2APL

A Practical Agent Programming Language

User Guide

Version 1.0

1 Introduction

The 2APL Platform and its corresponding Eclipse plug-in editor are developed to support the implementation and execution of multi-agent systems programmed in 2APL programming language [3, 4, 5]. 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 load, run, and monitor the execution of 2APL multi-agent programs using several tools, such as different execution modes, state tracer, log window, and the list of exchanged messages. 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. In order to facilitate the implementation of 2APL multi-agent programs, a 2APL editor plug-in for Eclipse is developed. Using this editor it is possible to load, run, and monitor 2APL multi-agent programs directly from Eclipse.

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.

1.2 Getting 2APL Software

In order to use the 2APL Platform and/or its corresponding editor follow the next steps:

1. Go to the 2APL webpage using the following URL:

   `http : //apapl.sourceforge.net/`

   and select the `Downloads` option.

2. There are three download options available. You can download only the 2APL platform without the 2APL Eclipse plug-in editor, only the 2APL Eclipse plug-in editor without the 2APL platform, or the full package consisting of 2APL platform, 2APL Eclipse plug-in editor, and the Eclipse itself.

  - The 2APL platform, without its corresponding Eclipse plug-in editor, can be downloaded by selecting the option `2APL Platform`. This option allows loading, running, and monitoring 2APL multi-agent programs.
  - The 2APL Eclipse plug-in editor, without the 2APL platform, can be downloaded by selecting the option `2APL Eclipse Plug-in Editor`. This editor supports writing and editing 2APL multi-agent programs by recognizing and highlighting the 2APL syntax. For this option, the Eclipse software should be downloaded from its official website. Eclipse can be configured to start the 2APL platform, load a 2APL multi-agent program into it, and run the program (see section 1.5 for further instruction). To run a 2APL multi-agent program, the 2APL platform (previous item) should be downloaded.
  - The full integrated package can be downloaded by selecting the option `2APL Full Package: 2APL Platform, Eclipse Plug-in Editor, and Eclipse`.

3. Extract the contents of the downloaded document into a directory. In the following, we assume that this directory is named `2APL`.

The downloaded 2APL platform is a zipped document, called `2APL.zip`, containing the file `2apl.jar` and two folders `lib` and `examples`. This file `2apl.jar` includes all the class files of the 2APL Platform, the `lib` directory contains all necessary files to run the 2APL platform (e.g., the class files of JIProlog[1]), and the `examples` folder contains some multi-agent systems programmed in 2APL. The example directory includes a multi-agent system program, called `harry and sally`. This example is about two agents, `harry` and `sally`, who are located in the so-called `blockworld` environment. The `blockworld` environment is a $n \times n$ grid, which can contain other agents, bombs, and dustbins in which bombs can be thrown away. `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 more details. In the sequel, we will explain the 2APL programming language and its platform in terms of this example. The file `blockworld.jar` is the implementation of the blockworld environment. The details of the `blockworld` environment will be explained in chapter 3. In the example directory, there are more examples of 2APL multi-agent programs. Each example is explained in a readme document that is included in the directory of the example.

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, running and monitoring the execution of a 2APL multi-agent system program. The development of 2APL multi-agent programs is supported by an editor which is an Eclipse plug-in. This editor in explained in detailed in section 1.5.

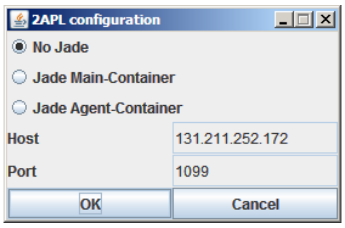
The 2APL platform can be started in:

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

[1]JIProlog is a Java based Prolog reasoning engine that is used by the individual 2APL agents to represent and reason about their beliefs



[✓ No Jade] Jade Main-Container Jade Agent-Container

Host 131.211.252.172

Port 1099

Figure 1.1: The window with 2APL configuration options.

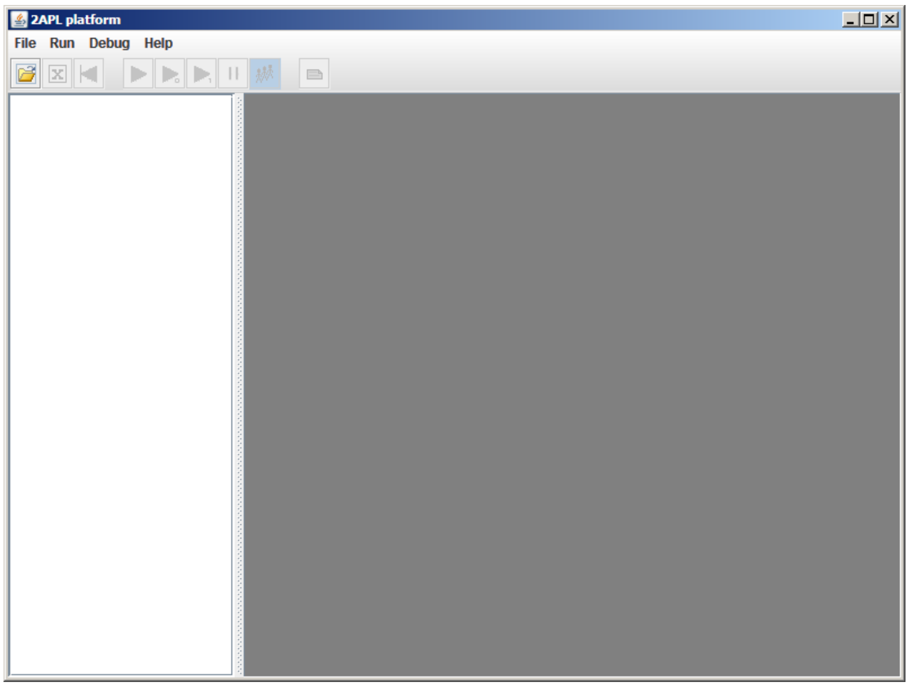


Figure 1.2: The main 2APL user interface.

As illustrated in figure 1.1, you need to select a configuration option. The first option allows you to use 2APL platform in a stand-alone mode and the last two options start 2APL on the top of the Jade [2] platform. The last two options are useful when a multi-agent program are to be run in a distributed manner on different machines. In this user guide, we focus on the stand-alone run mode of the 2APL platform.

After successful startup the window as depicted in figure 1.2 should appear. This is the main user interface of the 2APL platform that allows to load, run, and monitor the execution of a multi-agent system programmed in 2APL.

To load an example multi-agent system, you should perform the following steps:

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 `2APL\examples\harry and sally` and select the file named `harrysally.mas`.

