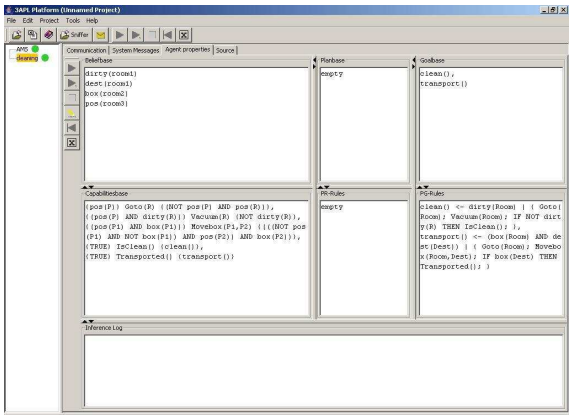

Figure 5: The Agent properties panel.

Icon	*Meaning*
	`Execute the selected agent.`
	`Execute the selected agent for one deliberation cycle.`
	`Stop the execution of the selected agent.`
	`Edit the 3APL program of the selected agent.`
	`Recompile the program of the selected agent.`
	`Remove the selected agent from the platform.`

Figure 6: Agent status icons.

can now change the program. The changes can then be saved and applied if the 3APL program is parsed correctly; otherwise a parse errors is generated. If the user wants to reset the configuration of the agent (i.e., its initial bases are set as specified by the agent program), the user can select the recompile button which is the fifth button. Finally, the sixth button closes the selected agent and removes it from the platform. Similar buttons on the top of the window have similar meaning for multi-agent cases (see section 5.3).

4.3 Sniffer

One of the tools, which can be activated to observe the communication between agents, is the Sniffer tool. It can be executed by clicking the `Sniffer` icon located at the top of the main window. This tool produces an output like displayed in figure 7.

This tool is used to visualize the conversations between the agents on the platform. On the left side of the sniffer window the loaded agents as well as the AMS are shown. Every message produces an arrow, with its origin starting at the position of the sender, and its head ending at the receiver of the message. A performative (speech acts such as inform and request [6]) is displayed on every arrow. If a user double clicks this performative, the content of the message appears in an additional message box, as illustrated in figure 8. This message box displays the contents of the message. We have chosen this form to avoid clutter on the sniffer output window.

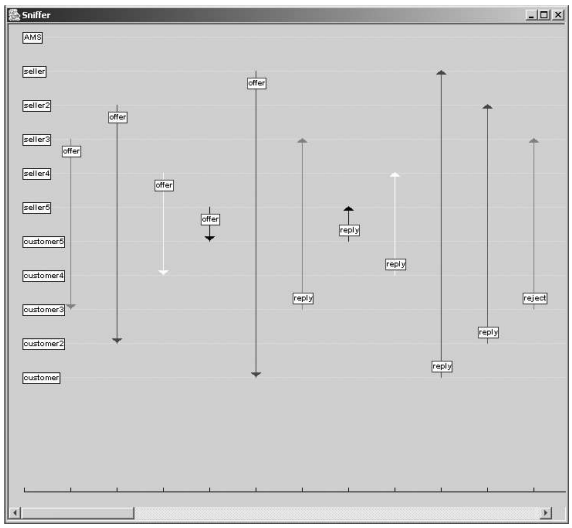

Figure 7: An example output of the Sniffer tool.

Figure 8: An example output of the message box.

4.4 Message sender

Another platform tool available to the user is the custom message sender. Using this tool, the user can send a message to an agent loaded on the platform. The custom message sender tool is illustrated in figure 9.

Figure 9: The message sender tool with example data entered.

All message parameters must be filled in by the user, i.e., the sender and the receiver of the message, which can be selected from a list of agents currently on the platform, the performative of the message, and the content of the message, which is a belief formula. When the user press on the `SEND` button, a confirmation dialog will be displayed if the message is successfully sent. The user can then send other messages or use the `CANCEL` button to close this tool. Whether a message sent to the agent will actually be handled depends on several factors. The most important one is if the agent is running. If it is not, then the message is stored in its message buffer until the agent becomes active again.

4.5 Template agents

If the user selects `Add New Agent` in the `Project` menu, a window is shown which is illustrated in figure 10.

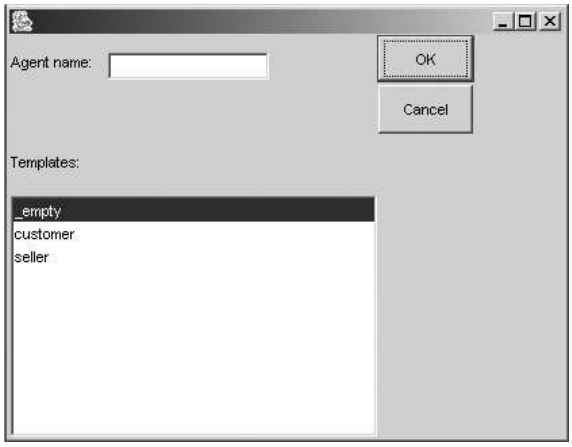

Figure 10: The "Add New Agent" dialog.

