1 Introduction

The 2APL Platform is a development tool that is designed to support the implementation and execution of multi-agent systems programmed in 2APL. 2APL (A Practical Agent Programming Language, pronounced double-a-p-l) is a BDI-based agent-oriented programming language that supports an effective integration of declarative programming constructs such as belief and goals and imperative style programming constructs such as events and plans. The 2APL platform provides a graphical interface through which a user can develop and execute 2APL multi-agent systems using several facilities, such as a syntax-colored editor, different execution modes, and several debugging/observation tools. The platform allows communication among agents and can run on several machines connected in a network. Agents hosted on different 2APL platforms can communicate with each other.

1.1 Software requirements

The 2APL platform has been tested on Windows 2000 and Windows XP, as well as Mac OS X, Linux and Unix (Solaris). In order to run the 2APL platform, you need to have at least Java Runtime Environment (JRE) 6 or Java Developers Kit (JDK) 6 installed on your computer. The 2APL platform takes less than 4.5 MB on your harddisk.

1.2 Installation

In order to install the 2APL Platform you need to follow the next procedure:

1. Download the `2apl.zip` file from the following URL:

   `http : //www.cs.uu.nl/2apl/download.html`

2. Extract the contents of the ZIP-file into a directory. In the sequel, we assume that this directory is named `2apl`.

The ZIP-file contains two files: `2apl.jar`, which contains all the class files of the 2APL Platform, and `blockworld.jar` which is a grid like external environment in which agents can move around and perform actions. The `blockworld` will be explained in chapter 4. Note that the content of the ZIP-file also includes two directories called `lib` and `examples`. The `lib` directory contains the class files of JIProlog (a Java based Prolog reasoning engine that is used by the individual 2APL agents to represent and reason about their beliefs), Jext (a third party source code editor) and Jade. The `examples` directory contains the source of an example multi-agent system programmed in 2APL. The example is about two agents, `harry` and `sally`, who are located in the so-called `blockworld`. This `blockworld` is a $n \times n$ grid, which can contain bombs. `Sally` is responsible for searching bombs and notifying Harry in case a bomb has been found. `Harry` is responsible for cleaning up the `blockworld` by picking up the bomb and throwing it in a dustbin. Chapter 2 will elaborate on this example. In the sequel, we will explain the 2APL programming language and its platform in terms of this example and by loading and executing it.

1.3 Getting started

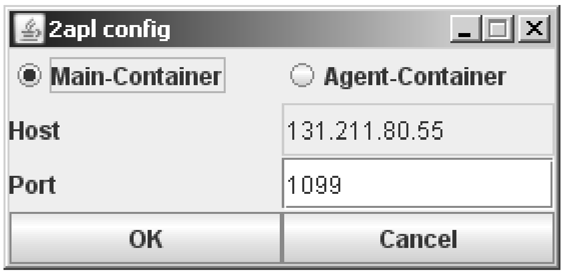
Before we explain the syntax of 2APL in the next chapter, we will first explain how you can start the 2APL platform and show you the basics of loading and executing a 2APL multi-agent system.

The 2APL platform can be started in:

Windows by double clicking the file `2apl.jar`, or alternatively, typing `java -jar 2apl.jar` in a command prompt window at the `2apl` directory

Mac OS by double clicking the file `2apl.jar`

Linux/Unix by typing `java -jar 2apl.jar` to a prompt in the `2apl` directory



● Main-Container ○ Agent-Container

Host	131.211.80.55
Port	1099

Figure 1.1: Container selection dialog window

After successful startup the dialog window as depicted in figure 1.1 should appear. Agents always run in a so-called container, an instance of the 2APL platform. Agents can either run within a main-container or within an agent-container. Agent-containers should always be connected to a main-container. This way agents running on different machines can communicate with each other. If you choose to run the agents in an agent-container, the IP-address and portnumber of the remote main-container the agent-container should connect to are to be specified. If you choose to run a main-container the IP-address equals the one of your own machine and you can specify the port other agent-containers can use to connect to this main-container. For now, select main-container and leave the IP and port unchanged.

After having selected to run a main-container the window as depicted in figure 1.2 appears (if an agent-container has been selected the window looks exactly the same). This is the main window of the 2APL platform that allows to load and execute a multi-agent system programmed in 2APL. To load an example multi-agent system, you should perform the following steps:

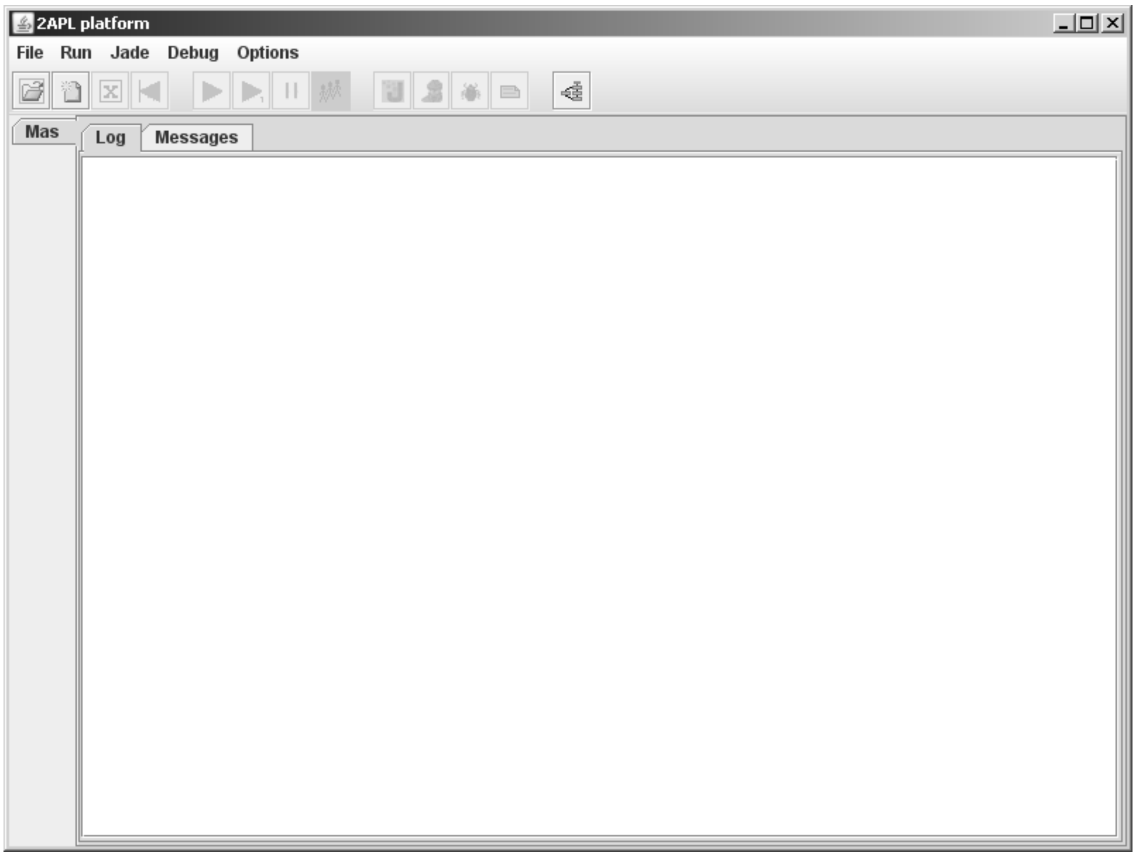
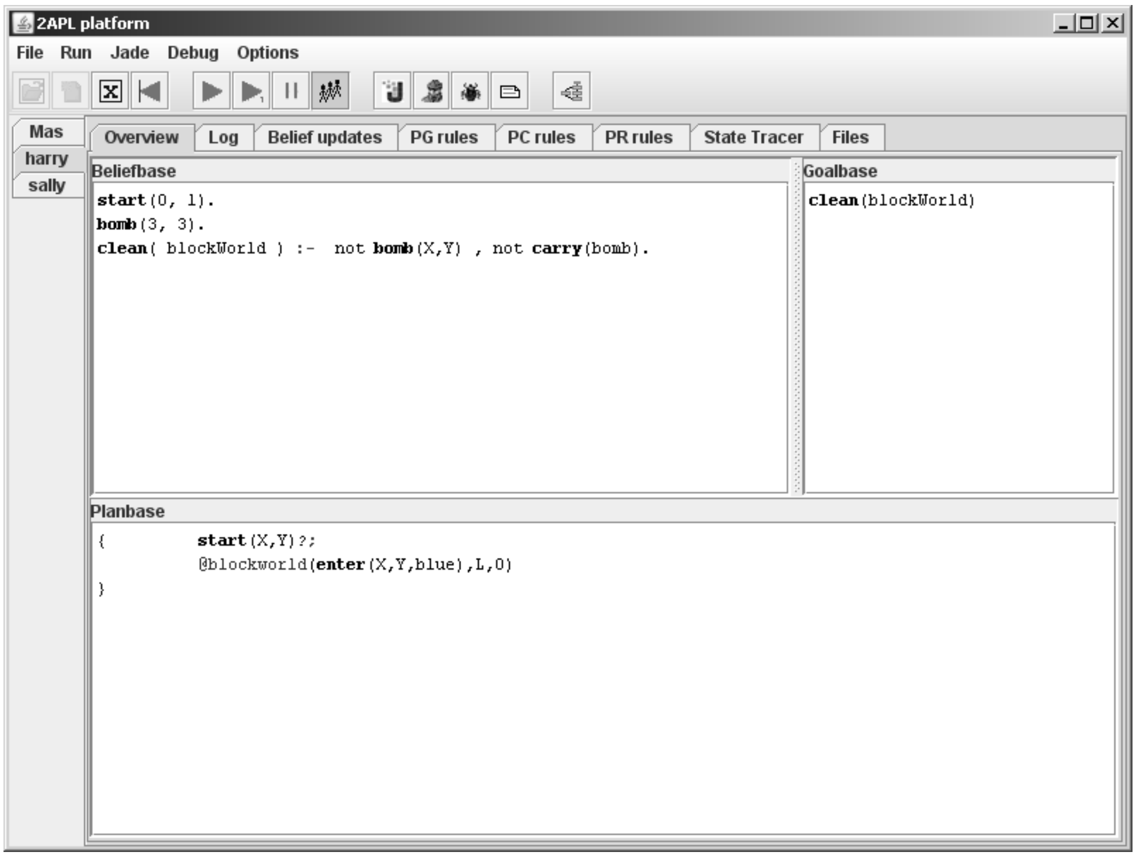


Figure 1.2: The main window

1. Select from the menu `file` → `open`, or alternatively the `open` button located on the toolbar, and an `open file` dialog appears
2. Browse to the directory `harrysally` in the `examples` directory
3. Open the file named `harryandsally.mas`

The `.mas` file specifies which agent files should be loaded, in this case the files `harry.2apl` and `sally.2apl`. After the file has been loaded the main screen looks like the window depicted in figure 1.3.