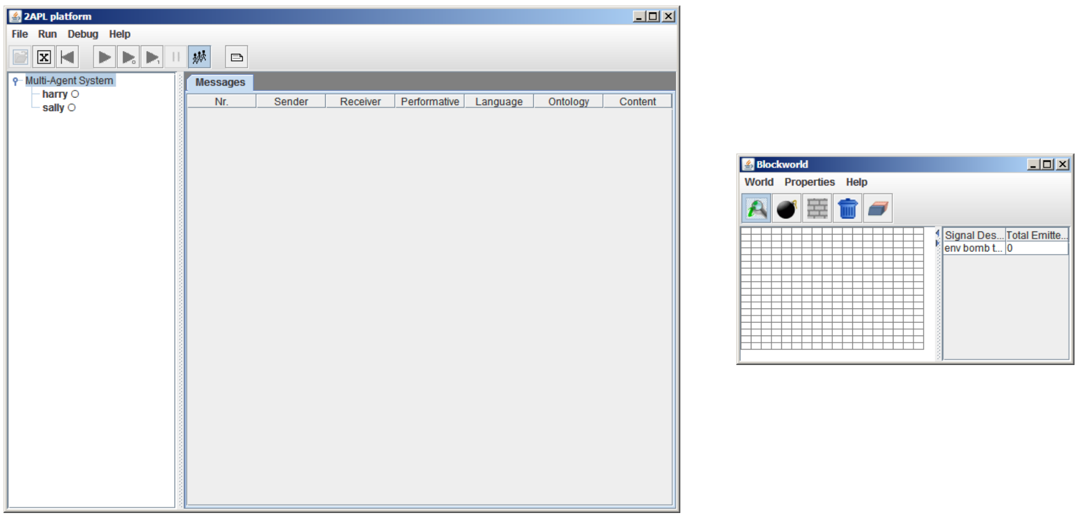


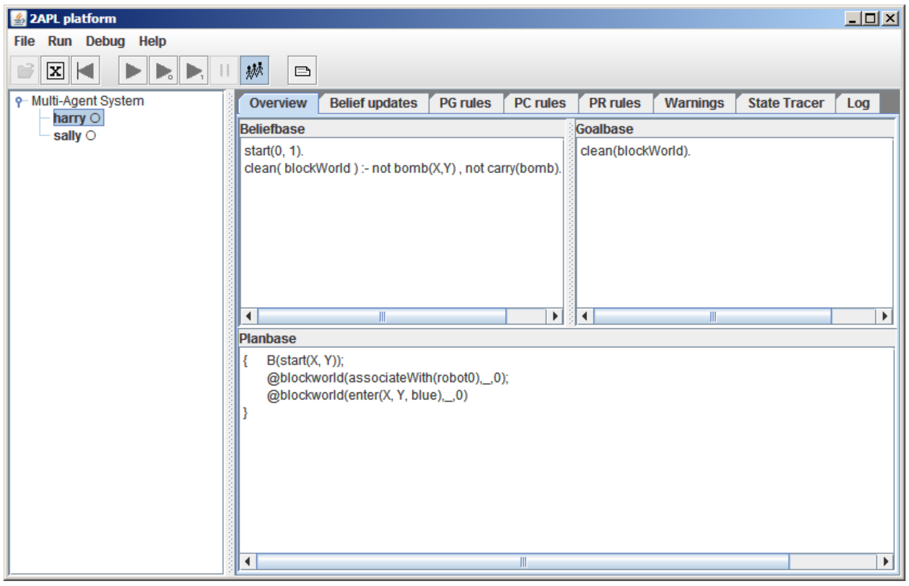
Figure 1.3: The main 2APL user interface loaded with `harry and sally` multi-agent program (left), and the `blockworld` user interface (right).

The file with the `.mas` extension specifies the multi-agent system by indicating which agents should be created, which `.2apl` files initialize the agents, which environments they can access and which `.jar` file implements the environment. In this case, two agents `harry` and `sally` will be created that act in the `blockworld` environment. The agents are initialized by `harry.2apl` and `sally.2apl`, respectively. The environment is implemented by `bloackworld.jar`. Note that the `blockworld` environment is configured by parameters that determine the size of the grid and the number of agents that will appear in the grid.

After the file has been loaded the main 2APL user interface as well as the interface of the `blockworld` appears[2]. These two interfaces are depicted in figure 1.3. The left panel of the window allows you to toggle between the multi-agent system tab `Multi-Agent System` 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 `Multi-Agent System`, one can only see in the right panel the `Messages` tab to observe the messages that are exchanged between the agents when they are executed.

In case an individual agent has been selected in the left panel, one can select different tabs in the right panel to observe various state ingredients of the selected 2APL agent. In this example, and as illustrated in figure 1.4, the `Overview` tab of the agent `harry` is selected. This tab enables the user to observe the mental state (beliefs, goals, and plans) of the agent `harry`. One can also select 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 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, the `Warnings` tab to see system warnings, the State Tracer tab to monitor the subsequent mental states of the agent `harry` when it is executed, and the `Log` tab to observe the agent's execution log. 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.

[2]The `blockworld` environment is implemented in terms of a user interface that presents a grid. Please note that an environment does not necessarily need to generate a user interface.



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

```
clean(blockWorld).
```

```
{    B(start(X, Y));
     @blockworld(associateWith(robot0),_,0);
     @blockworld(enter(X, Y, blue),_,0)
}
```

Figure 1.4: The main 2APL user interface when selecting agent `harry` in the left panel.

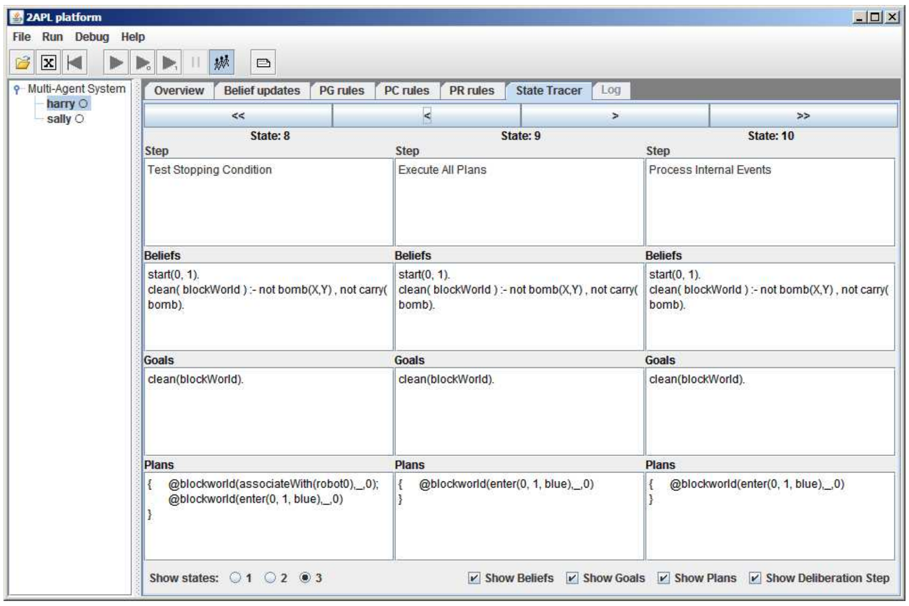
A 2APL program can be executed in different modes by means of the buttons on the toolbar, which can also be accessed through the `file` and `run` menu items. The meaning of these buttons is shown below.

📂	load a `mas` file
☒	close `mas` file
⏮	reload `mas` file
▶	execute agent(s) in continuous mode
▶ₒ	execute agent(s) for one deliberation cycle
▶₁	execute agent(s) in step-by-step mode
⏸	pause execution of agent(s)
👥	apply abovementioned operations to all agents
🚶	apply abovementioned operations to the selected agent
✉	send a message to an agent.

Alternatively, you may load and start the multi-agent system directly from the command line. One can also start the interpreter without a graphic user interface. To see the help on the available options execute `java -jar 2apl.jar -help` in the command line prompt.