The list shown in the `Templates` window is taken from the *templates* directory. The templates are 3APL programs which implement agents with some common abilities, for example seller or customer. If the user has given a name for the new agent and has selected a template, the agent will be added to the platform with the chosen name. The agent program is a copy of the selected template program with the exception of the first instruction which is *program NAME* instruction. This has been filled with the name of the agent the user has given in the agent name field. Users can add their own templates. In order for this to work, the template program must be placed in the *templates* directory. A copy of the template will be created. The name of the new agent is taken from the `Agent name` field as shown above in figure 10. This name will replace every occurrence of "%1" in the program. This means that a template must have a first sentence such as: *program* "%1".

5 Getting Started

In this section we show by examples how the 3APL platform can be used. In particular, we discuss simple examples of 3APL programs to show the meaning of the control buttons, manipulations of the agent's belief base, goal base, and plan base. Also, we discuss an example of a multi-agent system where the use of communication action `Send`, the way to request services of other agents through the AMS (Agent Management System) agent, and the use of a so called *project file* are explained. Finally, we discuss how a shared environment for 3APL agents can be implemented using external Java programs. The 3APL agents can then execute actions, including sensing actions, in the shared environment.

It should be noted that we do not present the formal syntax and semantics of the 3APL programming language in this user guide. In fact, we assume that the reader is familiar with the 3APL programming language and its formal semantics. A detailed presentation of the 3APL programming language and its semantics can be read in [1]. More information on 3APL programming language can be found in [2–4].

5.1 Loading and Executing a Simple Agent

In this section we explain how to write a simple 3APL program.

5.1.1 Program Structure

An example of a simple 3APL program is shown in figure 11. As you can see, this program consists of five fields, i.e. `PROGRAM`, `CAPABILITIES`, `BELIEFBASE`, `GOALBASE`, `PLANBASE`, `PG-RULES` and `PR-RULES`. These words are 3APL dedicated instructions. The basic action names start with capital letters, goals and beliefs are prolog-like terms, and punctation are used in specific way. The BNF specification of 3APL programming language can be found at 3APL webpage:

`http://www.cs.uu.nl/3apl/bnf.html`

```
PROGRAM "cleaning"

CAPABILITIES{
    { pos(P) }              Goto(R)         { NOT pos(P) , pos(R) },
    { pos(P) AND dirty(R) } Vacuum(R)       { NOT dirty(R) },
    { pos(P1) AND box(P1)}  Movebox(P1,P2)  { NOT pos(P1), NOT box(P1), pos(P2), box(P2)},
    {TRUE}                  IsClean()       {clean()},
    {TRUE}                  Transported()   {transport()}
}

BELIEFBASE{
    dirty(room1).
    dest(room1).
    box(room2).
    pos(room3).
}

GOALBASE{ clean(), transport() }

PLANBASE{ }

PG-RULES{
     clean() <- dirty(Room) |
                {   Goto(Room);
                    Vacuum(Room);
                    if not dirty(R) then IsClean()
                },
     transport() <- box(Room) AND dest(Dest) |
                {   Goto(Room);
                    Movebox(Room,Dest);
                    if box(Dest) then Transported()
                }
}

PR-RULES{}
```

Figure 11: A simple 3APL program example.

belief base This contains the information of the agent including its general information about the world as well as specific information about its environment. Beliefs are stored as Prolog clauses. For example `pos(room1)` can be used to indicate that the agent believes its current position is `room1`. The belief base can be extended with a Prolog program using the `LOAD` option (see section 5.1.4 below).

capabilities These are mental actions that an agent can perform. The execution of a mental action updates the agent's belief base. Capabilities are of the form

$$\{Pre\} \;\; Capability(\bar{t}) \;\; \{Post\}$$

A mental action can be executed if the belief base entails the beliefs corresponding to their pre-conditions. Like in Prolog, the NOT operator in pre- and post-conditions is interpreted as negation as failure. The execution of the mental actions implies that the belief base will entail the beliefs that correspond to their post-conditions after the execution. For example, the mental action `Goto(R)` can be executed if the agent is at a position `pos(P)`. After the execution of the mental action, the position of the agent is not `pos(p)` anymore, but `pos(R)`.

goal base This contains a set of goals separated by a comma. A goal denotes a state that the agent wants to achieve. For example, `clean() , transport()` indicates two goals `clean()` and `transport()`. The first goal indicates that the agent wants to clean the rooms and the second goal indicates that the agent wants to transport boxes to their destinations. These are just two examples of basic goals, also called atomic goals. Complex goals can be formed by conjunction of atomic goals.

plan base The plan base contains a list of plans that the agents wants to perform. Agents can start with an initial set of plans, but new plans can also be generated during agent executions. These plans are generated to achieve the agent's goals. In this example, the initial plan base is empty.

pg-rule base This rule base contains a list of planning rules, called PG-rules, each indicates which plan should be generated to achieve a goal. Each pg-rule is of the form

$$Head \;\leftarrow\; Guard \;\mid\; Body$$

If the head of the rule matches an agent's goal and the belief base of the agent entails the guard of the rule, then the body of the rule, which is a plan, will be added to the plan base. In the PG-rules of Figure 11, the accolades `{` and `}` are used to indicate the body of the rule.

