Goal User Manual

Koen V. Hindriks, Wouter Pasman, and Vincent Koeman

March 16, 2014

Contents

Chapter 1

Introduction

This user manual describes and explains how to:

- Install the GOAL platform,
- Start and use the GOAL plug-in for the Eclipse IDE,
- Run a MAS in stand-alone mode without the IDE; use this mode, e.g., for running simulations and running the test framework, and
- Create your own environments that you can connect agents to using the Environment Interface Standard (EIS).

This document does not introduce the agent programming language GOAL itself. For this we refer the reader to the GOAL Programming Guide [3] and GOAL's website [4].

Chapter 2

Installing the GOAL Platform

This chapter explains how to install the GOAL platform. We first describe the system requirements of the platform. Please verify that your system meets the minimal system requirements.

2.1 System Requirements

The GOAL platform runs on Windows, Macintosh OSX and the Linux operating system. For information on the exact versions of these OSs, please check GOAL's website at `http://ii.tudelft.nl/trac/goal`.

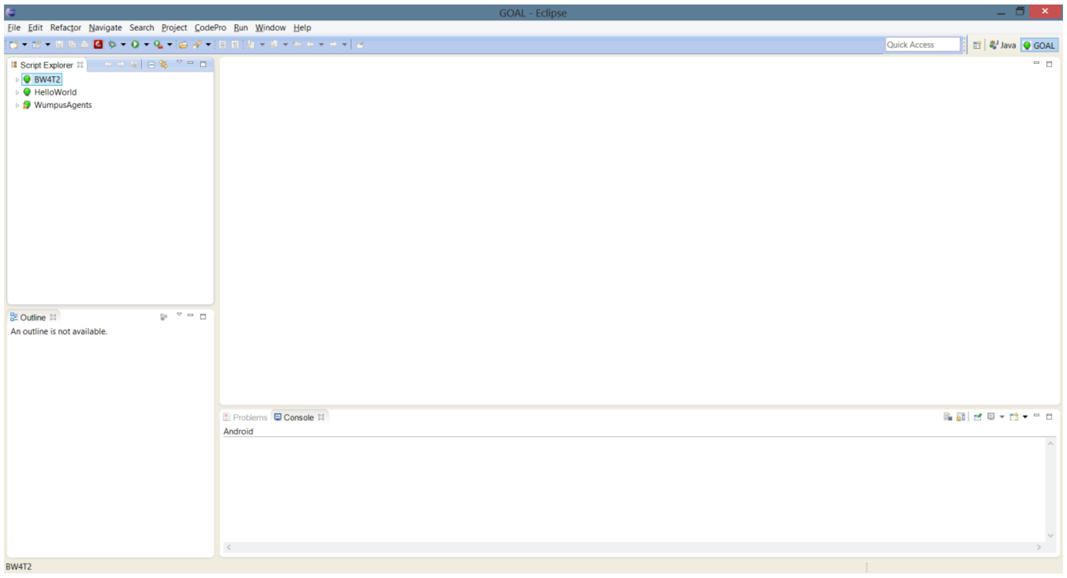
The GOAL platform requires Java (SUN or OpenJDK) version 1.6 or higher. We recommend using Sun Java 1.7.

2.2 Installation

GOAL is available as a plug-in for **Eclipse Kepler** (see `https://www.eclipse.org/downloads/`). It can be installed as follows:

- In Eclipse, select: Help → Install → New Software → Add.
- Add a new repository with any name you like using the following URL: `http://ii.tudelft.nl/trac/goal/export/head/GOALplugin/ECLIPSE/dist/`
- Select the new 'GOAL Agent Programming' field (make sure you have 'Group items by category' enabled) and click Next to follow the final steps.
- Ignore (accept) any warning about unsigned content, and restart Eclipse when the installation is done (you will be prompted).
- To start developing GOAL projects, switch Eclipse to the GOAL perspective through: Windows → Open Perspective → Other → GOAL. Your window should now look similar to 2.1.
- We recommend enabling automatic updates in Eclipse through: Window → Preferences → Install / Update → Automatic Updates.