The left panel of the window allows you to toggle between the multi-agent system tab `Mas` and the tabs related to the different agents located within this multi-agent system. The right panel shows different tabs, depending on whether a specific agent or the multi-agent system has been selected in the left panel. In case of selecting the multi-agent system tab `Mas`, one can select in the right panel the `Log` tab to read the multi-agent system logs, the `Files` tab to access the file in which the multi-agent system is specified, and the `messages` tab to observe the messages that are exchanged between the agents when they are executed. In case an agent has been selected in the left panel, one can select different tabs in the right panel to observe and edit various ingredients of the selected 2APL agent. In this example and as illustrated in figure 1.3, one can select the `Overview` tab to observe the selected agent's mental state, the `Log` tab to observe the agent's execution log, the `Belief updates` tab to see the specification of belief update actions that the agent can perform, the `PG rules` tab to see the agent's rule that specifies how and by which plans the agent can realise its goals, the `PC rules` tab to view the agent's rules that specify how the


2APL platform
File Run Jade Debug Options
Mas
harry
sally
Overview Log Belief updates PG rules PC rules PR rules State Tracer Files
Beliefbase
Goalbase
Planbase

```
start(0, 1).
bomb(3, 3).
clean( blockWorld ) :-  not bomb(X,Y) , not carry(bomb).
```

```
clean(blockWorld)
```

```
{         start(X,Y)?;
          @blockworld(enter(X,Y,blue),L,0)
}
```

Figure 1.3: The main window after loading `harrysally.mas`

received events and messages should be handled, the `PR rules` tab to view the agent's rules that specify how the agent can repair its failed plans, and the `Files` tab to access the complete source files that specify the agent. By clicking on the `Edit` button in the `Files` tab the file will be opened in the Jext editor which is specially designed for the 2APL programs (more information on Jext can be found at `http://jext.sourceforge.net`). For other examples, the set of tabs for an individual agent can be different depending on the programming constructs that are used in the source file of the agent.

The 2APL program of a multi-agent system can be executed in different modes by means of the buttons on the toolbar, which can also be accessed through the `file` and `run` menu items. The meaning of these buttons is shown below. The rest of the toolbar buttons will be explained in chapter 3.

load a `mas` file

create a new `mas` file

close `mas` file

reload `mas` file

Execute agent(s) in continuous mode

Execute agent(s) in step-by-step mode

pause execution of agent(s)

apply abovementioned operations to mas, or

apply abovementioned operations to the selected agent

2 The 2APL Language

The 2APL programming language supports the integration of declarative concepts such as belief and goals with imperative style programming such as events and plans. This chapter presents the complete syntax of 2APL, which is specified using the EBNF notation. In this specification, illustrated in figure 2.1, we use $\langle atom \rangle$ to denote a Prolog like atomic formula starting with lowercase letter, $\langle Atom \rangle$ to denote a Prolog like atomic formula starting with a capital letter, $\langle ident \rangle$ to denote a string and $\langle Var \rangle$ to denote a string starting with a capital letter. We use $\langle ground\_atom \rangle$ to denote a grounded atomic formula.

An individual 2APL agent may be composed of various ingredients that specify different aspects of the agency. A 2APL agent can be programmed by implementing the initial state of those ingredients. The state of some of these ingredients will change during the agent's execution while the state of other ingredients remains the same during the execution of the agent. In the rest of this chapter, we will discuss each ingredient and give examples to illustrate them. Before explaining the syntax of a single agent, however, we first consider the multi-agent system that is specified in terms of multiple agents.

2.1 Multi-agent system specification

2APL is a multi-agent programming language that provides programming constructs to specify both multi-agent configuration as well as individual agents. The configuration of a 2APL multi-agent system is specified through a MAS specification file, i.e., a file with `.mas` extension. This file specifies which agents participate in the multi-agent system, what names they get, and what external environments they have access to. A 2APL multi-agent system can be specified as follows:

```
agentname :  filename qt @env1,...,envn
⋮
agentname :  filename qt @env1',...,envn'
```

where

`agentname` is the name of the agent

```
⟨Agent_Prog⟩      =   { "Include:" ⟨ident⟩
                  |   "BeliefUpdates:" ⟨BelUpSpec⟩
                  |   "Beliefs:" ⟨belief⟩
                  |   "Goals:" ⟨goals⟩
                  |   "Plans:" ⟨plans⟩
                  |   "PG-rules:" ⟨pgrules⟩
                  |   "PC-rules:" ⟨pcrules⟩
                  |   "PR-rules:" ⟨prrules⟩ } ;
⟨BelUpSpec⟩       =   ( "{"⟨belquery⟩"}" ⟨beliefupdate⟩ "{"⟨literals⟩"}" )+ ;
⟨belief⟩          =   ( ⟨ground_atom⟩"." | ⟨atom⟩ ":-" ⟨literals⟩"." )+ ;
⟨goals⟩           =   ⟨goal⟩ { ","⟨goal⟩ } ;
⟨goal⟩            =   ⟨ground_atom⟩ { "and" ⟨ground_atom⟩ } ;
⟨baction⟩         =   "skip" | ⟨beliefupdate⟩ | ⟨sendaction⟩ | ⟨externalaction⟩
                  |   ⟨abstractaction⟩ | ⟨test⟩ | ⟨adoptgoal⟩ | ⟨dropgoal⟩ ;
⟨plans⟩           =   ⟨plan⟩ { "," ⟨plan⟩ } ;
⟨plan⟩            =   ⟨baction⟩ | ⟨sequenceplan⟩ | ⟨ifplan⟩ | ⟨whileplan⟩ | ⟨atomicplan⟩ ;
⟨beliefupdate⟩    =   ⟨Atom⟩ ;
⟨sendaction⟩      =   "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")" ;
                  |   "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")" ;
⟨externalaction⟩  =   "@"⟨ident⟩"("⟨atom⟩ "," ⟨Var⟩ ")" ;
⟨abstractaction⟩  =   ⟨atom⟩ ;
⟨test⟩            =   "B(" ⟨belquery⟩ ")" | "G"( ⟨goalquery⟩ ")" | ⟨test⟩ "&" ⟨test⟩ ;
⟨adoptgoal⟩       =   "adopta(" ⟨goalvar⟩ ")" | "adoptz(" ⟨goalvar⟩ ")" ;
⟨dropgoal⟩        =   "dropgoal(" ⟨goalvar⟩ ")" | "dropsubgoals(" ⟨goalvar⟩ ")"
                  |   "dropsupergoals(" ⟨goalvar⟩ ")" ;
⟨sequenceplan⟩    =   ⟨plan⟩ ";" ⟨plan⟩ ;
⟨ifplan⟩          =   "if" ⟨test⟩ "then" ⟨scopeplan⟩ ["else" ⟨scopeplan⟩] ;
⟨whileplan⟩       =   "while" ⟨test⟩ "do" ⟨scopeplan⟩ ;
⟨atomicplan⟩      =   "[" ⟨plan⟩ "]" ;
⟨scopeplan⟩       =   "{" ⟨plan⟩ "}" ;
⟨pgrules⟩         =   ⟨pgrule⟩+ ;
⟨pgrule⟩          =   [⟨goalquery⟩] "<-" ⟨belquery⟩ "|" ⟨plan⟩ ;
⟨pcrules⟩         =   ⟨pcrule⟩+ ;
⟨pcrule⟩          =   ⟨atom⟩ "<-" ⟨belquery⟩ "|" ⟨plan⟩ ;
⟨prrules⟩         =   ⟨prrule⟩+ ;
⟨prrule⟩          =   ⟨planvar⟩ "<-" ⟨belquery⟩ "|" ⟨planvar⟩ ;
⟨goalvar⟩         =   ⟨atom⟩{"and"⟨atom⟩} ;
⟨planvar⟩         =   ⟨plan⟩ | ⟨Var⟩ | "if" ⟨test⟩ "then" ⟨scopeplanvar⟩ ["else" ⟨scopeplanvar⟩]
                  |   "while" ⟨test⟩ "do" ⟨scopeplanvar⟩ | ⟨planvar⟩ ";" ⟨planvar⟩ ;
⟨scopeplanvar⟩    =   "{" ⟨planvar⟩ "}" ;
⟨literals⟩        =   ⟨literal⟩ { "," ⟨literal⟩} ;
⟨literal⟩         =   ⟨atom⟩ | "not" ⟨atom⟩ ;
⟨belquery⟩        =   "true" | ⟨belquery⟩ "and" ⟨belquery⟩ | ⟨belquery⟩ "or" ⟨belquery⟩
                  |   "(" ⟨belquery⟩ ")" | ⟨literal⟩ ;
⟨goalquery⟩       =   "true" | ⟨goalquery⟩ "and" ⟨goalquery⟩ | ⟨goalquery⟩ "or" ⟨goalquery⟩
                  |   "(" ⟨goalquery⟩ ")" | ⟨atom⟩ ;
⟨iv⟩              =   ⟨ident⟩ | ⟨Var⟩ ;
```

Figure 2.1: The EBNF syntax of 2APL individual agents.

`filename` is the name of a file (with .2apl extension) that contains the 2APL program of the agent

`qt` is the number of agents to be initiated based on the file `filename`. When specifying a quantity larger than one for agents, the name of these agents will be extended with a unique number. The number `qt` is an optional argument.

`env1,...,envn, env1',...,envn'` are the names of the environments to which the agents `agentname` have access. This is an optional argument.

An example of a MAS specification file, in which the two agents of the previous chapter (`harry` and `sally`) participate, is shown in code fragment 2.1. Both agents, named `harry` and `sally` are defined by `harry.2apl` and `sally.2apl` and are situated in the `blockworld` environment. Note that the optional argument `qt` has been omitted, meaning that there will be only one instance of each agent.

```
harry : harry.2apl @blockworld
sally : sally.2apl @blockworld
```

Code fragment 2.1: An example of a mas specification file.