5.1.2 Loading a Program

Use the menu item `Open File` to load a 3APL file (see Figure 12). Choose and load the file `cleaning.3apl` from the `example` directory. There are three possibilities after loading a 3APL

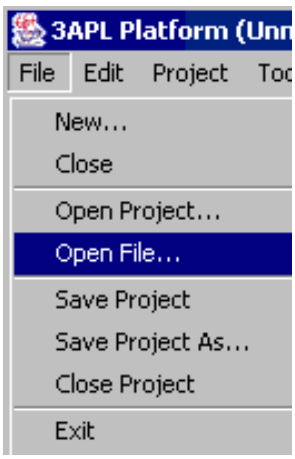

Figure 12: Loading a 3APL program program

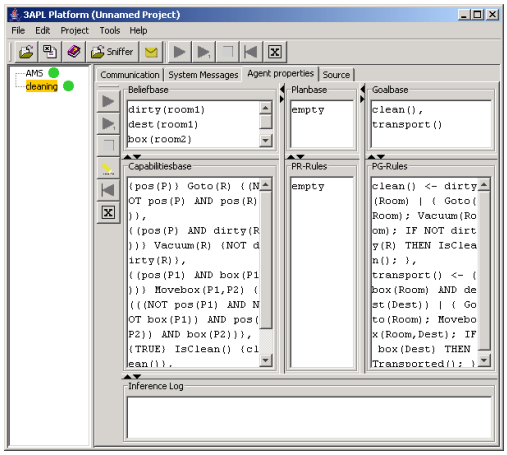

Figure 13: Agent Properties Panel

program. If the program can be parsed successfully, you will see the `Agent properties` panel, as in figure 13. Now you can edit the program or execute it directly. However, if there is a syntax error in the program, the program will not load. Error messages appear in the `System Messages` panel. There is also a small possibility that the 3APL program cannot be parsed, without specific syntax error messages being provided. In that case you must carefully recheck the syntax of your program.

5.1.3 Executing a Program

If the program `cleaning.3apl` is loaded successfully, you can execute it by pressing the run button ▶ on the left hand side of the `Agent properties` panel. Similar buttons on the top of the window have similar meaning for multi-agent cases (see section 5.3). You will now see various things moving. It is best to focus on the plan base. Press the ■ button to stop the program running. The button $\blacktriangleright_1$ will execute only one deliberation cycle at a time. After execution, you can trace the execution by means of the `Inference Log` window at the bottom part of the Agent properties panel. After a PG-rule is selected, the body of the rule is added to the plan base. After that, the first basic action is executed. This process of selection and execution continues until all plans have been executed and no rules are applicable anymore.

If you want to run the program again, you will have to reset the belief base, goal base, and the plan base. To reset, use the |◀ button. You can edit the program by clicking the yellow ⩥ button. After editing, the command `Save and recompile` in the edit window will reset and reload the program. The button can be used to remove a program from the `Agent properties` panel.

5.1.4 External Prolog Files

External Prolog files can be loaded into the 3APL belief base by means of the `LOAD` command at the beginning of a 3APL program. In this way the belief base of an agent is extended with the clauses defined in that file. These additional clauses will be used to execute test `actions` and to check the pre-conditions of mental actions, the guards of the rules, or the conditions of while loops and if-then-else statements. Since we use JIProlog [8] (a prolog engine written in JAVA), the standard JIProlog predicates and commands are also directly available in 3APL (see JIProlog

user manual [8] for more details).

In order to implement a 3APL agent which employs an external Prolog file, load in the 3APL platform the file `arithmetics.3apl` from the `examples` directory. This program is illustrated in Figure 14. Execute this 3APL program and observe that the agent has an initial plan, which is a sequence of three abstract plans. These abstract plans are subsequently refined by applying the corresponding PR-rules. As you can see, after the application of the first PR-rule, the plan is refined and executed. The test action `times(X,Y,Z)?` is then executed, which provides substitutions for the variables `X`,`Y` and `Z`. Note that the predicate `times(X,Y,Z)` is defined in the external file `cal.pl` which is loaded by the `LOAD` statement. In this (Prolog) file, which is also included in the `examples` directory, two other predicates are defined as well: `plus` and `min` (see figure 15).

```
PROGRAM "externalProlog"

LOAD "cal.pl"

CAPABILITIES{

    { TRUE } Multiplies(X,Y,Z)  { multiplies(X,Y,Z) },
    { TRUE } Addition(X,Y,Z)    { addition(X,Y,Z) },
    { TRUE } Subtraction(X,Y,Z) { substraction(X,Y,Z) }
}

BELIEFBASE{
    num1(10).
    num2(43).
}

GOALBASE{}

PLANBASE{
    { multiply();add();subtract() }
}

PG-RULES{}

PR-RULES{
     multiply() <- num1(X) AND num2(Y) |
                   { times(X,Y,Z)?;
                     Multiplies(X,Y,Z)
                   },
     add() <- num1(X) AND num2(Y) |
                   { plus(X,Y,Z)?;
                     Addition(X,Y,Z)
                   },
     subtract() <- num1(X) AND num2(Y) |
                   { min(X,Y,Z)?;
                     Subtraction(X,Y,Z)
                   }
}
```