If you are a Linux user, please make sure to run Eclipse as administrator at all times. On Windows, if User-Account-Control is enabled, Eclipse itself should either be installed in a user-owned folder or run as administrator.



An outline is not available.

Figure 2.1: The GOAL perspective in Eclipse

2.3 Uninstalling

For uninstalling the GOAL plug-in, either remove your whole Eclipse installation, or follow these steps to uninstall just the plug-in:

- In Eclipse, select Help → About Eclipse → Installation Details.
- Select 'GOAL Eclipse' in the list of Installed Software, and click 'Uninstall'.
- Follow the remaining steps and restart Eclipse to complete the process.

2.4 Upgrading

Upgrading the GOAL plug-in to a newer version, outside of the aforementioned automatic update functionality, is possible in Eclipse by selecting: Help → Check for Updates.

2.5 Customizing

Various GOAL platform features can be customized, e.g., debugging features.

- Select Window → Preferences, and unfold the GOAL menu item.
- Select one of the three categories (Logging, Runtime, or Templates) to edit the related settings.

The default settings can always be restored using the 'Restore Default' button.

Chapter 3

Creating and Managing Multi-Agent Projects

This chapter explains how to create new, and manage existing multi-agent projects.

3.1 Creating a New Multi-Agent Project

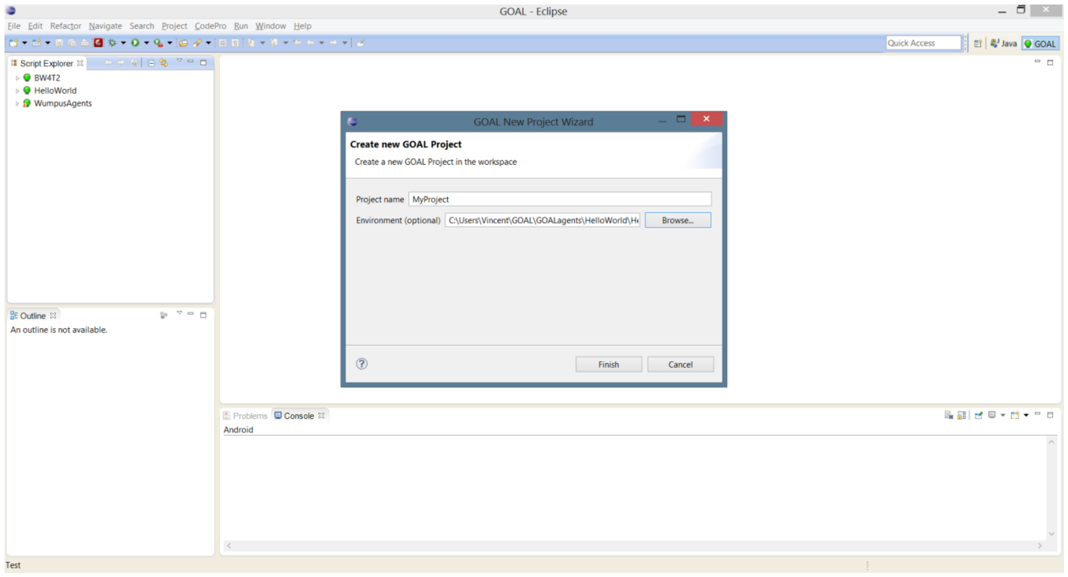
How to create a new multi-agent project:

- In the GOAL perspective in Eclipse, select File → New → GOAL Project
- Enter a name, and optionally browse for an environment file to include in the project.
- Press Finish. A MAS file will have been automatically created within the new project.

It is also possible to start-off with one of the many example project included in this plug-in by selecting 'GOAL Example Project' instead of 'GOAL Project' in the first step above. In the following screen, select one of the example projects, and press Finish to start working on it.