2.2 Beliefs and goals

A 2APL agent may have beliefs and goals which change during the agent's execution. The *beliefs* of the agents are implemented by the belief base, which contains information the agent believes about its surrounding world including other agents. The implementation of the initial belief base starts with the keyword '`Beliefs:`' followed by one or more belief expressions of the form ⟨*belief*⟩. Note that a ⟨*belief*⟩ expression is treated as a Prolog fact or rule such that the belief base of a 2APL agent becomes a Prolog program. All facts are assumed to be grounded.

```
Beliefs:
  bomb(3,4).
  clean( blockWorld ) :- not bomb(X,Y) , not carry(bomb).
```

Code fragment 2.2: The initial belief base of harry.

Code fragment 2.2 illustrates the implementation of the initial belief base of a 2APL agent. This belief base represents the information of `harry` about its `blockWorld` environment. In particular, the agent believes that there is a bomb at location $(3, 4)$, and that the `blockWorld` environment is clean if there are no bombs anymore and the agent is not carrying a bomb.

The *goals* of a 2APL agent are implemented by its goal base, which consists of formulas each of which denotes a situation the agent wants to realize (not necessary all at once). The implementation of the initial goal base starts with the keyword '`Goals:`' followed by a list of goal expressions of the form ⟨*goal*⟩. Each goal expression is a conjunction of ground atoms. Note that the separated goals in the goal base are separated by a comma. Ground atoms are treated as Prolog facts. Note that having a single conjunctive goal, say '`a and b`', is different than having two separate goals '`a , b`'. In the latter case, the agent wants to achieve two desirable situations independently of each other.

The example listed in code fragment 2.3 is the implementation of the initial goal base of a 2APL agent. This goal base indicates that the agent wants to achieve a desirable situation in which the `blockworld` is clean.

```
Goals:
  clean( blockWorld )
```

Code fragment 2.3: An example goal base of an agent.

The beliefs and goals of agents are related to each other. In fact, if an agent believes a certain fact, then the agent should not pursue that fact as a goal. This means that if an agent modifies its belief base, then its goal base may be modified as well.

2.3 Basic actions

In order to achieve its goals, a 2APL agent needs to act. Basic actions specify the capabilities that an agent can perform to achieve its desirable situation. The basic actions will constitute an agent's plan, as we will see in the next subsection. In 2APL, six types of basic actions are distinguished: actions to update the belief base, communication actions, external actions to be performed in an agent's environment, abstract actions, actions to test the belief and goal bases, and actions to manage the dynamics of goals.

Belief Update Action

A *belief update action* updates the belief base of an agent when executed. A belief update action ⟨*beliefupdate*⟩ is an expression of the predicate argument form where the predicate starts with a capital letter. Such an action is specified in terms of pre- and post-conditions. An agent can execute a belief update action if the pre-condition of the action is derivable from its belief base. The pre-condition is a formula consisting of literals composed by disjunction and conjunction operators. The execution of a belief update action modifies the belief base in such a way that after the execution the post-condition of the action is derivable from the belief base. The post-condition of a belief update action is a list of literals. The update of the belief base by such an action removes the atom of the negative literals from the belief base and adds the positive literals to the belief base. The execution of a belief base has as side-effect that goals that become believed to be achieved are removed. The specification of the belief update actions starts with the keyword '`BeliefUpdates:`' followed by the specifications of a set of belief update actions ⟨*BelUpSpec*⟩.

```
BeliefUpdates:
  { bomb(X,Y) }           RemoveBomb(X,Y) { not bomb(X,Y) }
  { true }                AddBomb(X,Y)    { bomb(X,Y) }
  { carry( bomb ) }       Drop( )         { not carry( bomb ) }
  { not carry( bomb ) }   PickUp( )       { carry( bomb ) }
```

Code fragment 2.4: An example of belief updates of `harry`.

Code fragment 2.4 shows an example of the specification of the belief update actions of `harry`. In this example, the specification of the `PickUp()` indicates that this belief update action can be performed if the agent does not already carry a bomb (i.e., the agent can carry only one bomb) and that after performing this action the agent will carry a bomb. Note that the agent cannot perform two `PickUp()` action consecutively. Note the use of variables in the specification of `RemoveBomb()`; it requires that an agent can remove a bomb if it is the same location as the bomb. Note also that variables in the post-conditions are bounded since otherwise the facts in the belief base will not be grounded.

Communication Action

A *communication action* passes a message to another agent. A communication action ⟨*sendaction*⟩ can have either three or five parameters. In the first case, the communication action is the expression `send(Receiver, Performative, Language, Ontology, Content)` where `Receiver` is a name referring to the receiving agent, `Performative` is a speech act name (e.g. inform, request, etc.), `Language` is the name of the language used to represent the content of the message, `Ontology` is the name of the ontology used in the content of the message, and `Content` is an expression representing the content of the message. It is often the case that agents assume a certain language and ontology such that it is not necessary to pass them as parameters of their communication actions. The second version of the communication action is therefore the expression `send(Receiver, Performative, Content)`. It should be noted that 2APL interpreter is built on the JADE platform. For this reason, the name of the receiving agent can be a local name or a full JADE name. A full jade name has the form `localname@host:port/JADE` where `localname` is the name as used by 2APL, `host` is the name of the host running the agent's container and `port` is the port number where the agent's container, should listen to (see [1] for more information on JADE standards). An example of a communication action is `sally` informing `harry` about a bomb location, as listed in code fragment 2.5.

```
send( harry, inform, La, On, bombAt( X1, Y1 ) )
```

Code fragment 2.5: An example of a communication action performed by `sally`.

External Action

An *external action* is supposed to change the external environment in which the agents operate. The effects of external actions are assumed to be determined by the environment and might not be known to the agents beforehand. The agent thus decides to perform an external action and the external environment determines the effect of the action. The agent can come to know the effects of an external action by performing a sense action, defined as an external action, or by means of events generated by the environment. An external action ⟨*externalaction*⟩ is an expression of the form `@Env(ActionName,Return,Time-out)`, where `Env` is the name of the agent's environment to which it has access, implemented as a Java class. The parameter `ActionName` is a method call (of the Java class) that specifies the effect of the external action in the environment. The environment is assumed to have a state represented by the instance variables of the class. The execution of an action in an environment is then a read/write operation on the state of the environment. The parameter `Return` is a list of values, possibly an empty list, returned by the corresponding method. The parameter `Time-out` is optional and equal to zero when it is not specified. The time-out parameter is used for specifying a timespan in which actions can be retried again in case of failure. When the execution of the external action fails and the time specified by the time-out is not elapsed the action is executed again. In this case the agent is oblivious about the action failure. Otherwise, when the time-out is elapsed, the action is not tried again and the action is candidate for repairment by a PR-rule (see section 2.5). Actions with a time-out of zero are thus never retried in case of failure. Consider, for example, an action that takes three seconds to execute and a time-out of five seconds. Then if the action fails, it is retried. Even though the second attempt will not finish within the time-out, the action can still succeed (depending on the result of the action). An example of the implementation of an external action is `@blockworld( east(),_,2)`. This action causes an agent to go one step to the east in the blockworld environment. The timeout value of two means that every time the agent cannot move east and less than two seconds have elapsed since the first try, the external action will be executed again. A successful execution of this action shifts the position of the agent in the blockworld environment one slot to the right. An empty list is returned.

Abstract Action

An *abstract action* is an abstraction mechanism allowing the encapsulation of a plan by a single action. An abstract action will be instantiated with a concrete plan when the action is executed. The instantiation of a plan with an abstract action is specified through special rules called PC-rule, which stands for procedure rules (see section 2.5 for a description of PC-rules). In fact, the general idea of an abstract action is similar to a procedure in imperative programming languages while the PC-rules function as procedure definitions. Like a procedure call, an abstract action ⟨*abstractaction*⟩ is an expression of the predicate argument form starting with a lowercase letter.

Belief and Goal Test Actions

A *belief test action* is to test whether a belief expression is derivable from an agent's belief base, i.e., it tests whether the agent has a certain belief. A belief test action is an expression starting with a capital `B` followed by a ⟨*belquery*⟩, an expression of the predicate argument form. Such a test may generate a substitution for the variables that are used as arguments in the belief expression. A belief test action is basically a (Prolog) query to the belief base which can be used in a plan to 1) instantiate a variable in rest of the plan, or 2) block the execution of the plan (if the test fails).

A *goal test action* is to test whether a formula is derivable from the goal base, i.e., whether the agent has a certain goal from which the formula is derivable. A goal test action is an expression starting with a capital `G` followed by a ⟨*goalquery*⟩, an expression of predicate argument form. For example, if an agent has a goal `p(a) and q(b)`, then the goal test action `G(p(X))` succeeds resulting in the substitution `[X/a]`. Like a belief test action, this action can be used to instantiate a variable with a value, or to block the execution of the rest of a plan.

In addition, 2APL provides a variant of the goal test action, called *exact goal test action.* This action succeeds only if the agent has the tested goal as one of its goals. The exact goal test action is implemented as a *goal test action* preceded by an exclamation mark '`!`'. For example, if the agent has only the goals `p(a) and q(b) , r(c)`, then the exact goal test action `!G(p(X) and q(Y))` will succeed (resulting in substitution `[X/a , Y/b]`) while the exact goal test action `!G(p(X))` fails. It should be noted that the derivability is defined with respect to one single goal present in the goal base, and not with respect to the set of agent's goals. That is to say, if the agent has two goals `p(a), q(a)` (and not one single conjunctive goal), then both goal test actions `G(p(X) and q(X))` and `!G(p(X) and q(X))` fail.

Goal tests and belief tests can be combined within one ⟨*test*⟩ expression, allowing an agent to assert both its beliefs and goals at the same time. Goal tests and belief tests are then composed by an ampersand (`&`), intuitively meaning that both goal and belief are derivable at the same time from goal and belief base, respectively. Again, if the ⟨*test*⟩ cannot be entailed by the belief and goal base of teh agent the rest of the plan will block until ⟨*test*⟩ is derivable.

Goal Dynamics Actions

The *adopt goal* and *drop goal* actions are used to adopt and drop a goal to and from the agent's goal base, respectively. The adopt goal action ⟨*adoptgoal*⟩ can have two different forms: `adopta(`ϕ`)` and `adoptz(`ϕ`)`. These two actions can be used to add the goal ϕ (a conjunction of literals) to the begin and to the end of an agent's goal base, respectively. Note that the programmer has to ensure that the variables in ϕ are instantiated before these actions are executed since the goal base should always be grounded.