1.4 Execution Monitoring Tools

In this section we present the tools that can be used in the 2APL platform to monitor the execution of individual agents. There are three monitoring tools: `Auto-update Overview`, `State`



State: 8	State: 9	State: 10
Step	**Step**	**Step**
Test Stopping Condition	Execute All Plans	Process Internal Events
Beliefs	**Beliefs**	**Beliefs**
start(0, 1). clean(blockWorld) :- not bomb(X,Y) , not carry(bomb).	start(0, 1). clean(blockWorld) :- not bomb(X,Y) , not carry(bomb).	start(0, 1). clean(blockWorld) :- not bomb(X,Y) , not carry(bomb).
Goals	**Goals**	**Goals**
clean(blockWorld).	clean(blockWorld).	clean(blockWorld).
Plans	**Plans**	**Plans**

{ @blockworld(associateWith(robot0),_,0);
@blockworld(enter(0, 1, blue),_,0)
}
	{ @blockworld(enter(0, 1, blue),_,0)
	}
		{ @blockworld(enter(0, 1, blue),_,0)
		}

Show states: ○ 1 ○ 2 [● 3] [✓ Show Beliefs] [✓ Show Goals] [✓ Show Plans] [✓ Show Deliberation Step]

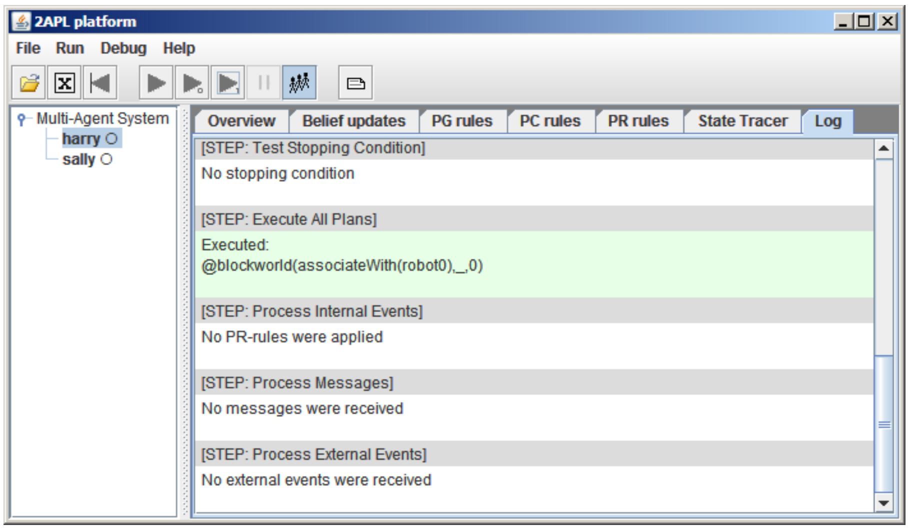
Figure 1.5: The `State Traces` tab shows the execution trace of agent `harry`.

`Tracer`, and `Log`. These tools, which can be accessed through the `Debug` toolbar button of the 2APL platform, are activated by default. The tools can be deactivated to improve the execution performance of the 2APL platform. The activation of the tools present specific information about the execution of individual agents to tabs corresponding to the tools. The tabs can be used in the right panel of the 2APL main user interface by selecting an individual agent in the left panel of the user interface.

The `Overview` tab consists of three panels called `Beliefbase`, `Goalbase`, and `Planbase`. When this tab is active, the beliefs, goals, and plans of the last execution state of each agent is presented in the `Beliefbase`, `Goalbase`, and `Planbase` panels, respectively. Figure 1.4 illustrates the use of the `Overview` tab which presents the beliefs, goals, and plans of agent `harry` from its initial state directly after loading the multi-agent program. Executing the multi-agent program will present updated information about the agent `harry`.

The `State Tracer` tab is a temporal version of the `Overview` tab and presents the beliefs, goals, and plans of agents during the execution. In particular, this tool stores the beliefs, goals, and plans of all agents during execution. Because an execution generates a trace of agents' states (beliefs, goals, and plans), the tools and its corresponding tab are called state tracer. This tool allows a user to execute a multi-agent program for a while, pause the execution, and browse through the execution of each agent. With the buttons in the upper part of the tab one can navigate through the state trace. The user 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. Figure 1.5 illustrates the use of the state trace tool for monitoring the execution of the `harry` agent. This tab shows the execution of the multi-agent program at states 8, 9, and 10. It also shows that `harry` has performed action `@blockworld(associateWith(robot0),_,0)` in state 8 of the execution.

Finally, the `Log` tool presents information about the deliberation steps of individual agents. As illustrated in Figure 1.6, the `Log` tab indicates whether `harry` has executed an action, processed an internal event, a message, or an external event. The user can browse through this window to



```
[STEP: Test Stopping Condition]
No stopping condition

[STEP: Execute All Plans]
Executed:
@blockworld(associateWith(robot0),_,0)

[STEP: Process Internal Events]
No PR-rules were applied

[STEP: Process Messages]
No messages were received

[STEP: Process External Events]
No external events were received
```

Figure 1.6: The `Log` tab presents information about deliberation steps of agent `harry`.

see which deliberation steps are preformed.

1.5 2APL Eclipse Plug-in Editor

The 2APL editor is an Eclipse Xtext based plugin that supports the editing of 2APL multi-agent programs. In this section we will describe how to install, configure and use this editor.

1.6 Requirements

You should have a basic understanding of the 2APL platform itself as described in previous section. In order to run the 2APL editor you need the latest version of Eclipse Xtext.

1.7 Getting Software and Installation

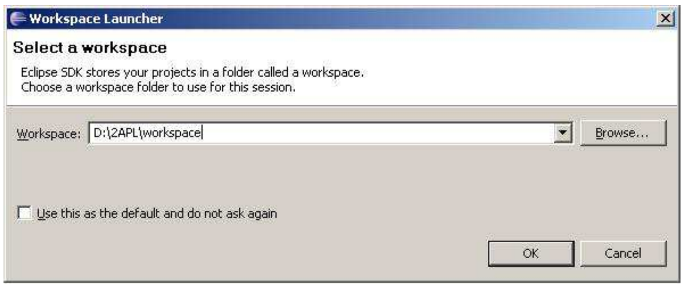
You can download the `2APL Full Package` that contains the 2APL platform, the Eclipse plug-in editor, and the Eclipse Xtext version all configured and installed (286 MB), or alternatively download all the packages separately and configure them by hand.

In order to use the total package:

1. Download the full package from the 2APL homepage using the following URL:

   `http://apapl.sourceforge.net/`

   and follow the `Downloads` option.
2. Extract the files into a directory. In the sequel, we assume that this directory is names `D:\2APL`
3. Run the `2APL Eclipse Editor` shortcut located in the directory and select the workspace in the same directory (i.e., `D:\2APL\workspace`) as illustrated in the following Figure.



Workspace Launcher

Select a workspace

Eclipse SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: D:\2APL\workspace

To manually install the 2APL editor you need to follow the next procedure:

1. Download the Eclipse Xtext Galileo 0.7.2 version from its homepage using the following URL:

`http://www.eclipse.org/Xtext/download/`

2. Extract the contents of the file into a directory. In the sequel, we assume that this directory is named `eclipse`.
3. Download the Eclipse plugin files from the 2APL website using the following URL:

   `http://apapl.sourceforge.net/wordpress`

   and follow the `Downloads` option and select `2APL Eclipse Plug-in Editor`.
4. Extract the files to the Eclipse plugin directory named `eclipse\plugins`.
5. Download a XML plugin for Eclipse using the following URL:

   `http://sourceforge.net/projects/editorxml/files/Rinzo`

6. Extract the files to the Eclipse plugin directory named `eclipse\plugins`.

1.8 Getting Started with the Editor

A 2APL multi-agent program, which we will call a 2APL project, consists of one `.mas` file and one or more `.2apl` files. This editor allows the user to create a new 2APL project, or to load and modify an existing 2APL project.

1.8.1 Editing Existing Projects

Existing projects, e.g., the multi-agent program `harry and sally` in the `2APL\examples\harry and sally`[3], can be loaded and edited using the import wizard. For importing an existing project, you should have the existing projects in the `D:\2APL\workspace` directory. Once you have your projects in this directory, choose `File -> import`. A selection window will appear. Select item `General -> Existing Projects into Workspace`, press `Next`, browse from directory containing the project files, and press `Finish`. The project will appear in the right panel. These steps are illustrated in Figure 1.7.

1.8.2 Creating and Editing a New Project based on existing Multi-Agent Program Files

This option can be used if you have already written a multi-agent program with another editor and want now to develop it further with the 2APL Eclipse editor.