Figure 14: Arithmetic 3APL program using external Prolog file

Although the Prolog clauses from an external file are part of the belief base, they are not shown explicitly in the belief base window of the Agent properties panel. The reason for this is methodological. External Prolog file will typically contain utilities, like general rules for some problem

```
plus(A,B,C)  :- C is A + B.
min(A,B,C)   :- C is A - B.
times(A,B,C) :- C is A * B.
```

Figure 15: Prolog file `cal.pl`

domain. We expect that these Prolog clauses remain unchanged during execution. The Agent properties panel is meant to trace changes to the belief base for debugging purposes. Therefore we decided not to clutter the belief base window with external Prolog clauses which are not meant to change. For this reason, we consider it bad practice to store dynamic beliefs as part of an external Prolog file.

5.1.5 List Manipulation in 3APL

The 3APL language does not provide common data-structures like arrays or records. Instead of records, 3APL uses Prolog facts, which represent data in the belief base of the agent. For example, an address can be stored as follows:

```
address(john_baley,15, oxford_road, islington, uk).
```

Instead of arrays or vectors, 3APL uses Prolog lists. By definition, a list is either an empty list `[]`, or a complex list such as `[H|T]`, `[H1,H2|T]` and `[[H1|T1],[H2|T2]]`. In general, you cannot access the elements of a list directly. Instead you have to decompose it step-by-step. There are many possible ways to do this, for example through PR-rules or while construct, as shown in Figure 16 . The use of PR-rules to access the elements of a list employs two PR-rules: one that recognizes the empty list, and one that reduces the list. To use while goals for repetition, we must somehow store the progress of the loop in the belief base of the agent. In this example, this is achieved by deleting elements from the belief base. If none are left, the loop is done. In order to run this example, load the 3APL program `list.3apl` from the `examples` directory and execute it. Observe the changes in the belief base and plan base.

5.2 Goal Dynamics

There are five actions in 3APL operating on the goal-base. These actions are designed to add and remove goals to/from the goal base and test if the goal base entails a goal. The syntax of these actios are follows:

goalaction ::=
“AdoptGoal(” *goal* “)” |
“DropGoal(” *goal* “)” |
“DropAllGoals(” *goal*? “)” |
goal “!” |
goal “!!”

“AdoptGoal” takes as an argument a grounded goal and adds that goal to the end of the goal-base. Execution blocks if not all variables in the goal argument have been substituted. “DropGoal” takes as an argument a goal which may contain free variables. The first goal in the goal-base which is equivalent with the goal argument is then removed. Execution blocks if such a goal cannot be found. “DropAllGoals” also takes as an argument a goal which may contain free variables. All goals in the goal-base that entail the goal argument are removed from the goal-base. If no goal argument is given to DropAllGoals, the entire goal-base is emptied. DropAllGoals never blocks. The exclamation marks are used to test the goal-base, just as the question mark is used to test the belief-base. The variables contained in the test goal are bounded by the terms in the first matching goal in the goal-base. A double exclamation mark forces an exact match, whereas a

```
PROGRAM "list"

CAPABILITIES{
    { list1([H|T]) }  Extract([H|T])  { NOT list1([H|T]),list1(T) },
    { TRUE }          Add(H)          { elem(H)}
}

BELIEFBASE{
    list1([john,mary,sue,ellen,bill]).
    empty([]).
}

GOALBASE{}

PLANBASE{
    { prloop(); whileloop() }
}

PG-RULES{}

PR-RULES{
    prloop()    <- list1(L) | loop(L),
    loop(L)     <- (L = []) | SKIP,
    loop(L)     <- (L =[H|T]) | { Add(H);loop(T)},
    whileloop() <- TRUE |
                    {
                        WHILE list1(L) AND NOT empty(L) DO
                            {Extract(L)}
                    }
}
```

Figure 16: List manipulations in 3APL by PR-rules or while goal

single exclamation mark allows the matched goal in the goal-base to contain additional subgoals. Execution blocks if a match cannot be found.

5.3 Communicating Agents

In multi-agent systems, individual agents must be able to do two things: they must be able to find other agents that offer some service and they must be able to communicate with one another. To this end, the 3APL multi-agent platform supports a *service directory* and *communication facilities*. Both of these facilities are implemented as functionalities of AMS.

5.3.1 Agent Communication

Messages from one agent to another agent are delivered by the underlying transport layer, as shown in 17. The messages themselves have a specific structure, which is compliant with the FIPA standards for agent communication [6]. The following fields are necessary elements of a message.