MAS files are files with extension `.mas2g`. When a new MAS file is created, a **MAS template** is automatically loaded. This template inserts the the sections that are required in a MAS file; you should adapt the content of these sections of the MAS file to your needs. A MAS file must have an **agentfiles** and **launchpolicy** section; an **environment** section is optional. After opening a new MAS file, you should see something similar to:

Warnings are displayed because the sections are empty. As can be seen, warnings and errors are shown in at the line at which they occur, underlining the specific text causing the issue. In addition, the *Problems* tab can be used to list all errors/warnings. Any errors need to be fixed before we can launch a MAS. Note that multiple MAS files can be added to a single project, e.g. for different run configurations. Save a file either by using the Save icon in the menu, or by pressing CTRL-S (Windows and Linux) or APPLE-S (OSX).



Create new GOAL Project

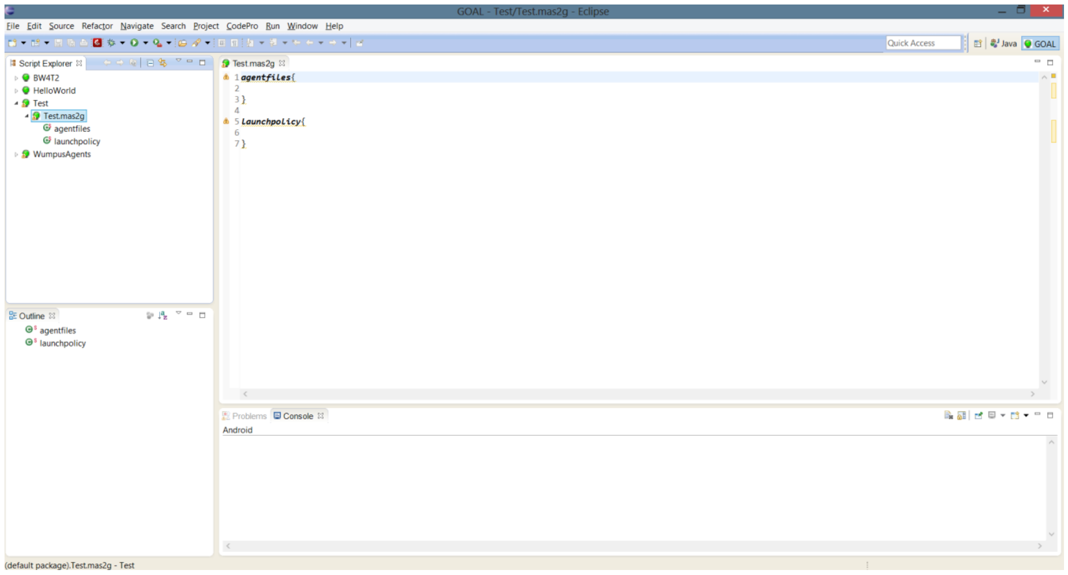
Create a new GOAL Project in the workspace

Project name: MyProject

Environment (optional): C:\Users\Vincent\GOAL\GOALagents\HelloWorld\He

An outline is not available.

Figure 3.1: The New Project Dialogue Window.



```
agentfiles{

}

launchpolicy{

}
```

Figure 3.2: A new MAS File.

3.1.1 Adding an Agent File to a MAS Project

How to add a new agent file to a MAS project:

- Right-click any file within the project or the project itself (in the Script Explorer), and select New → GOAL Agent File.
  Alternatively, if a file or project is already selected, File → New → GOAL Agent File can also be used.
- Give a (unique) name to the agent file, and press Finish.

Do not forget to add the agent file you have just created to the **agentfiles** section of a MAS file if required.

A new agent file is automatically loaded with a corresponding template. Again, warnings are displayed for a new file because the main module's program section is empty.