1. To create a new empty project without any files, select the menu `File -> new`. In the Wizard window, select `General -> Project`, press `Next`, give the project a name, and press `Finish`. The new empty project will appear in the left panel of the Eclipse window.
2. Select the created project and right click the mouse and choose `Import`. Select in the `Select Wizard` the option `General -> File System` and click `Next`.

[3]For unix systems, path representations should be modified accordingly.

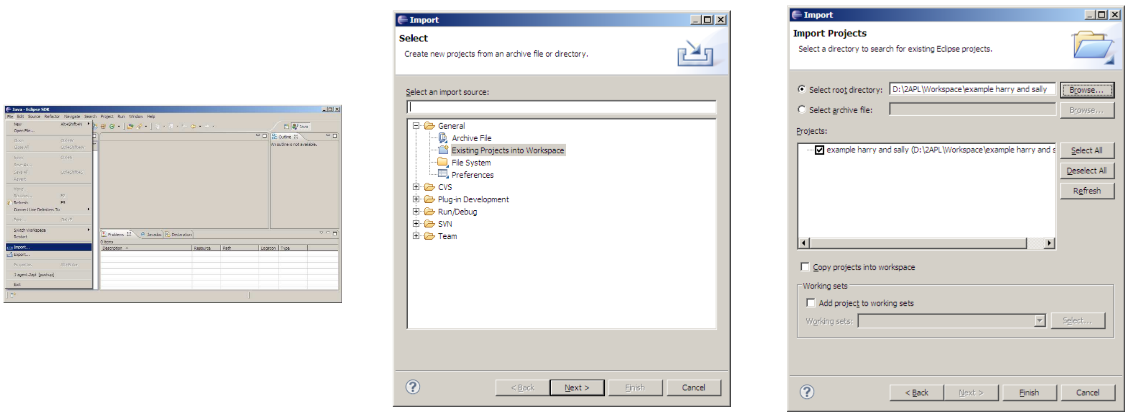

Import
Select
Create new projects from an archive file or directory.
Select an import source:
General
Archive File
Existing Projects into Workspace
File System
Preferences
CVS
Plug-in Development
Run/Debug
SVN
Team
< Back
Next >
Finish
Cancel
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
D:\2APL\Workspace\example harry and sally
Browse...
Select archive file:
Projects:
example harry and sally (D:\2APL\Workspace\example harry and s
Select All
Deselect All
Refresh
Copy projects into workspace
Working sets
Add project to working sets
Working sets:
Select...

Figure 1.7: Editing existing project.

3. Use `Browse` and select the directory in which your already written multi-agent system files are located.

4. You can now select the files to import and click `Finish`. Your files are now available in the project in the left panel of the editor.

1.8.3 Creating and Editing a New Project

In order to create a new 2APL project, the following steps should be taken.

1. Follow the `File -> New -> Project...` of the Eclipse user interface. Choose `APAPL project` by following the item `Xtext` in the select wizard and press `Next`. These steps are illustrated in Figure 1.8.

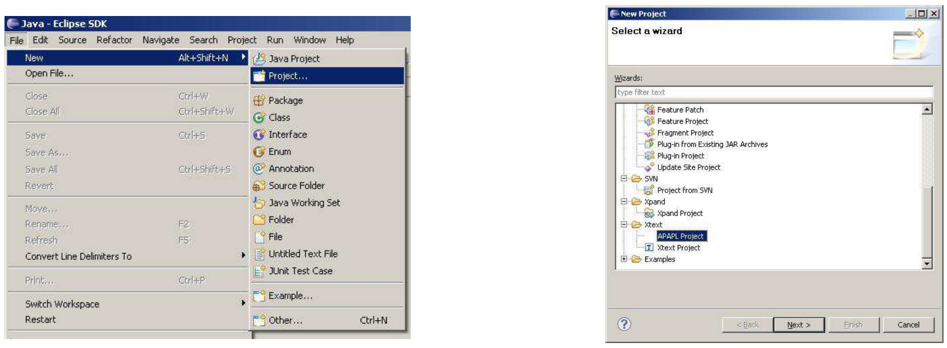
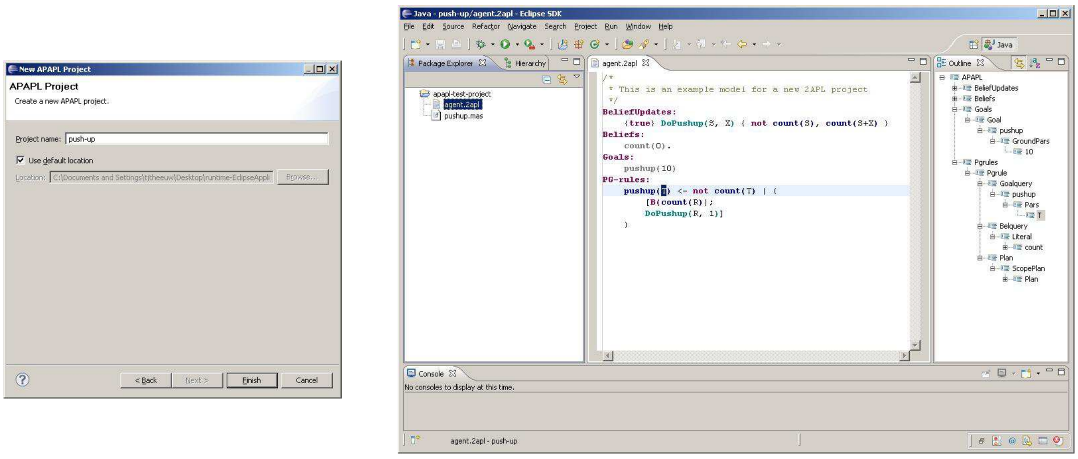


Figure 1.8: Create a new 2APL project.

2. Give the project a name (e.g. `push-up`), as illustrated in Figure 1.9(left).

3. Open the created project by double click the `push-up` item in the left panel of the Eclipse window. This opens the `push-up` project consisting of the `pushup.mas` and `agent.2apl` files. These files can be opened and edited by clicking on the files `agent.2apl` or `pushup.mas`. Figure 1.9 (right) illustrates the opened file `agent` in the 2APL editor which can now be edited.



Project name: push-up

☑ Use default location

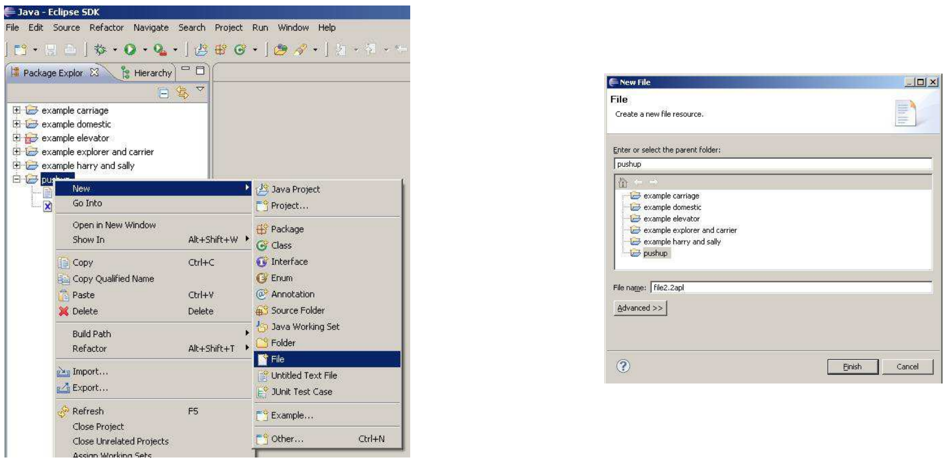
Location: C:\Documents and Settings\ttheeuw\Desktop\runtime-EclipseAppli

agent.2apl

```
/*
 * This is an example model for a new 2APL project
 */
BeliefUpdates:
    {true} DoPushup(S, X) { not count(S), count(S+X) }
Beliefs:
    count(0).
Goals:
    pushup(10)
PG-rules:
    pushup(T) <- not count(T) | {
        [B(count(R));
        DoPushup(R, 1)]
    }
```

Figure 1.9: Giving a name to a new project (left), and Editing 2APL files (right).