- Receiver (String): identity of the receiving agent
- Performative (String): communicative act type (inform, request, ...)
- Content (WFF): formula that expresses the content of the message

Messages can be sent by the dedicated 3APL capabilities

`Send(Receiver, Performative, Content)`

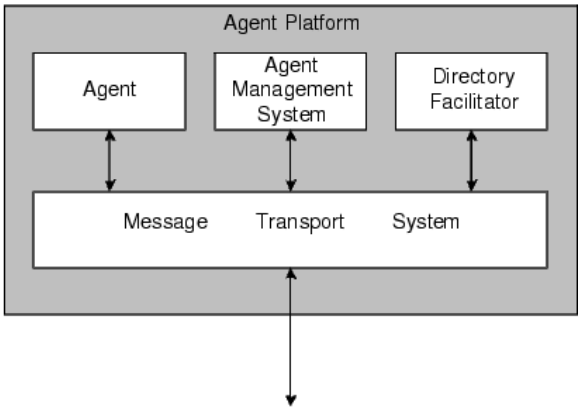

Figure 17: Multi-agent platform architecture

Agents can receive a message in their belief base at each moment in time. When an agent sends a message to a second agent, then the belief base of the sender is updated with the formula

$$\texttt{sent}(\texttt{Receiver}, \texttt{Performative}, \texttt{Content})$$

and the belief base of the receiver is updated with the formula

$$\texttt{received}(\texttt{Sender}, \texttt{Performative}, \texttt{Content})$$

An example of two agents capable of saying hello to each other is shown in figure 18. These agents are called `harry` and `sally`. Now load both 3APL programs `harry` and `sally` from the `examples` directory. If you run them, they will start by saying hello and respond to each other's hello with a thanks. The process of the message passing can be followed, by two tools provided in the platform, i.e., the `Communication` panel tool as shown in figure 4 and the `Sniffer` tool as shown in figure 7. To see the content of a message, you can double click the message labels.

In the previous example, the agents Harry and Sally were loaded separately. In order not to load all agents one by one for the next time they are needed, one can save a set of agents as a project such that they (i.e., the project) can be loaded simultaneously next time. A number of agents that are loaded on the platform can be saved as a project by the command `Save Project` or `Save Project As ...` under the `File` menu.

5.3.2 Service Directory Facilitator

In general, a directory facilitator provides yellow page services to agents on a platform. It maintains a searchable list of services of agents which have been registered. Agents can search the directory to find another agent that offers a service. A full directory facilitator has not yet been implemented in the 3APL platform. However, agents can use the agent management system to simulate the functionality of a directory facilitator.

To register a service, an agent can send an *inform* message to the AMS, of which the content is a simple expression of the from `description(Service)` that describes the service to be announced. To unregister, the same procedure is used, with `NOT` added before the description. To *request* a list of agents that provide a particular service, an agent can send a query message to the AMS, with a service description in the content. Suppose we have an agent *seller*, that wishes to announce to the AMS that it can sell computers. This agent will execute the communication action:

$$Send(ams, inform, description(sellcomp))$$

When the agent *seller* has no more computers to sell, it can notify the AMS by the action:

```
PROGRAM "harry"

BELIEFBASE{
    me(harry).
    you(sally).
}

GOALBASE{
    hello()
}

PLANBASE {}

PG-RULES{
    hello() <- you(You) AND NOT sent(You,inform,hello(You)) |
                { Send(You, inform, hello(You) ) },
            <- me(Me) AND received(You, inform, hello(Me)) AND
               NOT sent(You,inform,thanks(You)) |
                { Send(You, inform, thanks(You) ) }
}

PR-RULES{}

=====================================================================
PROGRAM "sally"

BELIEFBASE{
    me(sally).
    you(harry).
}

GOALBASE{
    hello()
}

PLANBASE{}

PG-RULES{
    hello() <- you(You) AND NOT sent(You,inform,hello(You)) |
                { Send(You, inform, hello(You) ) },
            <- me(Me) AND received(You, inform, hello(Me)) AND
               NOT sent(You,inform,thanks(You)) |
                { Send(You, inform, thanks(You) )}
}

PR-RULES{}
```

Figure 18: Communicating agents in 3APL

$$Send(ams, inform, NOTdescription(sellcomp))$$

On the other hand, an agent that wants to buy computers and is therefore looking for a computer seller, can use a message like:

$$Send(ams, query, description(sellcomp))$$

If the AMS finds agents with the required description, it returns a message with a list of agent names in its content. In the example, this will be a list consisting of one element: `[seller]`. So the sender of the query will get the following fact in its belief base:

$$received(ams, reply, name([seller]))$$

If the AMS cannot find an agent with the required description, it will return the empty list.

5.4 Shared Environment and External Actions

As explained, 3APL agents can either perform mental actions to update their beliefs or perform communication actions to communicate with other agents. In addition to mental and communication actions, 3APL agents can perform external actions with respect to an environment. The environment of 3APL agents is assumed to be implemented as a Java program. In particular, the actions that can be performed in this environment are determined by the methods of the Java program (i.e. the methods specify the effect of those actions in that environment) and the state of the environment is represented by the data structures of the Java program (i.e. either public or private). The external actions that can be performed by 3APL agents have the following general form:

$$Java("CLASSNAME", METHOD, LIST)$$

where $CLASSNAME$ is the name of the Java Class that implements the environment, $METHOD$ is the actual action to be performed in the environment, and $LIST$ is a list of returned values. It is important to note that $METHOD$ is a parameterized method of the java class $CLASSNAME$ and that $LIST$ is a list of values returned by the $METHOD$. An example of an external action is as follows:

$$Java("BlockWorld", east(), L)$$

where the external action `east()` is performed in the environment `BlockWorld`. The effect of this action is that the positions of the agent in the block world environment is shifted one slot to the right.