3.1.2 Adding an Environment to a MAS Project

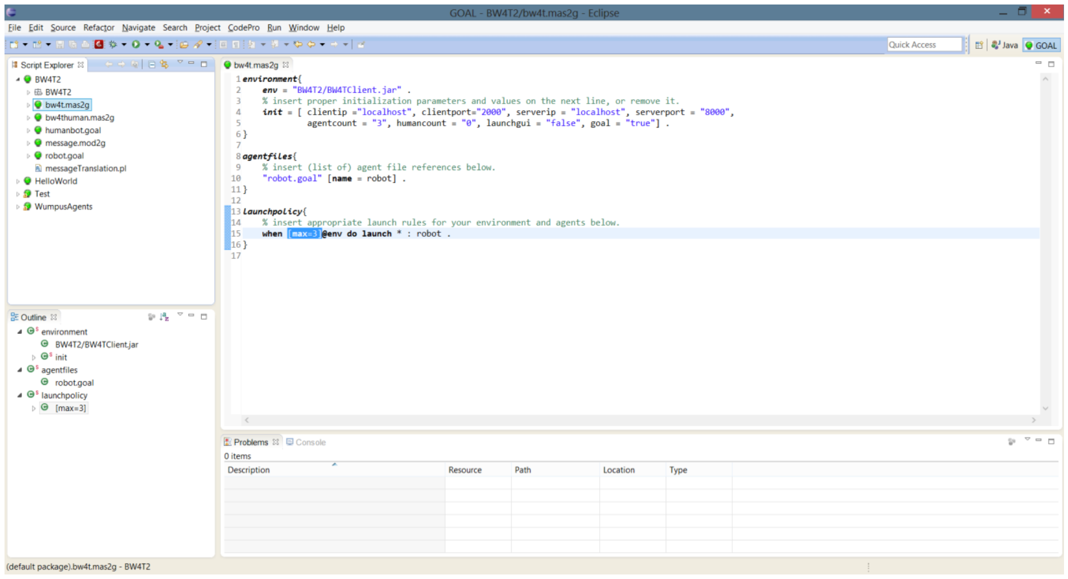
An environment is added to a MAS project by first adding it to the project (which can be done for a new project or an existing one), and then editing the **environment** section in a MAS file. Note that the **environment** section is *optional* and that a multi-agent system can be run without an environment.

- Right-click any file within the project or the project itself (in the Script Explorer), and select Import.
  Alternatively, if a file or project is already selected, File → Import can also be used.
- Select General → File System, and press Next.
- Browse for the directory the environment file is in. In the right pane, the desired file(s) can now be selected.
- After having selected the right file(s), press Finish.
- Add `env = "filename"` to the **environment** section in the MAS file.
- If the environment supports initialization parameters (check the documentation that comes with the environment), add `"init = [<key=value>, ...]".` to the **environment** section in the MAS file; here, where `key` is the name of the parameter and `value` the value you choose to initialize the environment with.

Please note that you can also manually place the JAR file within the actual project location on your hard-drive; Eclipse will automatically update your project (or right-click the project and select Refresh to force a reload). References to environments in a MAS file are resolved by checking whether the environment can be found relative to the folder where the MAS file is located, or else, an absolute path should be specified to the environment interface file.

The GOAL platform supports the Environment Interface Standard (EIS) to connect to environments [2, 1]. See Chapter 7 on how to develop an environment for creating your own environments. Any environment that is compliant with the most up to date version of EIS that is supported can be used in combination with GOAL.

For connecting your agents to an environment that has been launched by another instance of the GOAL platform, possibly on another machine, check out Section 4.6.



```
environment{
    env = "BW4T2/BW4TClient.jar" .
    % insert proper initialization parameters and values on the next line, or remove it.
    init = [ clientip ="localhost", clientport="2000", serverip = "localhost", serverport = "8000",
             agentcount = "3", humancount = "0", launchgui = "false", goal = "true"] .
}

agentfiles{
    % insert (list of) agent file references below.
    "robot.goal" [name = robot] .
}

launchpolicy{
    % insert appropriate launch rules for your environment and agents below.
    when [max=3]@env do launch * : robot .
}
```

Figure 3.3: Launch Policy of the BW4T example project.

3.1.3 Adding a Launch Policy

In order to create agents when a multi-agent system is launched, a launch policy for creating the agents is needed. A launch policy is a set of rules which specify when to launch an agent.

How to add a launch policy:

- Either, add a *launch instruction* of the form `launch <agentname> : <agentfile>`.
- Or, add a *launch rule* `when entity@env do launch <agentname> : <agentfile>`.