4. New agent programs (i.e., `.2apl` files) can be added to this project by selecting the `push-up` project, left click mouse, and follow `New -> File` menu item. Enter a file name with `.2apl` extension. This operation is illustrated in Figure 1.10. The new file is now available in the Eclipse project.



Enter or select the parent folder: pushup

File name: file2.2apl

Figure 1.10: Adding new files to a project.

1.9 Syntax coloring

To change to colors in the 2APL editor go to the `Preferences` page, under the menu `Window`, and then select `Xtext Languages`, `APAPL`, `Syntax Coloring`. Here you can change to colors for some rules. You can also restore the color settings, back to black as default, by clicking the `Restore Defaults` button. This is illustrated in Figure 1.11.

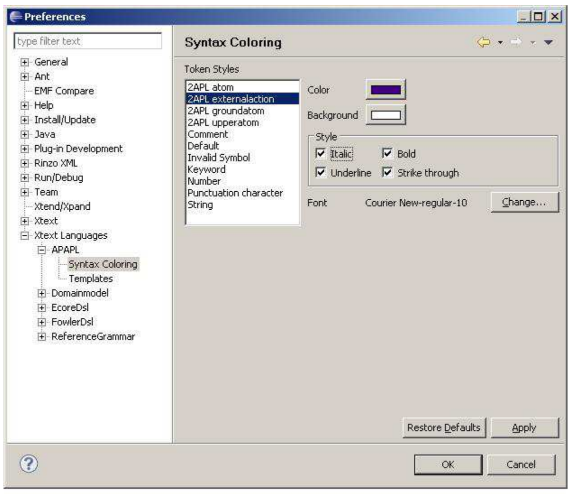
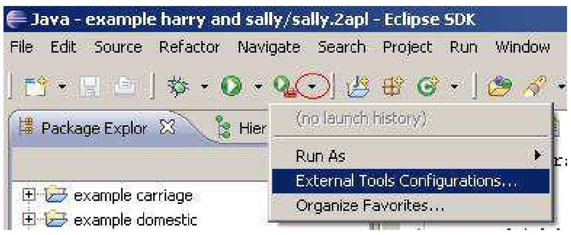


Figure 1.11: Modifying the color of 2APL syntax in the editor.

1.10 Run the project from Eclipse

To run the 2APL platform from Eclipse follow the next steps.

1. Open the `External Tools Configurations` as illustrated in the next Figure.


Java - example harry and sally/sally.2apl - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window
Package Explor
Hier
(no launch history)
Run As
External Tools Configurations...
Organize Favorites...
example carriage
example domestic

2. In the External Tools Configurations window, right click `program` and select `new`. This is illustrated in the next Figure.

3. In the configurations window, set the following values.

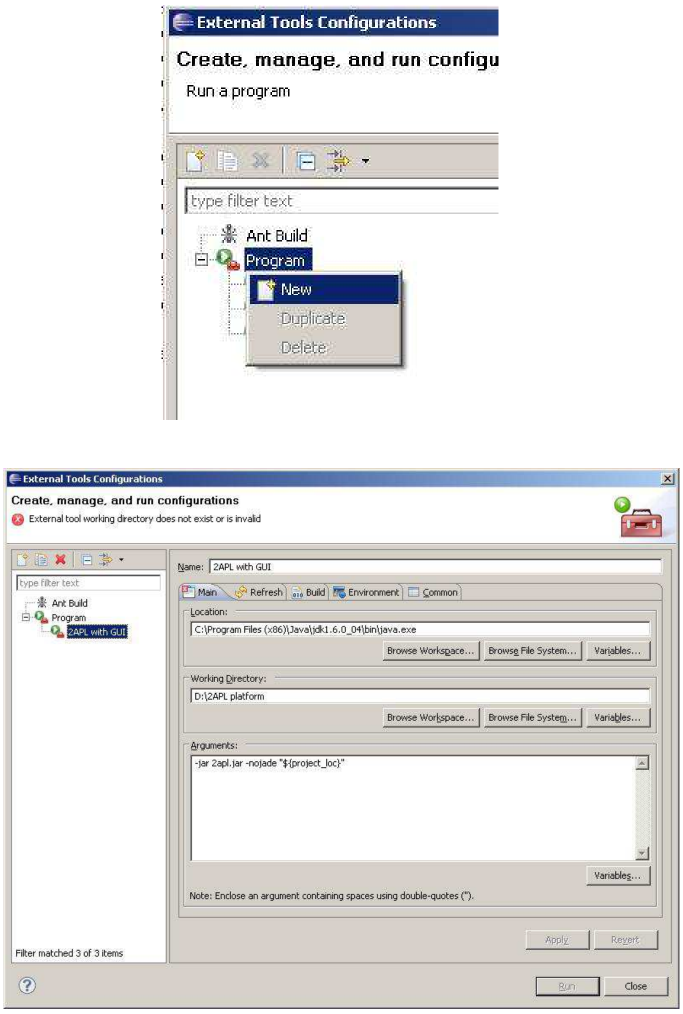
   (a) Name: a name for the run configuration (e.g. `2APL with GUI`)

   (b) Location: the full path to your java.exe
   (e.g. `C:\Program Files\Java\jdk1.6.0_04\bin\java.exe`).

   (c) Working Directory: the path to your 2APL directory where `2apl.jar` is located
   (e.g. `D:\apapl`).

   (d) Arguments: `-jar 2apl.jar -nojade "${project_loc}"`

   These steps are illustrated in Figure 1.12. Note that the required paths are correctly entered. It is also important to note that the behavior of the run configuration can be modified by changing the Arguments. Add the option `-nogui` after `-nojade` to disable the 2APL GUI. Or leave the `-nojade` option out of the Arguments to get a choice to enable the jade platform.



External Tools Configurations

Create, manage, and run configurations

External tool working directory does not exist or is invalid

Name: 2APL with GUI

Location: C:\Program Files (x86)\Java\jdk1.6.0_04\bin\java.exe

Working Directory: D:\2APL platform

Arguments:

```
-jar 2apl.jar -nojade "${project_loc}"
```

Note: Enclose an argument containing spaces using double-quotes (").

Filter matched 3 of 3 items

Figure 1.12: Creating a direct 2APL Run item in Eclipse.

4. Select the multi-agent system file with the `.mas` extension to run. Choose the run configuration you just created as illustrated in Figure 1.13. The 2APL platform will start running the selected multi-agent program.

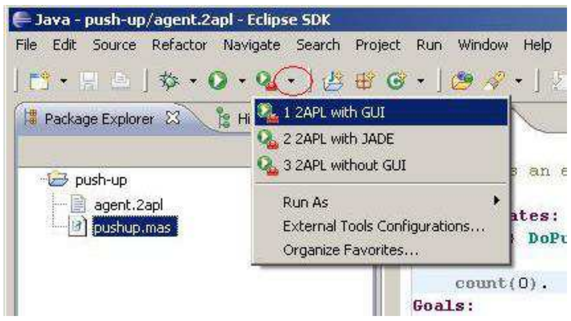

Java - push-up/agent.2apl - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
push-up
agent.2apl
pushup.mas
1 2APL with GUI
2 2APL with JADE
3 2APL without GUI
Run As
External Tools Configurations...
Organize Favorites...