Finally, the drop goal action ⟨*dropgoal*⟩ is used to drop goals and can have three different forms: `dropgoal(`ϕ`)`, `dropsubgoals(`ϕ`)`, and `dropsupergoals(`ϕ`)`. These actions can be used to drop from an agent's goal base, respectively, exactly the goal ϕ, all goals that are subgoal of ϕ, and all goals that have ϕ as a subgoal (i.e. are supergoal of ϕ). To illustrate the use of these three different actions to drop a goal consider the goal base: `{a(1), a(1) and b(1), a(1) and b(1) and c(1)}`. Below is listed which goals are dropped from this goal base in case of different drop goal actions performed on it.

```
dropgoal(a(1) and b(1))         drops  a(1) and b(1)
dropsubgoals(a(1) and b(1))     drops  a(1), a(1) and b(1)
dropsupergoals(a(1) and b(1))   drops  a(1) and b(1), a(1) and b(1) and c(1)
```

2.4 Plans

In order to reach its goals, a 2APL agent adopts *plans.* A plan consists of basic actions composed by process operators. In particular, basic actions can be composed by means of the sequence operator, conditional choice operators, conditional iteration operator, and an unary operator to identify (region of) plans that should be executed atomically, i.e., the actions should not be interleaved with the actions of other plans of the agent.

The sequence operator `;` is a binary operator generating the plan ⟨*sequenceplan*⟩ from two other plans. It indicates that the first plan should be performed before the second plan. The conditional choice operator generates the plan ⟨*ifplan*⟩, which is an expression of the form `if` ⟨*test*⟩ `then` π_1 `else` π_2. The condition of such an ⟨*test*⟩ expression is composed of belief and goal test as described above. In contrast to the usage of such a test outside a conditional test, when such a test is used within an `if`-statement the test is non-blocking. The ⟨*test*⟩ is thus evaluated with respect to an agent's belief and goal base possibly resulting a substitution. The scope of application of the substitution is limited to the body of this ⟨*ifplan*⟩ expression. This expression can thus be interpreted as to perform the if-part of the plan (i.e., π_1) when the ⟨*test*⟩ is derivable from the agent's belief and goal base, otherwise the agent performs the else-part of the plan (i.e., π_2). The conditional iteration operator generates the plan ⟨*whileplan*⟩ which is an expression of the form `while` ⟨*test*⟩ `do` π. As before, the condition ⟨*test*⟩ is also evaluated with respect to an agent's belief and goal base. Like the conditional choice operator, the scope of application for a possible substitution is limited to the body of this ⟨*whileplan*⟩ expression. This iteration expression is then interpreted as to perform the plan π in the body of the while loop as long as the agent can derive ⟨*test*⟩, meaning that the agent has beliefs and goals as expressed by the ⟨*test*⟩ expression. The last unary operator generates the plan ⟨*atomicplan*⟩ which is an expression of the form `[`π`]`. This plan is interpreted as an atomic plan, which should be executed at once (atomically) ensuring that the execution of π is not interleaved with the execution of the actions of other plans of the same agent. Note that an agent can have different plans at the same time.

The plans of a 2APL agent are implemented by its plan base. The implementation of the initial plan base starts with the keyword '`Plans:`' followed by a list of plans. Code fragment 2.6 illustrates the implementation of the initial plan base of `harry`, which enters the `blockworld` by performing an external action `enter`. The exact meaning of the external actions that could be performed in the `blockworld` is explained in chapter 4. As explained in the next section, during execution of the agent, the plan base will be filled with plans by means of reasoning rules.

```
Plans:
  @blockworld( enter( 0, 0, blue ), L )
```

Code fragment 2.6: An example of an initial plan base of an agent.

2.5 Reasoning rules

The 2APL programming language provides constructs to implement practical reasoning rules that can be used to implement the generation of plans. In particular, three types of practical reasoning rules are proposed: planning goal rules, procedural rules, and plan repair rules. In the following subsections, we explain these three types of rules.

Planning Goal Rules (PG rules)

A planning goal rule specifies that an agent should generate a plan if it has certain goals and beliefs. The specification of a planning goal rule ⟨*pgrule*⟩ consists of three entries: the head of the rule, the condition of the rule, and the body of the rule. The head and the condition of a planning goal rule are query expressions used to test if the agent has a certain goal and belief, respectively. The body of the rule is a plan in which variables may occur. These variables can be bound by the variables that occur in the goal and belief expressions. A planning goal rule of an agent can be applied when the goal and belief expressions (in the head and the condition of the rule) are derivable from the agent's goal and the belief bases, respectively. The application of a planning goal rule involves an instantiation of variables that occur in the head and condition of the rule as they are queried from the goal and belief bases. The resulted substitution will be applied to the generated plan to instantiate its variables. A planning goal rule is of the form:

[⟨*goalquery*⟩] "< −" ⟨*belquery*⟩ "|" ⟨*plan*⟩

Note that the head of the rule is optional which means that the agent can generate a plan only based on its belief condition. Code fragment 2.7 is an example of a planning goal rule of `harry` indicating that a plan to achieve the goal `clean(blockWorld)` can be generated if the agent believes there is a bomb at position `(X,Y)`. Note that `goto(X,Y)` is an abstract action.

Procedural Rules (PC rules)

Procedural rules generate plans as a response to 1) the reception of messages sent by other agents, 2) events generated by the external environment, and 3) the execution of abstract actions. Like planning goal rules, the specification of procedural rules consist of three entries. The only difference is that the head of the procedural rules is an atom ⟨*atom*⟩ (predicate-argument expression), rather than a goal query ⟨*goalquery*⟩, which represents either a message, an event, or an abstract

```
PG-rules:
  clean( blockWorld ) <- bomb( X, Y ) |
  {
    goto( X, Y );
    @blockworld( pickup( ), L1 );
    PickUp( );
    RemoveBomb( X, Y );
    goto( 0, 0 );
    @blockworld( drop( ), L2 );
    Drop( )
  }
```

Code fragment 2.7: An example of planning goal rules of `harry`.

action. A message and an event are represented by atoms with the special predicates `message` and `event`, respectively. An abstract action is represented by any predicate name starting with a lowercase letter. Note that like planning goal rules, a procedural rule has a belief condition indicating when a message (or received event or abstract action) should cause the generation of a plan. Thus, a procedural rule can be applied if the agent has received a message (or an event or if the agent executes an abstract action) and the belief query of the rule is derivable from its belief base. The instantiation of variables and the application of the resulting substitutions to the plan variables are the same as with planning goal rules. A procedural rule ⟨*pcrule*⟩ is of the form:

⟨*atom*⟩ "< −" ⟨*belquery*⟩ "|" ⟨*plan*⟩

Code fragment 2.8 shows two examples of procedural call rules of `harry`. The first rule indicates that if `harry` receives a message from `sally` informing him that there is a bomb at position `(X,Y)` and `harry` does not believe there are any bombs, then `harry` updates his beliefs with this new fact and adds a goal to clean the `blockworld` again. The second rule indicates that the abstract action `goto` should be performed as a certain sequence of actions, i.e. moving one square at a time. Note the use of recursion in this PC-rule.

Plan Repair Rules (PR rules)

The execution of an agent's action might fail. To repair such actions 2APL provides so-called plan repair rules. Like other practical reasoning rules, a plan repair rule consists of three entries: two abstract plans and one belief query expression. We have used the term abstract plan since such plans include variables that can be instantiated with a plan. A plan repair rule indicates that if the execution of an agent's plan (i.e., any plan that can be instantiated with the abstract plan) fails and the agent has a certain belief, then the failed plan should be replaced by another plan. A plan repair rule ⟨*prrule*⟩ has the following form:

⟨*planvar*⟩ "< −" ⟨*belquery*⟩ "|" ⟨*planvar*⟩

A plan repair rule of an agent can thus be applied if:

1. the execution of one of its plan fails,
2. the failed plan can be matched with the abstract plan in the head of the rule, and

```
PC-RULES:
  message( sally, inform, La, On, bombAt( X, Y ) ) <- true |
  {
    if ( not bomb( A, B ) ) then
    { AddBomb( X, Y );
      adoptz( clean( blockWorld ) )
    }
    else
    { AddBomb( X, Y )
    }
  }

  goto( X, Y ) <- true |
  {
    @blockworld( sensePosition(), POS );
    B(POS = [A,B]);
    if B(A > X) then
    { @blockworld( west(), L );
      goto( X, Y )
    }
    else if B(A < X) then
    { @blockworld( east(), L );
      goto( X, Y )
    }
    else if B(B > Y) then
    { @blockworld( north(), L );
      goto( X, Y )
    }
    else if B(B < Y) then
    { @blockworld( south(), L );
      goto( X, Y )
    }
  }
```

Code fragment 2.8: An example of procedural rules of `harry`.

3. the belief query expression is derivable from the agent's belief base.

The satisfaction of these three conditions results in a substitution that binds the variables that occur in the abstract plan in the body of the rule. Note that some of these variables will be instantiated with a part of the failed plan through the match between the abstract plan in the head of the rule and the failed plan. For the matching of plan variables, the strategy as described below is used.

Matching Strategy Given a plan and an abstract plan, the resulted match should satisfy the following two conditions:

- Each plan variable in the abstract plan should be matched with the shortest possible section of the plan
- A plan variable can only match with a section of the plan longer than one action iff:
  1. the plan variable is the last part of the abstract plan, or

2. matching the plan variable with one action does not result in a match between the plan and abstract plan

Let π, π_1, π_2 be plans and X be a plan variable. The following examples illustrate the match between plans and abstract plans according to the matching strategy as described above:

Plan	Abstract Plan	Substitution
$\pi_1; \pi; \pi_2$	$\pi_1; X; \pi_2$	$X = \pi$
$\pi_1, \pi; \pi; \pi_2$	$\pi_1; X; \pi_2$	$X = \pi; \pi$
$\pi_1, \pi_2; \pi; \pi$	$\pi_1; X$	$X = \pi_2; \pi; \pi$
$\pi_1, \pi_2; \pi; \pi$	$\pi_1; X; Y; \pi; \pi$	No Substitution

The resulted substitutions will be applied to the second abstract plan, which constitutes the body of the rule, to generate a new (repaired) plan. Code fragment 2.9 shows an example of a plan repair rule of `harry`. This rule is used for the situation in which the execution of a plan that starts with the external action `@blockworld(pickup(),L)` fails. Such an action fails in case there is no bomb to be picked up; it is possibly removed by another agent. The rule states that then the plan should be replaced by a plan in which the agent updates its beliefs about the location of the bombs. Note that in this case the rest of the original plan denoted by `REST` is dropped, as it is not used anymore within the rule.

```
PR-rules:
  @blockworld( pickup(), L ); REST <- true |
  {
    @blockworld( sensePosition(), POS );
    B(POS = [X,Y]);
    RemoveBomb( X, Y )
  }
```

Code fragment 2.9: An example of a plan repair rule of `harry`.