A launch instruction creates and adds a new agent to the MAS when the MAS is launched. A launch rule only creates an agent when a controllable entity in an environment becomes available (i.e., when the GOAL platform is notified that an entity has been born).

Agent name and agent file name can but do not have to be different.

Tip By using an asterix `*` instead of an agent name in a launch rule, the name of the agent that will be created will be determined by the name of the environment entity that the agent is connected to. This makes it easier to relate agent and entity in an environment. For example, an agent may be called `bot` if the entity in the environment is called `bot`. When leaving the agent name away, the name will be set to that of the agent file automatically.

Finally, more than one agent can be connected to a single entity. The GOAL Programming Guide [3] provides a comprehensive explanation of launch policies.

3.1.4 Importing Knowledge and Module Files

When you start writing larger agent programs it is often useful to add more structure to the MAS project. GOAL provides the `#import` command to import contents from other files into an agent program. An important reason for moving code to separate files and importing those files is to facilitate **reuse**.

How to create and import a knowledge file into an agent file:

- Use the File → New → Prolog File menu to create a new `<knowledge>.pl` file, where `<knowledge>` is a name for the knowledge file.
- Select and cut all code in the **knowledge** section of the agent program.
- Paste the selected code from the **knowledge** section into the `<knowledge>` file, and save.
- Insert `#import "<knowledge>.pl".` in the **knowledge** section of the agent program, and save.

A procedure very similar to that for knowledge files can also be used to move complete modules to separate *module files* with extension `.mod2g`.

How to create and import a module file into an agent file:

- Use the File → New → GOAL Module File menu to create a new `<modulefile>.mod2g` file, where `<modulefile>` is a name for the module file.
- Select and cut all code for a module in the agent program.
- Paste the selected module code into the `<modulefile>`, and save.
- Insert `#import "<modulefile>.mod2g".` in the agent program at the top-level, and save.[a]

[a]Top-level simply means that the code is not included in any other structure such as a module or other program section.

These are the two types of files that can be imported: Files that contain modules and files that contain knowledge. Even if an agent is not a very large program, it sometimes is useful to put some of the code in separate module or knowledge files to facilitate *reuse*. The code in a **knowledge** section of an agent, for example, can be (re)used in another agent program. Reusing the same code by creating a separate file that is used in multiple agent files prevents possible problems introduced by modifying the same code that appears at different places in only one location. By using a dedicated file the code is needs to be maintained and modified in only one place.

3.2 Managing an Existing Multi-Agent Project

Loading an existing multi-agent system project can be done through the following steps:

How to import an existing project into Eclipse:

- Choose File → Import, select Existing GOAL Project (in the GOAL Agent Programming category), and press Next.
- Browse for the `.mas2g` file that indicates the system that is to be imported.
- Optionally, check the Copy into workspace checkbox in order to make a copy of all the files, instead of using them directly in the new project.
- Press Finish. A new project with the name of the MAS file will have been created.

Please not that only files that are in the (sub)directory of the selected `.mas2g` file will be copied (or directly used) in the new project. Any external files will have to be moved into the project manually, by using the file system or the steps as explained in Section 3.1.2.

3.2.1 Renaming, Moving and Deleting a File

The name of a file can be changed by right-clicking on it, and selecting Refactor → Rename. Note that a file can also be moved through the Refactor menu. Deleting a file can simply be done by right-clicking on it and selecting Delete. A **warning** is in place, however: *Changing a file name or location that is part of a MAS project does not modify the contents of the mas file itself.* When changing an agent file name, or moving or deleting it, you need to manually change the **agentfiles** section of the mas file. All of these operations can also be done on the file system itself.

3.2.2 Automatic Completion

Automatic code completion is fully supported.