Figure 1.13: Running 2APL from Eclipse.

2 The 2APL Language

2.1 Multi-agent system specification

2APL is a multi-agent programming language that provides programming constructs to specify both multi-agent systems as well as individual agents. The configuration of a multi-agent system is specified in XML format using the tag `apaplmas`. The specification is in included in a file with the `.mas` extension. Within this tag, agents and the environments in which the agents operate are specified. The environment tags are optional since multi-agent systems may consist of only individual agents without any external environment. An example of a multi-agent system configuration is illustrated in Figure 2.1.

```
<apaplmas>

    <environment name="blockworld" file="blockworld.jar">
        <parameter key="gridWidth" value="18" />
        <parameter key="gridHeight" value="18" />
        <parameter key="entities" value="2" />
    </environment>

    <agent name="harry" file="harry.2apl" />
    <agent name="sally" file="sally.2apl" />

</apaplmas>
```

Figure 2.1: The specification of the `harry and sally` multi-agent system system.

The individual agents are specified using the tag `agent` with two mandatory attributes: `name` and `file`. The name will be used to identify the agent in the multi-agent system and the file (with `.2apl` extension) is used to define the initial state of the agent (see below for the syntax of `.2apl` files). In addition, one can use the tag `beliefs` within the tag `agent` in order to extend the agent's beliefs with additional facts. For example, the following is the specification of the agent `harry` from the above multi-agent configuration, extended with some additional beliefs stored in the file `friends.pl`. We assume this file contains the identity of agents who are friends of `harry`.

```
<agent name="harry" file="harry.2apl" />
    <beliefs file="friends.pl" shadow="true"/>
</agent>
```

This option is specially useful for creating various, but different, agents from one and the same `.2apl` file. These agents differ from each other with respect to the additional facts. The `beliefs` tag has an mandatory attribute `file` referring to a Prolog file with a `.pl` extension. The tag `beliefs` can use an optional `shadow` attribute with values `true` or `false`. The first value causes the additional beliefs to be visible in the 2APL user interface, while the second value hides the additional beliefs from the user interface. The tag `beliefs` is not used in the `harry and sally` example.

The environments are specified using the tag `environment` which has two mandatory attributes: `name` and `file`. The first attribute defines the name of the environment (used by agents to perform their actions) and the second attribute defines a jar-file that implements the environment (see section 3 for how an implemented environment can be used). The environments can be initialized with specific parameters, depending on their implementations, using the tag `parameters` each with two mandatory attributes: `key` and `value`. In the `harry and sally` example, the parameters determine the size of the `blockworld` environment and the number of agents expected to appear in that environment.

The 2APL programming language for individual agents supports the integration of declarative concepts such as belief and goals with imperative style programming such as events and plans. This section presents the complete syntax of 2APL (the file with `.2apl` extension), which is specified using the EBNF notation. In this specification, as illustrated below, we use ⟨*atom*⟩ to denote a Prolog like atomic formula starting with lowercase letter, ⟨*Atom*⟩ to denote a Prolog like atomic formula starting with a capital letter, ⟨*ident*⟩ to denote a string and ⟨*Var*⟩ to denote a string starting with a capital letter. We use ⟨*ground_atom*⟩ to denote a grounded atomic formula. Note that the question mark '?' used in ⟨*testbelief*⟩ is a syntactic entity of the 2APL language, while other question marks are used as elements of the EBNF notation indicating the optional choice.

```
⟨APAPL⟩               =  {"Include:" ⟨includes⟩
                      |  "BeliefUpdates:" ⟨beliefupdates⟩
                      |  "Beliefs:" ⟨beliefs⟩
                      |  "Goals:" ⟨goals⟩
                      |  "Plans:" ⟨plans⟩
                      |  "PG-rules:" ⟨pgrules⟩
                      |  "PC-rules:" ⟨pcrules⟩
                      |  "PR-rules:" ⟨prrules⟩}*;
⟨includes⟩            =  ⟨include⟩+;
⟨include⟩             =  ⟨ident⟩ ".2apl";
⟨beliefupdates⟩       =  ⟨beliefupdate⟩+;
⟨beliefupdate⟩        =  "{" [⟨belquery⟩] "}" ⟨beliefupdatename⟩ "{" ⟨literals⟩ "}";
⟨beliefupdatename⟩    =  ⟨upperatom⟩;
⟨beliefs⟩             =  ⟨belief⟩+;
⟨belief⟩              =  ⟨ground_atom⟩"."
                      |  ⟨atom⟩ ":-" ⟨literals⟩ ".";
⟨goals⟩               =  ⟨goal⟩ {"," ⟨goal⟩};
⟨goal⟩                =  ⟨ground_atom⟩ {"and" ⟨ground_atom⟩};
⟨baction⟩             =  "skip"
                      |  ⟨beliefupdatename⟩
                      |  ⟨sendaction⟩
                      |  ⟨externalaction⟩
```

```
                        |   ⟨abstractaction⟩
                        |   ⟨test⟩
                        |   ⟨adoptgoal⟩
                        |   ⟨dropgoal⟩;
⟨plans⟩                 =   ⟨plan⟩ {"," ⟨plan⟩};
⟨plan⟩                  =   ⟨baction⟩
                        |   ⟨sequenceplan⟩
                        |   ⟨ifplan⟩
                        |   ⟨whileplan⟩
                        |   ⟨atomicplan⟩
                        |   ⟨scopeplan⟩;
⟨sendaction⟩            =   "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")"
                        |   "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")";
⟨externalaction⟩        =   "@" ⟨ident⟩ "(" ⟨atom⟩ "," ⟨var⟩ ")";
⟨abstractaction⟩        =   ⟨atom⟩;
⟨test⟩                  =   "B(" ⟨belquery⟩ ")"
                        |   "G(" ⟨goalquery⟩ ")"
                        |   ⟨test⟩ "&" ⟨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⟩+;
⟨pgrule⟩                =   ⟨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⟩
                        |   ⟨infixatom⟩
                        |   "not" ⟨atom⟩
                        |   "not" ⟨infixatom⟩;
⟨belquery⟩              =   "true"
                        |   ⟨belquery⟩ "and" ⟨belquery⟩
                        |   ⟨belquery⟩ "or" ⟨belquery⟩
                        |   "(" ⟨belquery⟩ ")"
                        |   ⟨literal⟩;
⟨goalquery⟩             =   "true"
                        |   ⟨goalquery⟩ "and" ⟨goalquery⟩
```

```
                    |   ⟨goalquery⟩ "or" ⟨goalquery⟩
                    |   "(" ⟨goalquery⟩ ")"
                    |   ⟨atom⟩;
⟨iv⟩                =   ⟨ident⟩ | ⟨var⟩;
⟨groundatom⟩        =   ⟨ident⟩ "(" ⟨groundpars⟩ ")";
⟨groundpars⟩        =   ⟨groundpar⟩ {"," ⟨groundpar⟩};
⟨groundpar⟩         =   ⟨ident⟩ | ⟨num⟩ | "_" | ⟨atom⟩
                    |   "[" [⟨groundpars⟩] "]"
                    |   "[" ⟨groundpars⟩ "|" ⟨var⟩ "]";
⟨upperatom⟩         =   ⟨var⟩ "(" [⟨pars⟩] ")";
⟨atom⟩              =   ⟨ident⟩ "(" [⟨pars⟩] ")";
⟨infixatom⟩         =   ⟨par⟩ ("=" | ">" | "<" | "<=" | ">=" | "=>" | "=<") ⟨par⟩;
⟨pars⟩              =   ⟨par⟩ {"," ⟨par⟩};
⟨par⟩               =   ⟨var⟩ | ⟨num⟩ | "_" | ⟨atom⟩
                    |   ⟨par⟩("+" | "-" | "*" | "/") ⟨par⟩
                    |   "[" [⟨pars⟩] "]"
                    |   "[" (⟨artexps⟩ | ⟨pars⟩) "|" ⟨var⟩ "]";
⟨artexps⟩           =   ⟨artexp⟩ {"," ⟨artexp⟩};
⟨artexp⟩            =   ⟨var⟩ | ⟨num⟩
                    |   ⟨artexp⟩("+" | "-" | "*" | "/") ⟨artexp⟩
                    |   "(" ⟨artexp⟩ ")";
⟨var⟩               =   "A".."Z" {"a".."z" | "A".."Z" | "0".."9" | "_"};
⟨ident⟩             =   "a".."z" {"a".."z" | "A".."Z" | "0".."9" | "_"};
⟨num⟩               =   ("0".."9")+;
```

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 section, 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.