The execution of a plan fails if the execution of its first action fails. When the execution of an action fails depends on the type of action. The execution of: (1) a belief update action fails if the action is not specified, (2) an abstract action if there is no applicable procedural rule, (3) an external (Java) action if the environment succeeds, possibly after retrying within the specified time-out limit, (4) a belief test action if the belief expression is not derivable from the belief base, (5) a test goal action if the goal expression is not derivable from the goal base, and (6) an atomic plan section if one of its actions fails. The execution of all other actions will always be successful. When the execution of an action fails, then the execution of the whole plan is stopped. The failed action will not be removed from the failed plan such that it can be repaired by a PR-rule.

2.6 The deliberation cycle

The beliefs, goals, plans and reasoning rules form the mental states of the 2APL agent. What the agent should do with these mental attitudes is defined by means of the deliberation cycle. The deliberation cycle states which step the agent should perform next, e.g. execute an action or apply a reasoning rule. The deliberation cycle can thus be viewed as the interpreter of the agent program, as it determines which deliberation steps should be performed in which order. 2APL provides the deliberation cycle as illustrated in figure 2.2. it should be noted that in the first step of the cycle each applicable PG-rule is only applied once. This means that if a PG-rule is applied for a goal it is not applied another time during this deliberation step.

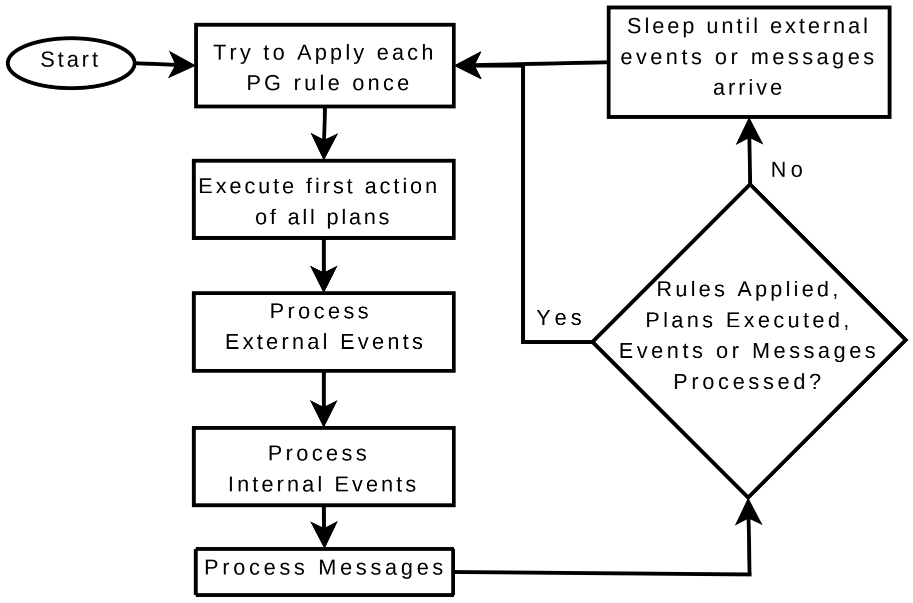

Start
Try to Apply each PG rule once
Execute first action of all plans
Process External Events
Process Internal Events
Process Messages
Rules Applied, Plans Executed, Events or Messages Processed?
Yes
No
Sleep until external events or messages arrive

Figure 2.2: The deliberation cycle of a 2APL agent.

2.7 Including files

In a multi agent system it can happen that there are more instances of one agent type. Although these agents are almost the same, they might have slightly different beliefs, goals, plans or rules. 2APL introduces an encapsulation mechanism to include agent files into other agent files. You can include another file with the `Include: filename` command. The resulting agent will be a union of the two (or more) files. In our example, for instance, both `harry` and `sally` are capable of moving to a certain location in the `blockworld`. That is to say, they have the same PC-rule `goto(X,Y)`. This rule is specified in a different file `person.2apl` which is included by both `harry` and `sally`. An included file can also include files. This makes it possible to specify specific information shared by agents in a separate file that can be included by all agents with that role.

3 The 2APL Platform

The 2APL platform is built on the top of JADE (Java Agent DEvelopment Framework). JADE is a software framework fully implemented in Java with the goal to simplify the implementation of multi-agent systems through a middleware that complies with the FIPA specifications (`www.fipa.org`) and through a set of graphical tools that supports the debugging and deployment phases. In this chapter we will briefly discuss the tools that can be used in the 2APL platform (a more extensive description can be found at `jade.tilab.com`). Besides the tools from the JADE platform the 2APL platform provides some 2APL specific tools. In this chapter we describe these tools which can be accessed through the toolbar buttons of the 2APL platform. The list of these buttons is illustrated below. The tools will be explained in the following subsections.

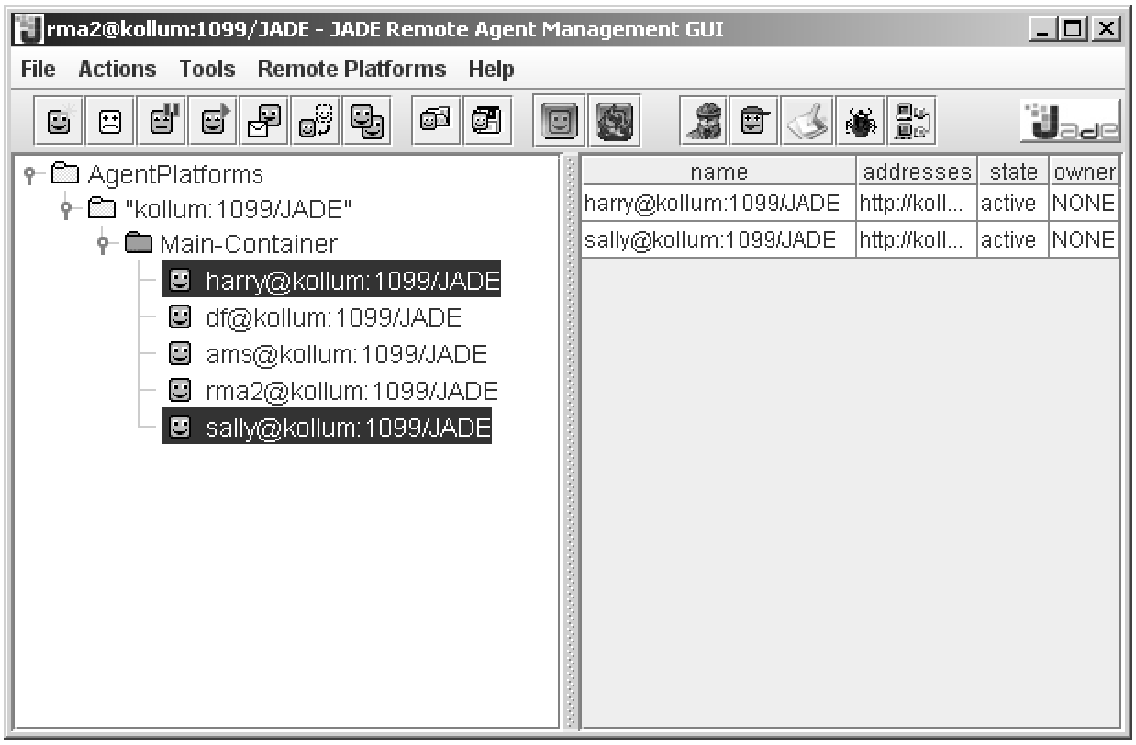
JADE tools	2APL specific tools
JADE RMA	message agent
JADE sniffer	FGDC (Flexible Graphical Deliberation Cycle)
JADE introspector	

3.1 JADE tools

The current 2APL platform comes with three JADE tools: the RMA tool, sniffer tool, and the introspector tool. The *RMA (Remote Monitoring Agent)* (figure 3.1) tool can be used to control the life cycle of the agent platform and of all the registered agents.

The *sniffer* (figure 3.2) can be used for tracking messages exchanged in a JADE based environment. You can select agents to be sniffed. Every message directed to this agent(s) or coming from this agent(s) is tracked and displayed in the sniffer window.

The *introspector* (figure 3.3) allows to monitor and control the life-cycle of a running agent and its exchanged messages, both the queue of sent and received messages. It allows also to monitor the queue of behaviours, including executing them step-by-step.



name	addresses	state	owner
harry@kollum:1099/JADE	http://koll...	active	NONE
sally@kollum:1099/JADE	http://koll...	active	NONE

Figure 3.1: The Remote Monitoring Agent tool window

3.2 The FGDC tool

The FGDC (Flexible Graphical Deliberation Cycle) tool (figure 3.4) can be used to visualize and program the deliberation cycle of an agent. The agent's deliberation cycle is used to determine what the agent should do next, e.g., execute an action, or select a plan to reach a goal.

By clicking on the `FGDC` button of the toolbar, an extra tab will become available in the agent panel. The left panel of the FGDC window shows the deliberation cycle of the currently selected agent. In this panel, the rectangles represent the steps, whereas the arrows represent the sequence in which the steps are to be executed. The panel on the right shows the predefined deliberation steps that can be used. A deliberation step can be added to the deliberation cycle by double clicking on a predefined deliberation steps. The selected predefined deliberation step will then appear in the left panel. Deliberation steps can be linked by arrows by right clicking on the source, dragging the arrow to its destination and dropping it there. The steps and arrows can be deleted by right clicking on them. Some appearance settings of the FGDC can be changed in the `FGDC settings` window that can be accessed via the `settings` button located on the right panel. Any modification of the deliberation cycle will be lost by reloading the agent.

3.3 The state tracer

The state tracer (figure 3.5) can be used for showing the execution trace of an agent. Each state shows the beliefs, plans, goals, and log of the agent as a result of executing one deliberation step. With the state tracer it is possible to browse through the history of the execution of the agent.