In order to illustrate the use of 3APL external actions consider the 3APL agent `envAgent.3apl` shown in figure 19. This agent performs external actions in the `"BlockWorld"` environment. This environment is the first window that pops up when the 3APL platform is started (see section 6 for more details on how the environment part of the platform works). The initial plan of this agent is to `enter` the block world followed by the abstract plan `start`. The entering part of the plan can be achieved by performing the external action `enter(4,1,-1)`, which has the effect that the agent enters the block world at position $(4, 1)$. The last parameter of the enter action is to indicate the time-out of the action. The value -1 indicates that the time-out for the action is infinite; a positive integer would indicate the time-out in milliseconds (see the specification of the environment actions under the `Library` item of the `Tools` menu of the 3APL platform.). The start abstract plan can be achieved by sensing at each row of the block world for bombs (starting from the the first row). If there is a bomb on its right block, then the bomb will be moved to its left block. The sensing action is accomplished by the `senseBombs` action resulting in the list of perceivable bombs. The list is then examined to find out if there is a bomb on the right block. This is done through the Prolog clause called `bombAt.pl`. If there a bomb on its right block, then a sequence of external actions are performed in order to go to the bomb, pick it up, going twice to the left side, dropping the goal there, and going back to the original position. The agent then moves to the next row and starts the process of moving bombs again.

```
PROGRAM "envAgent"

BELIEFBASE{
    bombAt(X,Y,[[X,Y]|T]).
    bombAt(X,Y,[[X1,Y1]|T]):-
        bombAt(X,Y,T).
}

GOALBASE{}

PLANBASE{
    { Java("BlockWorld", enter(4,1,-1), L) ; start(1) }
}

PG-RULES{}

PR-RULES{
    start(Y)    <- TRUE |
                    {
                        Java("BlockWorld", senseBombs(), BOMBS);
                        IF bombAt(5,Y,BOMBS) THEN
                            {
                                Java("BlockWorld", east(), L);
                                Java("BlockWorld", pickup(), L);
                                Java("BlockWorld", west(), L);
                                Java("BlockWorld", west(), L);
                                Java("BlockWorld", drop(), L);
                                Java("BlockWorld", east(), L)
                            };
                        Java("BlockWorld", south(), L);
                        start(Y+1);
                    }
}
```

Figure 19: External Actions in BlockWorld environment

Load the 3APL agent `envAgent` from the `examples` directory. Load a block world configuration through the `Load from File` button under the `World` menu of the block world environment and select the `blockworld.world` from the `examples` directory (see Figure 20). Execute the 3APL agent and observe the behavior of the agent. The blue circle around the agent in the environment indicates the range of its sense actions, i.e., which blocks of the environment the agent can perceive if it performs a `senseBombs` action. The sense range can be modified by the `Sense Range` button of the block world environment. The list of all actions that the 3APL agents can perform in this block world is as follows:

`void enter(X,Y,C)` : Create a agent representation in the environment at position (X,Y). The time-out of this action is indicated by C milliseconds (or -1 for infinite time).
`boolean north()` : Move agent on block north.
`boolean east ()` : Move agent on block east.
`boolean south ()` : Move agent on block south.
`boolean west ()` : Move agent on block west.
`boolean pickup ()` : Attempt to take bomb at agents' current location.
`boolean drop ()` : Attempt to drop bomb at agents' current location.
`ListPar senseTrap ()` : Sense the position of the bomb trap.
`ListPar sensePosition ()` : Sense the position of the agent.

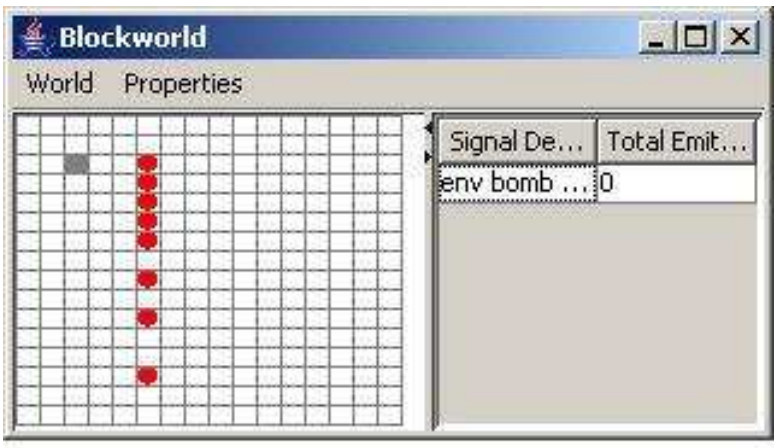

Figure 20: Block World Window

`ListPar senseBombs ()` : Sense for bombs in the visible area.
`ListPar senseStones ()` : Sense for stones in the visible area.
`ListPar senseAgents ()` : sense for agents in visible area

6 Plugin Developers Guide

The motivation for introducing Plugins to the 3APL platform is to extend the capabilities of the agents running on the platform with Java programs (external shared environment). The Plugin is a systematic way to achieve this goal by using it as an interface between the 3APL platform and Java programs. In particular, the Plugin facilitates the interaction between individual agents running on the platform and the Java programs. These interactions include method calls from agents to Java programs and event notification from the platform GUI. The implementation documentation of a plugin example (i.e. blockworld plugin) can be found at:

`http://www.cs.uu.nl/3apl/docs/blockworld-refman/index.html`

6.1 Anatomy of a plugin

Plugins are Java methods, grouped together under a single name: the plugin name. These methods are callable from the 3apl agent. The agent can pass arguments to these functions and they in turn can return values.