2.2 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 You can use arbitrary Prolog programs to represent an agent's belief base, but this should be done with care as dynamics constructs of Prolog such as assert and retract may interact with belief update actions, resulting in undesirable behavior. All facts are assumed to be grounded. The code fragment below illustrates the implementation of the initial belief base of `harry` which represents his information about its `blockworld` environment. In particular, `harry` believes that it starts working at location $(0, 1)$, and that the `blockworld` environment is clean if there are no bombs anymore and the agent is not carrying a bomb.

```
Beliefs:
  start(0,1).
  clean( blockworld ) :- not bomb(X,Y) , not carry(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 code fragment below is the implementation of the initial goal base of `harry` which indicates that he wants to achieve a desirable situation in which the `blockworld` is clean.

```
Goals:
  clean( blockworld )
```

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 specification of the belief update actions starts with the keyword '`BeliefUpdates:`' followed by the specifications of a set of belief update actions ⟨*beliefupdatespecification*⟩.

The code fragment below 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(X,Y)`; 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.

```
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 ) }
```

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 [2] for more information on JADE standards). The following is an example of a communication action by which `sally` informs `harry` about a bomb location.

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

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(Action,ReturnVal)`. The parameter `Env` is the name of the agent's environment and `Action` is the action the agent wants to perform 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. An example of the implementation of an external action is `@blockworld( enter( X, Y, blue ), _ )`. This action causes an agent to enter the blockworld environment at position `(X,Y)`. The appearance color of the agent will be blue. The execution of this action causes the agent to appear in blue at position `(X,Y)` in the blockworld environment. 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 $\langle abstractaction \rangle$ 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 $\langle testbelief \rangle$ is an expression of the predicate argument form followed by a question mark. 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). For example, let the belief based of `harry` contains the fact `start(0,1)`. the belief test action `B(start(X,Y))` succeeds resulting in the substitution `[X/0 , Y/1]`. If there are more than one substitutions possible, then the first substitution will be taken.

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 $\langle testgoal \rangle$ is an expression of the predicate argument form followed by an exclamation mark. 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.

The belief and goal test actions can be combined forming complex test actions. For example, `B(start(X,Y)) & G(pos(V,W))` tests the agent's start and desired positions.

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 $\langle adoptgoal \rangle$ 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 $\langle dropgoal \rangle$ 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 subgoals of ϕ, and all goals that have ϕ as a subgoal. For example, let an agent have the goal `p(a) and q(b)`. Then, the action `dropgoal(p(a))` or `dropgoal(p(a) and q(b) and r(c))` will not remove the goal `p(a) and q(b)` from the goal base, but the action `dropgoal(p(a) and q(b))` does remove the goal. Also, the actions `dropsubgoal(p(a) and q(b) and r(c)))` `dropsupergoal(p(a)` will remove the goal `p(a) and q(b)` from the goal base.

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, e.g., `goto( X, Y ); @blockworld( pickup( ), _ )` is a sequence plan consisting of an abstract action followed by an external action. A sequence plan $\pi_1;\pi_2$ indicates that the first plan π_1 should be performed before the second plan π_2.

The conditional choice operator generates the plan ⟨*ifplan*⟩, which is an expression of the form `if` ϕ `then` π_1 `else` π_2. The following is an example of such a plan:

```
if B(A > X) then
{   @blockworld( west(), L );
    goto( X, Y )
}
else if B(A < X) then
{   @blockworld( east(), L );
    goto( X, Y )
}
```

The condition of such an expression (i.e., ϕ) is evaluated with respect to an agent's belief 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 agent believes ϕ, 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` ϕ `do` π. The condition ϕ should also be evaluated with respect to an agent's belief 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 believes ϕ.

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 initial 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 (plans are separated by comma). The following code fragment illustrates the implementation of the initial plan base of `harry`, which first tests where he believes to enter, then associate a robot body to get an appearance in the environment, and finally enters the `blockworld` at the believed position. The exact meaning of the external actions that could be performed in the `blockworld` is explained in chapter 3. 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:
    B(start(X, Y));
    @blockworld(associateWith(robot0),_,0);
    @blockworld(enter(X, Y, blue),_,0)
```

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 can 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. The planning goal rules are preceded by the keyword `PG-rules:` and have the following form:

⟨*query*⟩? "< −" ⟨*query*⟩ "|" ⟨*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. The code fragment below 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 the execution of which replaces a plan for going to position `(X,Y)`. After performing this plan, `harry` performs a pickup action in the blockworld, modifying his beliefs such that he now believes he carries a bomb, modifying his beliefs that there was a bomb at `(X,Y)`, going to position `(0,0)`, where the bomb should be dropped in a dustbin, perform the drop action in the blockworld, and finally modifying his beliefs that he does not carry a bomb anymore.

```
PG-rules:

clean( blockworld ) <- bomb( X, Y ) |
    {
        goto( X, Y );
        @blockworld( pickup( ), _ );
        PickUp( );
        RemoveBomb( X, Y );
        goto( 0, 0 );
        @blockworld( drop( ), _ );
        Drop( )
    }
```

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 query ⟨*query*⟩, which represents either a message, an event, or an abstract 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. The procedural rules are preceded by the keyword `PC-rules:` and have the following form:

⟨*atom*⟩ "$<-$" ⟨*query*⟩ "|" ⟨*plan*⟩

The code fragment below shows an example of procedural call rules that is used for implementing `harry`. This rule indicates that if `harry` receives a message from `sally` informing him that there is a bomb at position `(X,Y)`, then his beliefs will be updates with this new fact and goal to clean the `blockworld` is adopted, if he does not believe that there is already a bomb at a position `(A,B)`. Otherwise, he updates his beliefs with the the received information without adopting the goal. This is because `harry` is assumed to this goal as long as he believes there is a bomb somewhere.

```
PC-rules:

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

Another example of a procedural rule is illustrated below. This rule is used by `harry` by including the file `person.2apl` (see section 2.7 about include files). This rule indicates that the execution of the abstract action `goto(X,Y)` causes this action to be replaced with the plan from the body of the rule. This plan implements a movement toward position `(X,Y)` by moving one square at a time. Note the use of recursion in this PC-rule.