Using auto-completion:

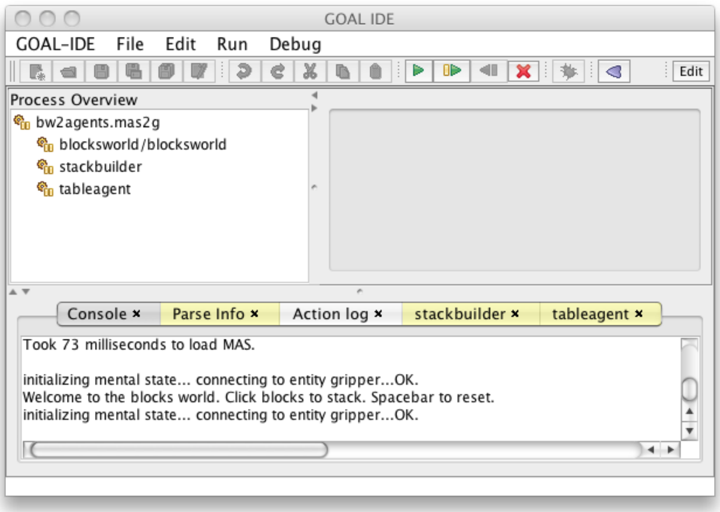
- After typing some initial code, it can be automatically completed by pressing CTRL-Space.
- If there is more than one option for completion, select an option from the pop-up menu (otherwise the word is completed right away).

Chapter 4

Running a Multi-Agent System

This chapter explains how to launch, run, and distribute a MAS project across multiple machines. Running a multi-agent system means launching or connecting to the environment the agents perform their actions in (if any), and launching the agents when appropriate. As explained in Section 3.1.3, which environment is launched depends on the **environment** section of the MAS file that is being run. Which agents are launched depends on the launch policy specified in the same MAS file. Section 4.6 explains how to distribute one or more MAS across multiple machines.

Please not that although currently, the GOAL runtime is launched in a separate window, we are working on fully integrating this in Eclipse.



```
Took 73 milliseconds to load MAS.

initializing mental state... connecting to entity gripper...OK.
Welcome to the blocks world. Click blocks to stack. Spacebar to reset.
initializing mental state... connecting to entity gripper...OK.
```

Figure 4.1: The runtime after launching`bw2agents.mas2g`.

4.1 Launching a Multi-Agent System

Launching a MAS means:

- Launching an environment referenced in the MAS file, if any, and
- Creating agents in line with the launch policy of the MAS.

How to launch a MAS:

- Right-click a MAS file in the Script Explorer.
- Select Debug As → GOAL.

Eclipse will automatically start the GOAL runtime when a MAS is launched. In this new window, the **Process panel** is shown in the main content area as illustrated in Figure 4.1. The Process panel shows the processes that have been created by launching the MAS. In the Blocks World example, an environment process is started labeled *Blocks World*, as well as two agent processes named *stackbuilder* and *tableagent*. Also, the Blocks World GUI shown in Figure 4.2 should appear. All processes that are created are *paused* initially, which is indicated by the icon just left of the process names.[1]

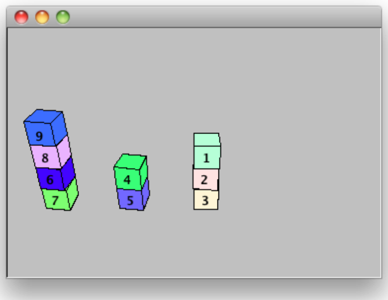


Figure 4.2: The Blocks World GUI after launching `bw2agents.mas2g`.

Before launching a MAS, make sure to check that the MAS project is clean. That is, make sure no warnings or errors have been produced. When you launch a multi-agent system, GOAL will not start the system when one or more program files contains an error. Apart from the need to remove parsing errors, it is also better to clean up program files and make sure that no warnings are present.

Please not that it also possible to run a MAS without a visual interface by selecting 'Run As → GOAL' instead of 'Debug As → GOAL'. The Run functionality will run the MAS through a console (command-line), which is useful for quick testing of the desired functionalities.

4.1.1 Initializing an Environment

In order to actually run a MAS, it may also be necessary to initialize an environment by means of the environment's user interface. In the UNREAL TOURNAMENT environment, for example, you can create new bots that can be controlled by agents. Please consult the environment documentation for details. Depending on the configuration and launch rules of your MAS file, new agents are automatically created when a new bot is added. Refer to the Programming Guide for more information on this [3]. Note that environments often also have entities that cannot be controlled by agents such as people taking the elevator in the Elevator environment or UNREAL bots that are controlled by the UNREAL engine itself.