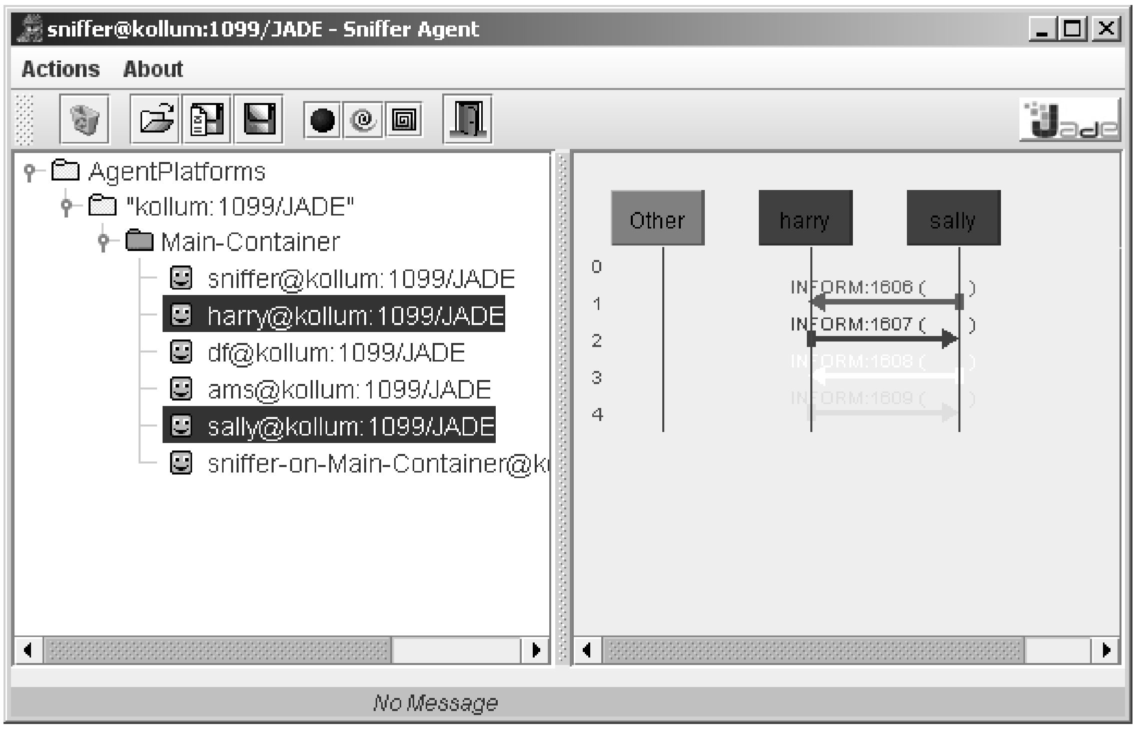
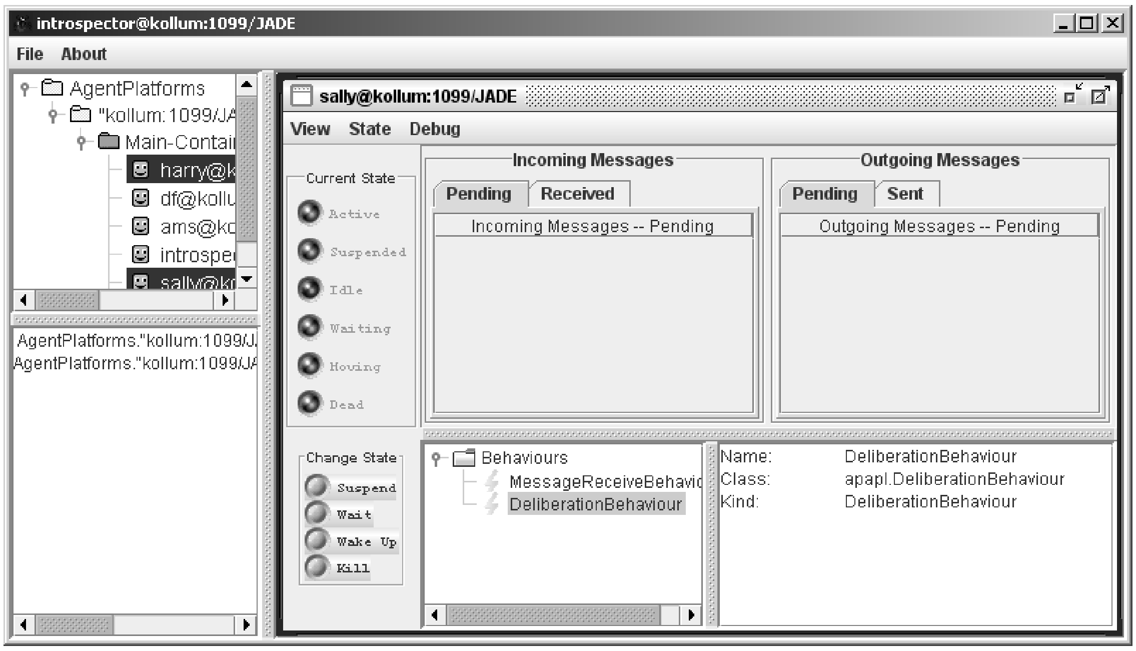


Figure 3.2: The sniffer tool window

The state tracer can be accessed by clicking on the `state tracer` tab in the agent panel. With the buttons in the upper part of the tab you can navigate through the state trace. You can select how many states (one, two, or three) to show on one screen and whether to show the beliefs, plans, goals, and log with the menu on the lower part of the tab.

3.4 The message agent

The message agent (figure 3.6) can be used for sending messages to running agents. The receiver, content, performaitve, language and ontology as explained in 2.3 should be specified. The message agent can be accessed by clicking on the `message agent` button in the toolbar.



AgentPlatforms."kollum:1099/J
AgentPlatforms."kollum:1099/JA

Name: DeliberationBehaviour
Class: apapl.DeliberationBehaviour
Kind: DeliberationBehaviour

Figure 3.3: The introspector tool window

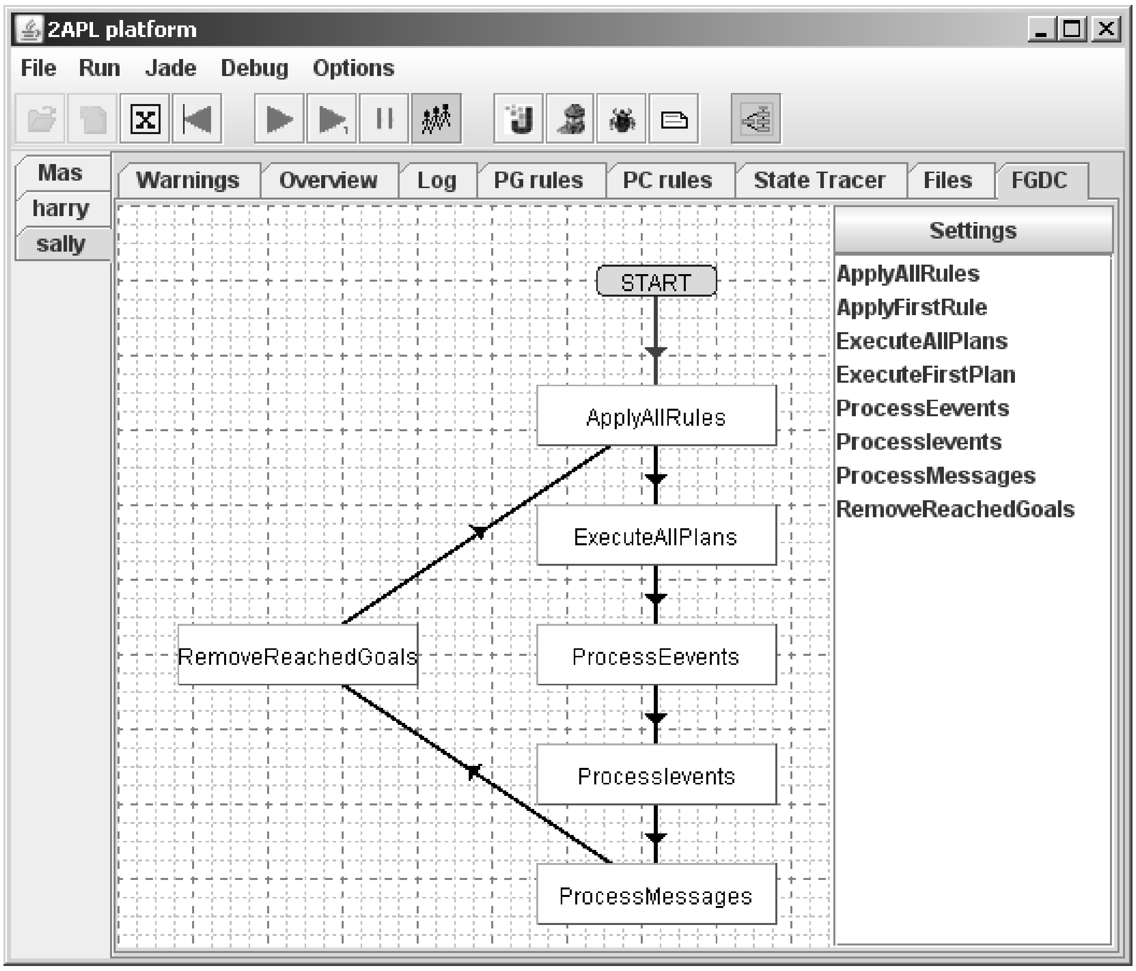
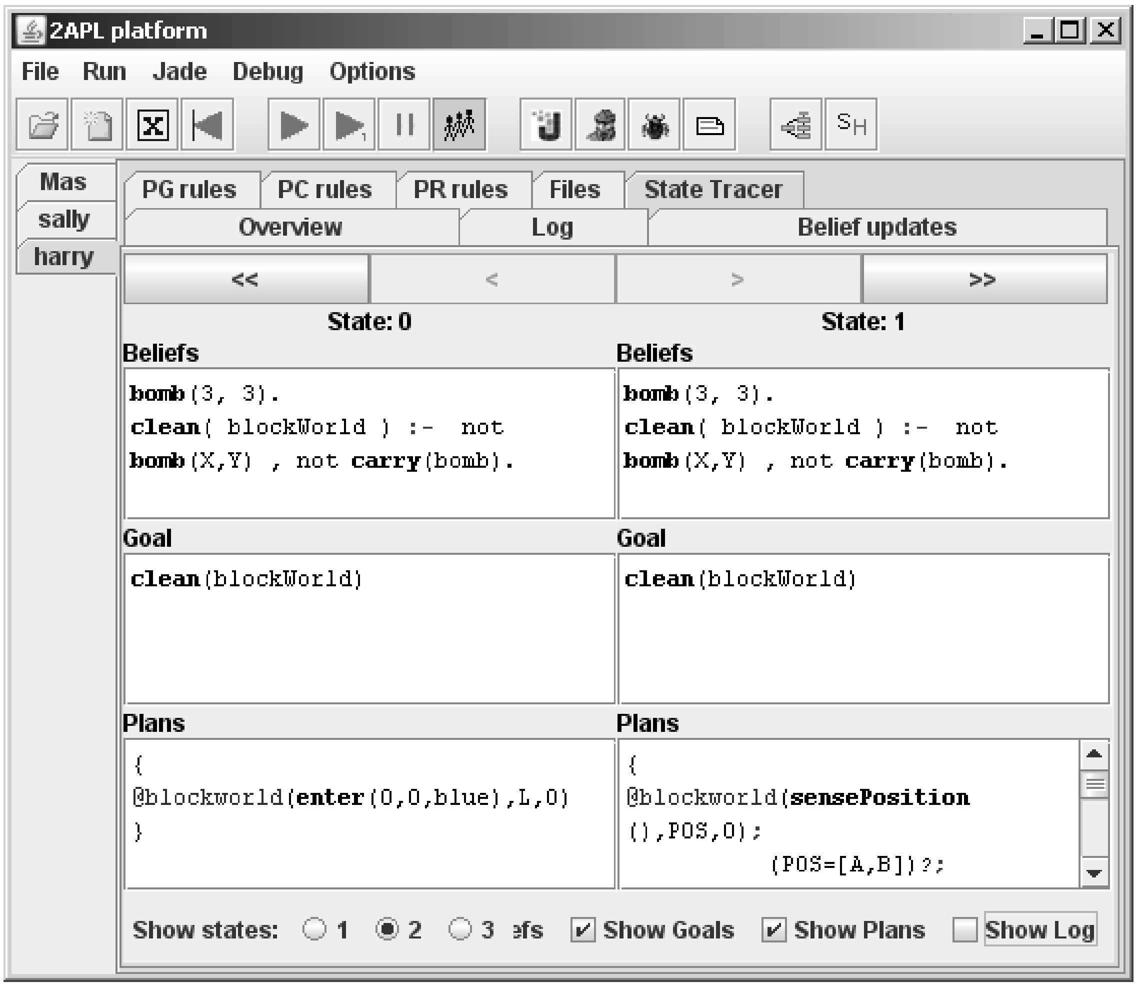