```
PC-rules:
```

```
goto( X, Y ) <- true |
    {
        @blockworld( sensePosition(), POS );
        B(POS = [actionresult([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 )
            }
    }
```

Plan Repair Rules (PR rules)

The execution of an agent's plan fails if the execution of the first action of the plan fails. To repair such a plan, 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 a variable that can be instantiated with a plan. A plan repair rule starts with an action (the execution of which is failed) followed by a plan variable. A plan repair rule indicates that if the execution of the first action of the agent's plan fails and the agent has a certain belief, then the failed plan should be replaced by another plan. The plan repair rules are preceded by the keyword `PR-rules:` and have the following form:

⟨*planvar*⟩ "< −" ⟨*query*⟩ "|" ⟨*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
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 variable that occur in the abstract plan in the body of the rule. Note that one of these variables is a plan variable which will be instantiated with the the plan without its first action. The resulted substitutions will be applied to the second abstract plan (the body of the rule) resulting a new (repaired) plan. The code fragment below 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(), _ );` fails. Such an action fails in case there is no bomb to be picked up (e.g., when it is removed by another agent). The rule states that the plan should be replaced by another plan consisting of an external action to sense its current position after which it removes the bomb from its current position. 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(), _ ); REST <- true |
    {
        @blockworld( sensePosition(), POS );
        B(POS = [actionresult([X,Y])]);
        RemoveBomb( X, Y )
    }
```

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. The 2APL deliberation cycle can be changed by editing and recompiling the codes of the 2APL interpreter. However, we plan to extend the platform such that the deliberation cycle can be modified through parameters.

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 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.

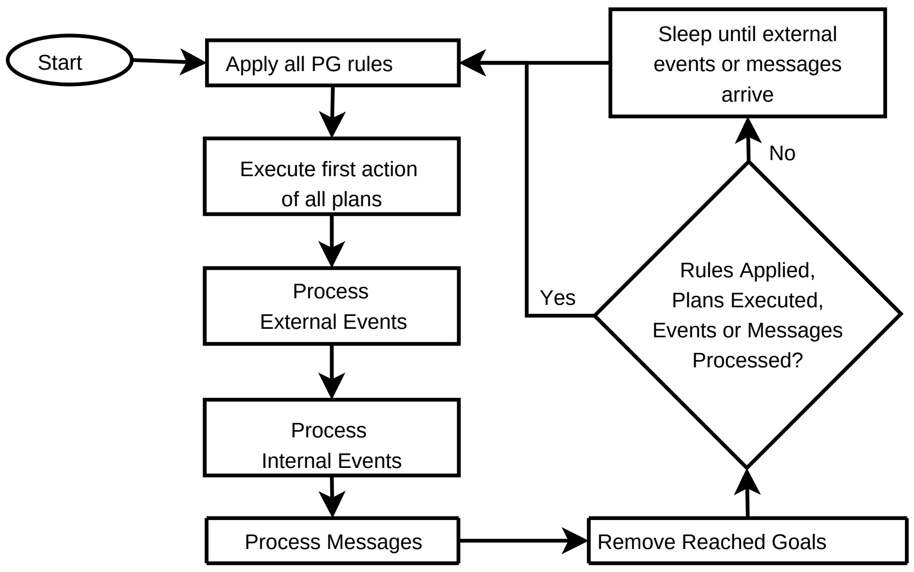

Start
Apply all PG rules
Execute first action of all plans
Process External Events
Process Internal Events
Process Messages
Remove Reached Goals
Rules Applied, Plans Executed, Events or Messages Processed?
Yes
No
Sleep until external events or messages arrive

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

3 2APL Environments

2APL agents can interact with external environments by performing actions in it. The environments for 2APL agents are implemented in Java. You can either use the `blockworld` environment that is provided by the 2APL platform (see section 2.3), or implement your own environment. For implementing your own environment, we refer to the following document [1]. In the following, we explain how to use the blockworld environment. In general, there are two kinds of actions that an agent can perform with respect to the environments: the management actions and the domain actions.

The management actions are generic to all kind of environments and are aimed at managing the agents-entities-relations, i.e., relating one 2APL agent with entities (for example their appearance) in the environments. The syntax of these actions are similar to external actions. The following are the management actions that an 2APL agent can perform with respect to an environment. For the blockworld environment, the term `env` should be replaced by `blockworld`.

action name:	getFreeEntities
parameter(s):	none
return value(s):	`R`
example:	`@env( getFreeEntities() , R )`

description:
It returns a list R of all free unassociated entities in the environment env.

action name:	getAllEntities
parameter(s):	none
return value(s):	`R`
example:	`@env( getAllEntities() , R )`

description:
It returns a list R of all entities in the environment env.

action name:	associateWith
parameter(s):	`E`
return value(s):	`R`
example:	`@env( associateWith(E) , R )`

description:
It associates the action performing agent with the entity or a list of entities E. It returns a list R with the only element 'success'.

action name:	disassociateFrom
parameter(s):	`E`
return value(s):	`R`
example:	`@env( disassociateFrom(E) , R )`

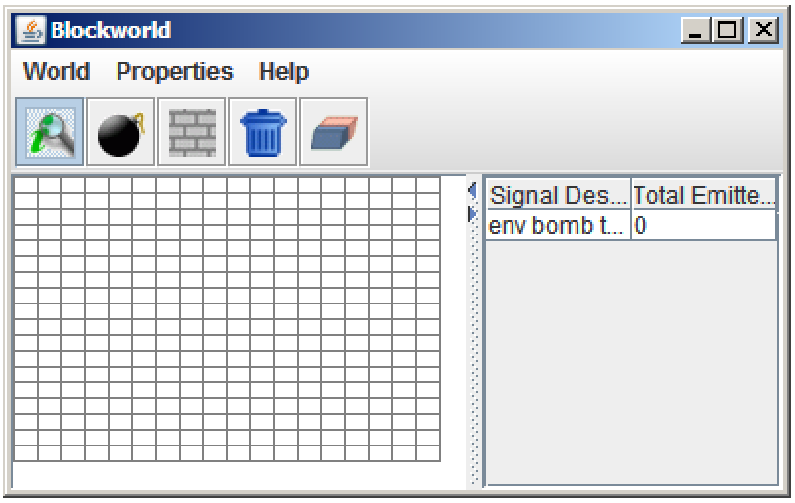
description:
It disassociates the action performing agent with the entity or a list of entities E. It returns a list R with the only element 'success'.

action name:	getAllPercepts
parameter(s):	none
return value(s):	`R`
example:	`@env( getAllPercepts() , R )`

description:
It returns a list R of all current percepts.

3.1 Using 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 dustbins. Agents can pickup and drop bombs. When a bomb is dropped in a dustbin, the bomb is destroyed. Figure 3.1 shows an example instance of the `blockworld` with one agent located in it. You can add and delete bombs, walls (stones), and dustbins. 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 by clicking the 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`.



Signal Des...	Total Emitte...
env bomb t...	0

Figure 3.1: An example instance of the blockworld with one agent located in it.

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
example:	`@blockworld( enter(5,5,red), R)`
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.

action name:	sensePosition
parameter(s):	none
return value(s):	`[X,Y]`
example:	`@blockworld( sensePosition(), R)`
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
example:	`@blockworld( north(), R)`
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
example:	`@blockworld( south(), R)`
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
example:	`@blockworld( east(), R)`
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
example:	`@blockworld( west(), R)`
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(), R)`
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(), R)`
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(), R)`
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(), R)`
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(), R)`
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(), R)`
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(), R)`
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(), R)`
description:	

Gives a list of all dustbins in the sense range. The first element of the sublist (each dustbin 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(), R)`
description:	

Gives a list of all dustbins, also the ones outside the sense range. The first element of the sublist (each dustbin 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(),R)`
description:	

Returns the sense-range of the agent.

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

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

Bibliography

[1] T. M. Behrens, J. Dix, and K. V. Hindriks. The environment interface standard for agent-oriented programming platform integration guide and interface implementation guide. Technical Report IfI-09-10, Clausthal University of Technology, Dec. 2009.

[2] 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.

[3] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[4] M. Dastani. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, chapter Modular Rule-Based Programming in 2APL. Number 1 in Information Science Reference. IGI, Hershey, PA, USA, 2009.

[5] M. Dastani and B. Steunebrink. Operational semantics for BDI modules in multi-agent programming. In *Proceedings of the tenth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-X)*, 2009.