[1]The status of the environment as indicated by the icon next to the environment process name may not always be known. In that case the icon may be shown to indicate the state is unknown. This icon will be shown with some environments that do not provide information about their current state.

4.2 Running a Multi-Agent System

After an environment has been initialized, a multi-agent system can be run.

How to run a MAS:

- Select the MAS node in the Process panel.
- Press the Run button to run the multi-agent system.

All processes, including the environment and the agents, must be in *running mode* to be able to run the *complete* multi-agent system. The symbol ▶ indicates that a process is in running mode. In running mode, an agent will execute a complete reasoning cycle without pausing.

If the agents in a MAS are paused but the environment is not the environment still may be changing due to processes that are not controlled by the MAS or simply because actions take time (durative actions). Vice versa, if an environment is paused but one or more agents in the MAS are running, typically you will see complaints that the agent was not able to retrieve percepts from the environment.

4.3 Terminating a MAS and/or Its Agents

How to terminate a MAS or an Agent:

- Select the MAS or agent you want to kill in the Process panel.
- Press the Kill button ✖ in the tool bar.
  Alternatively, press the red stop button within Eclipse, or just quit the runtime window. The GOAL kill-button is preferred though!

Terminating a MAS terminates the environment and all agents. Killing a single agent does not kill the MAS. The ✖ symbol indicates that a process has been killed. An agent that has been killed is not performing any actions any more. If a killed agent is put back in running mode, the agent is restarted and initialized using the initial mental state specified in the agent program. An agent will also be *automatically killed* if the entity that it is controlling in an environment has been terminated. Agents that are not associated with an entity are only killed if the user kills it.

4.4 Run Modes of a MAS and Its Agents

A MAS or agent can be either in the running, stepping, paused, or killed mode. Controlling the run mode of a process is done with the Run ▶, Step ⏯, Pause ⏸, and Kill ✖ buttons in the tool bar. Alternatively, you can either use the Run menu or select a process and right-click (ctrl-click on OSX) to set the run mode. You can *control all processes* that are part of the MAS at once by selecting the mas process node and using the relevant buttons. This is particularly useful when starting, stepping or killing all agents.

The following symbols are used to indicate the run mode of a process:

- ⏸ indicates that the process is paused,
- ▶ indicates that the process is running,
- ⏯ indicates that a process is in stepping mode,

- ✖ indicates that the process has been killed.

Note It may happen that an agent tries to perform an action in an environment when the environment has been paused because the agent process and environment run asynchronously. In that case, the environment may throw an exception which will be printed as a warning in the console tab. The agent will proceed as if the action has been succesfully executed.
Finally, note that environments can be in running and paused mode, but not in stepping mode.

4.5 Resetting a MAS and Its Environment

You can use reset to reset agents and environment. If you select a MAS process node and apply reset, only the agents are reset but not the environment. To reset the environment, select the environment and apply reset. Note that reset of the environment keeps the agents connected and alive, so this is less complete than a kill and restart of the entire MAS.

The Backward stepping button in the tool bar is not yet supported.

4.6 Distributed Running of One or More MAS

The agents in a multi-agent system and the environment to which the agents are connected can be distributed over different machines. The middleware infrastructure that is used to run the multi-agent system needs to support distributed processing.

4.6.1 Messaging Infrastructures

The term messaging refers to the communication infrastructure on top of which a multi-agent system runs. Messaging enables agents to exchange messages between each other. GOAL currently provides two alternative messaging platforms: Local messaging and a messaging based on RMI. The local middleware is selected by default, since the performance of the local middleware usually outperforms that of RMI. But to run a truly distributed agent system you need to select RMI Messaging. Use the Run menu to select one of these.