Figure 3.4: The Flexible Graphical Deliberation Cycle tool window



State: 0

Beliefs

```
bomb(3, 3).
clean( blockWorld ) :-  not
bomb(X,Y) , not carry(bomb).
```

Goal

```
clean(blockWorld)
```

Plans

```
{
@blockworld(enter(0,0,blue),L,0)
}
```

State: 1

Beliefs

```
bomb(3, 3).
clean( blockWorld ) :-  not
bomb(X,Y) , not carry(bomb).
```

Goal

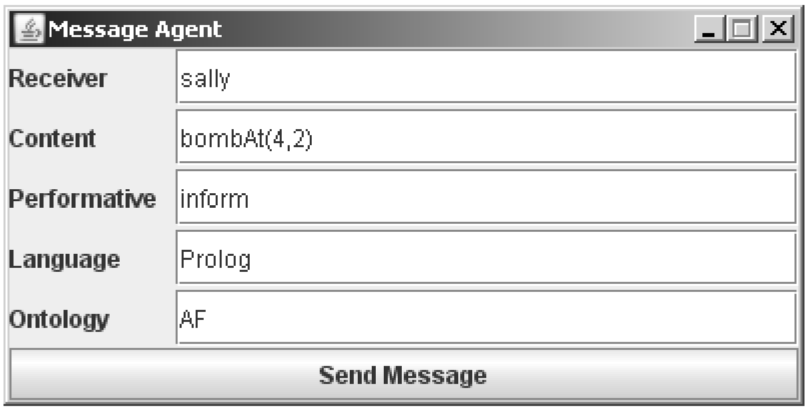
```
clean(blockWorld)
```

Plans

```
{
@blockworld(sensePosition
(),POS,0);
          (POS=[A,B])?;
```

Show states: ○ 1 [✓ ● 2] ○ 3 efs [✓ Show Goals] [✓ Show Plans] ☐ Show Log

Figure 3.5: The state tracer tool window



Receiver	sally
Content	bombAt(4,2)
Performative	inform
Language	Prolog
Ontology	AF

Figure 3.6: The message agent tool window

4 2APL Environments

2APL provides a couple of predefined external environments which can be directly used. 2APL also allows users to implement their own environments using the Java programming language. The implementation of an environment in Java is a matter of inheriting from the Environment class provided by the 2APL framework. External actions that can be performed in this environment are then to be implemented as Java methods that take some predefined 2APL data types (like lists and numbers) as input parameters and return such a 2APL data type. Also the failure of actions and the sending of events can be defined in a straightforward manner. The next section explains how to implement your own external environment. The sections after that provide an overview of the predefined environments.

4.1 Implementing environments

Each environment is implemented as a class `Env` extending the `Environment` class located in the `apapl` package. Naming of the environment is done by means of defining a package in which this `Env` is located. The name of the environment will thus be the same as the package name. Environments are always stored in a jar file. The name of the jar file containing the environment package must have the same name as the package. This enables 2APL to locate the environment. For example, the environment `exampleEnv` is implemented as a class `Env` that should be located in the package `exampleEnv` in a jar file with the name `exampleEnv.jar`.

Each environment inherits some methods from the `apapl.Environment` class that can be overridden or used to specify environment specific behaviour. These are listed below:

`addAgent(String agentname)` This method is invoked when an agent using this environment is initiated. The default implementation of this method is empty. This method can be overridden to, for example, let the environment keep track of the agents that are operating in the environment. The name of the agent uniquely identifies the agent.

`removeAgent(String agentname)` This method is invoked when an agent operating in this environment is killed. The default implementation of this method is empty.

`throwEvent(APLFunction event, String... receivers)` This method can be used to send events to the agents that are participating in the environment. The event specified by `event` is sent to all agents in the argument list. If no agents in the argument list are specified, the event will be sent to all agents.

2APL agents can use different types of terms such as numbers, lists, and constant identifiers. Recall that, for instance, external actions can take these terms as argument. These arguments are translated by the 2APL framework into a Java class, such that they can be used in the environment implementation. Each 2APL term type thus has a corresponding Java class inherited from the class `apapl.data.Term` as its counterpart. All the term types that can be used in the environment are explained below:

`APLIdent`	corresponding to a constant identifier.
methods:	`APLIdent(String name)`
	constructs an instance of `APLIdent` with name `name`.
	`toString()`
	returns a string with the name of this identifier.
`APLNum`	corresponding to a number.
methods:	`APLNum(int val)`
	constructs an instance of `APLNum` with value `val`.
	`APLNum(double val)`
	constructs an instance of APLNum with value `val`.
	`toInt()`
	returns the value of this number as an integer.
	`toDouble()`
	returns the value of this number as a double.
`APLFunction`	corresponding to a function term $\texttt{f}(\texttt{term}_1, ..., \texttt{term}_n)$.
methods:	`APLFunction(String name, Term... params)`
	constructs an instance of `APLFunction` with name `name` and the parameters as specified by the (optional) arguments `params`.
	`APLFunction(String name, ArrayList<Term> params)`
	constructs an instance of `APLFunction` with name `name` and the parameters as specified by the list `params`.
	`getName()`
	returns a `String` with the name of this function.
	`getParams()`
	returns an `ArrayList<Term>` with all the parameters of this function.
`APLList`	corresponding to a list of terms.
methods:	`APLList(Term... items)`
	constructs an instance of `APLList` wich contains the items as specified by the (optional) argument `items`.
	`APLList(LinkedList<Term> items)`
	constructs an instance of `APLList` which contains the items as specified by the list `items`.
	`toLinkedList()`
	returns a `LinkedList<Term>` containing the items of the list.

Next we explain how to implement the different parts of the environment, like external actions, throwing events, and failure of actions. The examples below are taken from the `Env.java` file that implements the `exampleEnv` environment. This environment can be found in the `examples` directory of the standard 2APL distribution.

Sending events

As explained above, external events can be sent by the use of the method `throwEvent(APLFunction event, String... receivers)` as defined in the `Environment` class, where `event` represents the event being sent and should be of the type `APLFunction`. The code fragment below shows an example that sends an event with the contents of a textbox `et`. The optional argument receivers is left blank, so the event is sent to all agents that are present in the environment.

```
APLIdent txt = new APLIdent( et.getText() );
APLFunction event = new APLFunction( "myevent", txt );
throwEvent(event);
```

Defining external actions

Recall that in 2APL external actions are expressions that are of the form `@Env( ActionName, Return, Time-out)`, where `Env` is the name of the agent's environment implemented as a Java class, and the parameter `ActionName` is of the form `action_name`(`term`$_1$,...,`term`$_n$). The latter should be implemented in the `Env` class by a method of the form:

`public Term action_name(String agent, Term term`$_1$`,...,Term term`$_n$`)`

Whenever the agent performs an external action this method is invoked by the 2APL platform. The first argument of such a method is always a string representing the name of the agent that executes this action. Note that the rest of the (optional) arguments are of type `Term` and can thus be of any subtype as explained above. It is possible to return a value from the execution of an external action back to the 2APL agent. This can be implemented by returning an object also of type `Term` in the java method representing the external action. The value being returned is then bound to the `Return` variable provided by the agent program. The return type of the method can also be `void` in case the external action does not return a value. Below an example of an external action that displays the first argument as text in a textbox `mt` is shown. The next subsection shows a more complicated example in which a value is returned.

```
public void say(String agent, APLIdent text)
{
  mt.setText(agent+" says: "+text);
}
```

Action failure

When an agent performs an external action, this action might fail. An external action then becomes eligible for repairment by a PR-rule. When exactly an external action fails is to be defined by the environment programmer. This is done by throwing an `ExternalActionFailedException`. The following code fragment shows an external action that calculates the sum of a list of numbers. In case one of the elements of the list is not a number, this action fails. The string provided to the constructor of the exception is displayed in the log of the agent in case this exception is thrown.

```
public APLNum sumOfList(String agent, APLList l)
throws ExternalActionFailedException
{
   LinkedList<Term> ll = l.toLinkedList();

   double sum = 0;
```