To create a plugin you need to implement three interfaces.

1. `ics.TripleApl.Plugin`: factory class
2. `ics.TripleApl.Instance`: product class
3. `ics.TripleApl.Method`: plugin method (function).

6.1.1 Overview

At startup, the platform loads all Plugin-implementing classes from the `plugins/` directory. It then queries the found plugin classes for their external functionalities (Java methods) they provide to individual agents. This is done by the platform through invocation of the method getMethods of the Plugin interface. The idea behind the plugin is to systematize the relation between agent platform and external functionalities that can be used by the agents. In particular, the external functionalities should be linked to the *individual* agents running on the platform such that the effect of any change on individual agents (create, reset or remove) on the platform can be realized on and passed to the corresponding external functionalities. For example, consider a blockworld as an external functionality in which the agents running on the platform can be present and move around. In such a case, if the user create, reset or remove an agent on the platform, the agent should be added to, reset (moved to initial position), or removed from the blockworld environment, respectively. The effects of the mentioned events (on the platform) are realized by the platform through invocation of one of the methods in the Plugin interface: createInstance, resetInstance, and removeInstance.

6.1.2 Details of `Plugin` (factory)

```
public interface Plugin {
    public List getMethods();
    public String toString();
    public String getDescription();
    public Instance createInstance(String agentName);
    public void resetInstance(Instance i);
    public void removeInstance(Instance i);
}
```

List getMethods() This method is called by the platform at startup to get the methods this plugin provides. It must return a Collection containing Method objects.

String toString() This method must return the name of the Plugin.

String getDescription() This method must return the description of the Plugin.

Instance createInstance(String agentName) This method is called by the platform every time an agent is added. It must create a new Instance object. That object refers/belongs to the agent identified by the argument agentName. Note that this name is unique.

void resetInstance(Instance i) Called by the platform when the agent this instance refers to is reset (that is, the user pressed the "recompile agent" button).

void removeInstance(Instance i) Called by the platform when the agent this instance refers to is about to be removed from the platform. The instance must take care that all listeners etc are removed so that it can be garbage collected.

6.1.3 Details of `Instance` (product)

```
public interface Instance {
    public String toString();
}
```

String toString() This method should return the agent's name this instance refers to.

6.1.4 Details of `Method`

```
public interface Method {
    public String getName();
    public String getDescription();
    public List getArguments();
    public ListPar execute(Instance instance, ListPar params);
}
```

String getName() Get the function name. The returned function name is assumed to be unique across all methods listed by Plugin.getMethods. This method is used by the platform in order to form an appropriate Java call. For example the name 'sense', from plugin 'Blockworld' would map to a 3APL Java call like this: `Java("Blockworld",sense(arg1,arg2,argN),RETURN_VALUE);`.

String getDescription() Get function description. This is used to display additional help information to the user in the 3APL platform interface.

List getArguments() Returns a collection of strings describing the arguments that should be passed to this method. The number of strings in this list is important as it is used to match the function calls based on their footprint, e.g. two string will only match a 3APL Java(...) call with two arguments. If less or more arguments are given and no other method with the given number of arguments exists, the platform will give a "Function not found" exception.

ListPar execute(Instance instance, ListPar params) Called by the platform to execute this method. The arguments passed are in `params`, documentation for the ListPar type can be found in the platform Javadoc generated documentation.

6.2 Example plugin

In this section, the use of the plugin system is illustrated by an example. The implementation is best understood by examining the source code (see section 6.2.2).

6.2.1 Description

The example is a simple window plugin. It allows each agent to open (at maximum) one window, write text to it, and close it again. Additionally, we want to hide the window if the agent is reset or removed. We need three methods:

- `windowOpen()` If hidden, show the window.
- `windowClose()` If shown, close the window.
- `windowWriteLn(String line)` Write a single text line to the window.

Apart from printing "Hello World", this plugin can actually be somewhat usefull for debugging.

6.2.2 Sourcecode

```
package windowplugin;

import Javax.swing.*; import Java.awt.BorderLayout; import
Java.util.*; import ics.TripleApl.*;


/************* Plugin *************/ public class WindowPlugin
```

```
    implements Plugin
{
    Vector _methods = new Vector();

    public WindowPlugin() {
        _methods.add(new WindowMethod_writeLn());
    }

    // return a Collection of Method objects this plugin
    // provides
    public List getMethods () {
        return _methods;
    }

    // return the name of the Plugin
    public String toString () {
        return "WindowPlugin";
    }

    // return the description of the Plugin
    public String getDescription () {
        return "A Window where agents can write text to";
    }


    // Create a new Instance, the instance refers/belongs
    // to a specific agent, identified by that agents name.
    public Instance createInstance (String agentName) {
        return new WindowInstance(agentName);
    }

    // Called by the interpreter when the agent this
    // instance refers to is reset.
    public void resetInstance (Instance agent) {
        ((WindowInstance)agent).hide();
    }

    // Called by the interpreter when the agent this
    // instance refers to is about to be remove from
    // the platform.
    public void removeInstance (Instance agent) {
        ((WindowInstance)agent).dispose();
    }
}


/************* Instance *************/
 class WindowInstance extends
JFrame
    implements Instance
{
    // widget to write text into
    JTextArea _textArea = new JTextArea();
```

```
    // string reference to (unique) agent name
    String _agentName;

    public WindowInstance(String agentName) {
        // set the window title and size
        super("Agent '" + agentName + "'");
        setSize(200,200);

        // store agent name
        _agentName = agentName;

        // add textarea (wrapped in scrollpane) to center
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(new JScrollPane(_textArea),
                BorderLayout.CENTER);

        // by default show agent
        show();
    }

    // write line to window
    public void writeLn(String line) {
        System.out.println("Window writeLn called!");
        // add text + newline character
        _textArea.append(line + "\n");

        // if the window is hidden, show it
        show();
    }

    // get this instance's agent name
    public String toString() {
        return _agentName;
    }
}


/************* Method ************/ class WindowMethod_windowOpen
    implements Method
{
    Vector _arguments = new Vector();

    public List getArguments () {
        // no arguments
        return _arguments;
    }

    public String getName() {
        return "windowOpen";
    }

    public String getDescription () {
        return "If hidden, show the window";
    }
```

```
    public ListPar execute (Instance instance, ListPar arg) {
        ((WindowInstance)instance).show();

        // no return value
        return null;
    }
}


/************* Method *************/ class
WindowMethod_windowClose
    implements Method
{
    Vector _arguments = new Vector();

    public List getArguments () {
        // no arguments
        return _arguments;
    }

    public String getName() {
        return "windowClose";
    }

    public String getDescription () {
        return "If shown, close the window";
    }

    public ListPar execute (Instance instance, ListPar arg) {
        ((WindowInstance)instance).hide();

        // no return value
        return null;
    }
}


/************* Method *************/ class WindowMethod_writeLn
    implements Method
{
    Vector _arguments = new Vector();
    public WindowMethod_writeLn() {
        _arguments.add("The line to write");
    }

    public List getArguments () {
        return _arguments;
    }

    public String getName() {
        return "writeLn";
    }
```

```
    public String getDescription () {
        return "Write a single line to the text window.";
    }

    public ListPar execute (Instance instance, ListPar arg) {
        final String string = arg.get(0).toString();
        ((WindowInstance)instance).writeLn(string);

        // no return value
        return null;
    }
}

/************* Method *************/ class WindowMethod_ readLn
    implements Method
{
   Vector _arguments = new Vector();
   public List getArguments () {
      // no arguments
      return _arguments;
   }

   public String getName() {
      return  "readLn";
   }

   public String getDescription() {
      return  "Read text from the text window.";
   }

   public ListPar execute (Instance instance, ListPar arg) {
      final String text = ((WindowInstance)instance).readLn();
      final ListPar p = new ListPar();
      p.add(new ConstSymPar(text));
      return p;
   }
}
```

References

[1] M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.

[2] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, John-Jules Meyer *A Programming Language for Cognitive Agents: Goal Directed 3APL* Proceedings of the First Workshop on Programming Multi-agent Systems: Languages, frameworks, techniques, and tools (ProMAS03), Mehdi Dastani, Jurgen Dix, Amal El Fallah-Seghrouchni (eds.), LNAI 3067, Springer, Berlin, 2004.

[3] Mehdi Dastani and Frank de Boer and Frank Dignum and John-Jules Meyer *Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language* Proceedings of The Second Conference on Autonomous Agents and Multi-agent Systems (AAMAS'03), Melbourne, Australia, 2003

[4] Mehdi Dastani and Jeroen van der Ham and Frank Dignum *Communication for Goal Directed Agents* In Communication in Multi-agent Systems - Agent Communication Languages and Conversation Policies, Marc-Philippe Huget (ed.), 239-252, LNCS, 2003.

[5] Eric ten Hoeve *3APL Platform* Master thesis computer science, Utrecht University, 2003 `http://www.cs.uu.nl/3apl/thesis/tenhoeve/EricTenHoeve.pdf`

[6] FIPA. *FIPA ACL Message Structure Specification.* Foundation for Intelligent Physical Agents,XC00061,2001.

[7] FIPA. *FIPA Contract Net Interaction Protocol Specification.* Foundation for Intelligent Physical Agents,SC00029H,2002.

[8] Ugo Chirico. *Java Internet Prolog.* http://www.ugosweb.com/jiprolog/index.shtml.