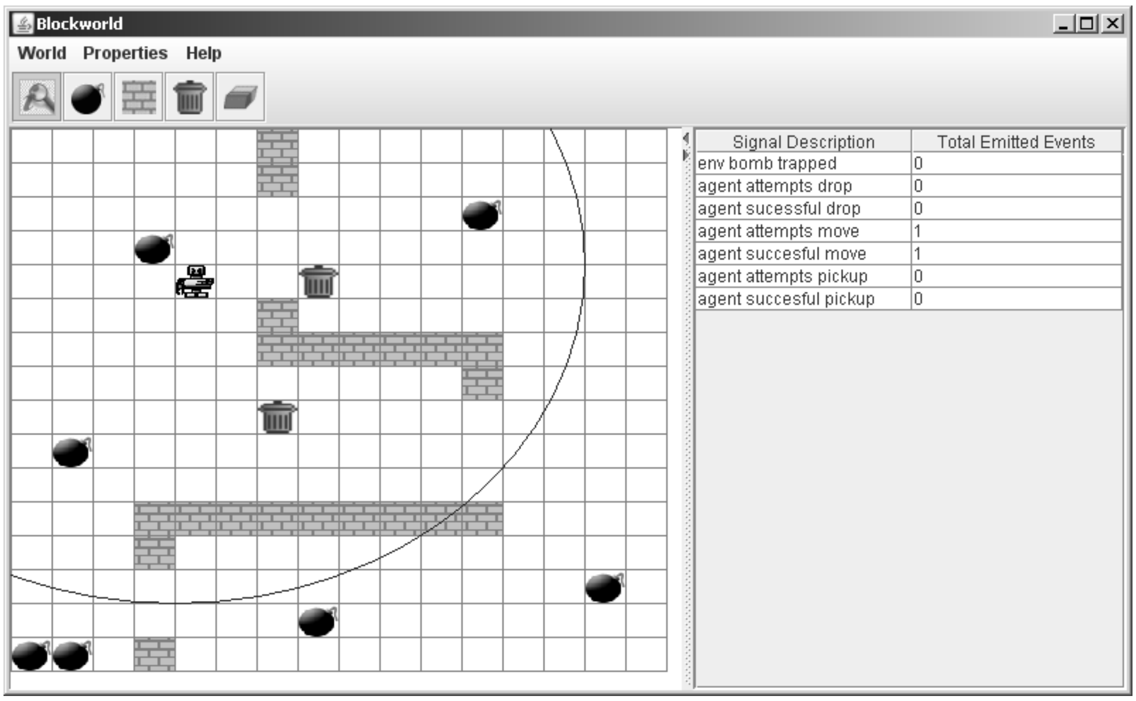
```
    for( Term e: ll )
    {
      if( e instanceof APLNum )
      { sum = sum + ((APLNum)e).toDouble();
      }
      else
      {
      throw( new ExternalActionFailedException(e.toString()
              + " is not a number.") );
      }
    }

     return( new APLNum( sum ) );
  }
```

4.2 The blockworld environment

The `blockworld` is an external environment in which agents can perform actions. The `blockworld` consists of a $n \times n$ world where agents can move in four directions (north, south, east and west). The world can contain bombs, stones, and traps. Agents can pickup and drop bombs. When a bomb is dropped in a trap, the bomb is destroyed. Figure 4.1 shows an example instance of the `blockworld` with one agent located in it. You can add and delete bombs, walls (stones), and traps. When a bomb is added within the sensing range of an agent a `bombAt` event is sent to this agent. You can also select an agent. When an agent is selected, the information about its actions and events is shown in the panel on the right. You can change some settings like the size of the world, and sensing range of the agent via the `properties` menu. The world can be saved and loaded via the `world` menu. An agent can perform the following actions in the `blockworld`. Note that each external action can be extended with the optional time-out argument as explained in chapter 2.

action name:	enter
parameter(s):	`X` - X position
	`Y` - Y position
	`C` - A constant representing a color.
return value(s):	none
fails if :	agent has already entered
	the position is outside the world
	the position is occupied by another agent or wall
example:	`@blockworld( enter`(5,5,`red`), `S`, 1)

description:
Tries to inserts an agent in the `blockworld` at a given position (X, Y). The options for the color are: army, blue, gray, green, orange, pink, purple, red (the color which will also be selected for invalid constants), teal, and yellow.



Signal Description	Total Emitted Events
env bomb trapped	0
agent attempts drop	0
agent sucessful drop	0
agent attempts move	1
agent succesful move	1
agent attempts pickup	0
agent succesful pickup	0

Figure 4.1: An example instance of the blockworld with one agent located in it.

action name:	sensePosition
parameter(s):	none
return value(s):	`[X,Y]`
example:	`@blockworld( sensePosition(), P)`
description:	

Senses the position of the agent within the `blockworld`. If the agent is not in the `blockworld` there will be no substitution for the return value, otherwise it will be the coordinates of the agent at `[X,Y]`.

action name:	north
parameter(s):	none
return value(s):	none
fails if:	the position is outside the world
	the position is occupied by another agent or wall
example:	`@blockworld( north(), S)`
description:	

Tries to move the agent through the `blockworld` to the position above $(Y-1)$ the current position.

action name:	south
parameter(s):	none
return value(s):	none
fails if:	the position is outside the world
	the position is occupied by another agent or wall
example:	`@blockworld( south(), S)`
description:	

Tries to move the agent through the `blockworld` to the position above $(Y+1)$ the current position.

action name:	east
parameter(s):	none
return value(s):	none
fails if:	the position is outside the world
	the position is occupied by another agent or wall
example:	`@blockworld( east(), S)`
description:	

Tries to move the agent through the `blockworld` to the position above $(X + 1)$ the current position.

action name:	west
parameter(s):	none
return value(s):	none
fails if:	the position is outside the world
	the position is occupied by another agent or wall
example:	`@blockworld( west(), S)`
description:	

Tries to move the agent through the `blockworld` to the position above $(X - 1)$ the current position.

action name:	senseBombs
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseBombs(), BL)`
description:	

Gives a list of all bombs in the sense range. The first element of the sublist (each bomb description) is always 'default', the second is the X position and the third the Y position.

action name:	senseAllBombs
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseAllBombs(), BL)`
description:	

Gives a list of all bombs, also the ones outside the sense range. The first element of the sublist (each bomb description) is always 'default', the second is the X position and the third the Y position.

action name:	pickup
parameter(s):	none
return value(s):	none
fails if:	the agent already carries a bomb
	there is no bomb to pickup
example:	`@blockworld( pickup(), S)`
description:	

Tries to pickup a bomb.

action name:	drop
parameter(s):	none
return value(s):	none
fails if:	the agent doesn't carry a bomb
	there is already a bomb in the position

The agent tries to drop the bomb it carries.

action name:	senseAgent
parameter(s):	none
return value(s):	`[[A1, X1, Y1], [A2, X2, Y2], ... ]`
example:	`@blockworld( senseAgent(), AL)`

description:
Gives a list of all agents in the sense range. The first element of the sublist (each agent description) is the name (A) of the agent, the second is the X position and the third the Y position.

action name:	senseAllAgent
parameter(s):	none
return value(s):	`[[A1, X1, Y1], [A2, X2, Y2], ... ]`
example:	`@blockworld( senseAllAgent(), AL)`

description:
Gives a list of all the agents independent of the sense range.

action name:	senseStones
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseStones(), SL)`

description:
Gives a list of all stones in the sense range. The first element of the sublist (each stone description) is always 'default', the second is the X position and the third the Y position.

action name:	senseAllStones
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseAllStones(), SL)`

description:
Gives a list of all stones, also the ones outside the sense range. The first element of the sublist (each stone description) is always 'default', the second is the X position and the third the Y position.

action name:	senseTraps
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseTraps(), TL)`

description:
Gives a list of all traps in the sense range. The first element of the sublist (each trap description) is always 'default', the second is the X position and the third the Y position.

action name:	senseAllTraps
parameter(s):	none
return value(s):	`[[default, X1, Y1], [default, X2, Y2], ... ]`
example:	`@blockworld( senseAllTraps(), TL)`

description:
Gives a list of all traps, also the ones outside the sense range. The first element of the sublist (each trap description) is always 'default', the second is the X position and the third the Y position.

action name:	getSenseRange
parameter(s):	none
return value(s):	`[SenseRange]`
example:	`@blockworld( getSenseRange(), SR)`
description:	

Returns the sense-range of the agent.

action name:	getWorldSize
parameter(s):	none
return value(s):	`[Width, Height]`
example:	`@blockworld( getWorldSize(), WS)`
description:	

Returns the size of the `blockworld`, the first parameter is the width and the second the height.

4.3 The msAgent environment

The msAgent environment provides an interface to Microsoft Agents. Microsoft Agents are interactive, animated characters. For more information about Microsoft Agent visit:
`http://msdn2.microsoft.com/en-us/library/ms695784.aspx`.

This section briefly discusses which functionalities the msAgent environment provides. Note that msAgent will only function under Windows. Before first usage of msAgent please visit the following page: `http://www.microsoft.com/msagent/downloads/user.asp` From there, install: Microsoft agent character files, the American-English text-to-speech engine and sapi 4.0 runtime support.

action name:	create
parameter(s):	`S` - character name ("Genie","Merlin","Peedy" or "Robby")
return value(s):	none
example:	`@msagent(create("Genie"), X, 0)`
description:	

Creates an msAgent.

action name:	show
parameter(s):	`D` - show fast (1) or let the agent apear (0)
return value(s):	none
example:	`@msagent(show(0), X, 0)`
description:	

Makes the agent visible. Can be used after, for example, a hide action.

action name:	hide
parameter(s):	`D` - hide fast (1) or let the agent disapear (0)
return value(s):	none
example:	`@msagent(hide(0), X, 0)`
description:	

Hides the agent.

action name:	speak
parameter(s):	`S` - speech output expression
return value(s):	none
example:	`@msagent(speak("Hello my name is Genie."), X, 0)`

description:
The agent pronounces the text denoted by the speech output expression. Special tags can be used to manipulate the output. More information can be found at: `http://msdn2.microsoft.com/en-us/library/ms699259.aspx`

action name:	think
parameter(s):	`S` - string representing the speech output expression
return value(s):	none
example:	`@msagent(think("Where is my lamp..."), X, 0)`

description:
Displays text in a thought bubble.

action name:	play
parameter(s):	`S` - the animation to play
return value(s):	none
example:	`@msagent(play("Search"), X, 0)`

description:
Plays an animation. The available animations for the different characters can be found at: `http://msdn2.microsoft.com/en-us/library/ms695821.aspx`

action name:	moveTo
parameter(s):	`X` - X position on the desktop to move to
	`Y` - Y position on the desktop to move to
	`M` - the speed at which the agent will move
return value(s):	none
example:	`@msagent(moveTo(300, 100, 10), X, 0)`

description:
Move an agent to the given X, Y position at a certain speed.

action name:	gestureAt
parameter(s):	`X` - X position to gesture at
	`Y` - Y position to gesture at
return value(s):	none
example:	`@msagent(gestureAt(300,100), X, 0)`

description:
Gesture at the X, Y position.

Bibliography

[1] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade - a java agent development framework. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 5. Springer-Verlag, 2005.