4.6.2 Distributing MAS

It is possible to connect a MAS to an existing environment that is already running in another instance of the GOAL platform. This can happen, for example, when a second MAS is run (using a second instance of the GOAL platform). To do this, you provide a reference to the name of the already running environment, e.g.,

```
env = "elevatorenv".
```

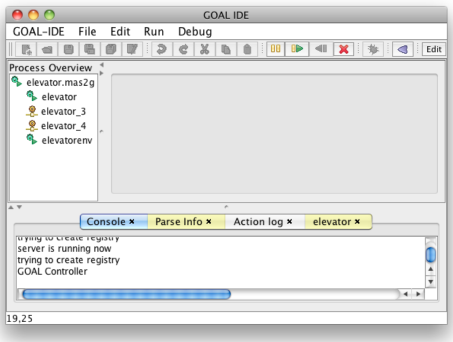
The name is derived from the jar file name, and gets a serial number added to it if you launch multiple of them. If you connect multiple MASs to a single environment, the second MAS will only be able to connect to entities that are still free.

A GOAL multi-agent system can be run in a truly distributed fashion, having various agents running on different JVMs and/or on different machines. In this case there is one *main platform* that is the access point for the other platforms. The main platform has to be launched first, after which the other platforms can join. *Currently only RMI supports distributed running.*

How to launch a distributed system (SimpleIDE only):

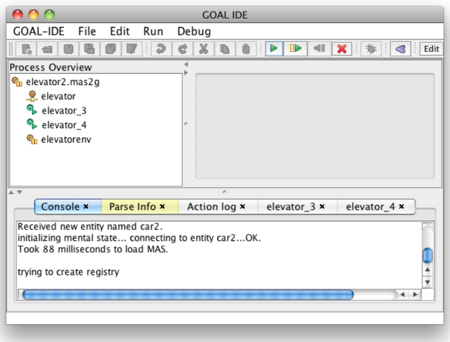
- Start the GOAL IDE on the main platform.
- Uncheck the *Always run the middleware locally* checkbox in the Preferences/PlatformManager menu (this is already off by default).
- Select 'Run RMI (Distributed)' in the Run menu.
- Open the `.mas2g` file that you want to run on the main platform.
- Launch the mas project.
- Use the default `localhost` in the middleware host dialogue window.
- You can start the agents if you like
- Start a new GOAL IDE on a second platform (this may be the same machine).
- Select RMI in the Run menu.
- Open the `.mas2g` file that you want to run on the second platform.
- Launch the second mas project.
- Enter the name of the main platform machine, e.g. `rmi://130.161.180.1` in the middleware host dialogue window.
- Start the agents on all platforms.

You should have two GOAL IDEs running (note: Eclipse not supported), similar to Figure 4.3 and Figure 4.4. Note the ”remote process” icon indicating that a process is running on a remote platform. The IDE only allows introspection and debugging of local agent processes.



```
trying to create registry
server is running now
trying to create registry
GOAL Controller
```

Figure 4.3: IDE for machine 1.



```
Received new entity named car2.
initializing mental state... connecting to entity car2...OK.
Took 88 milliseconds to load MAS.

trying to create registry
```

Figure 4.4: IDE for machine 2

If you take down the second MAS, the entities will be released and can be reused by other, still running MASs. If you take down the MAS that launched an environment, the environment will be taken down, removing all associated entities. All agents associated to these entities will then be killed.

Chapter 5

Debugging a Multi-Agent System

The GOAL runtime provides various options for debugging. While running a mas, GOAL provides a range of options for debugging which include the following:

1. options for viewing and controlling **debug output** produced while running a mas,
2. setting general and specific **breakpoints** to pause an agent on,
3. agent **introspectors** for inspecting the mental state of an agent, and percepts and messages received at runtime,
4. a **message sniffer** for tracing messages that are exchanged between agents (currently, only when running the JADE middleware).

[**TODO:** GOAL facilitates debugging of a multi-agent system in various ways. It provides various forms of feedback in the feedback area at the bottom of the runtime in debug mode, i.e. when a multi-agent system is running. In addition, for each agent, an introspector can be opened to inspect the contents of the mental state of an agent. Finally, it is possible to set breakpoints to pause an agent when it hits such a breakpoint.]

5.1 Introspecting Agents

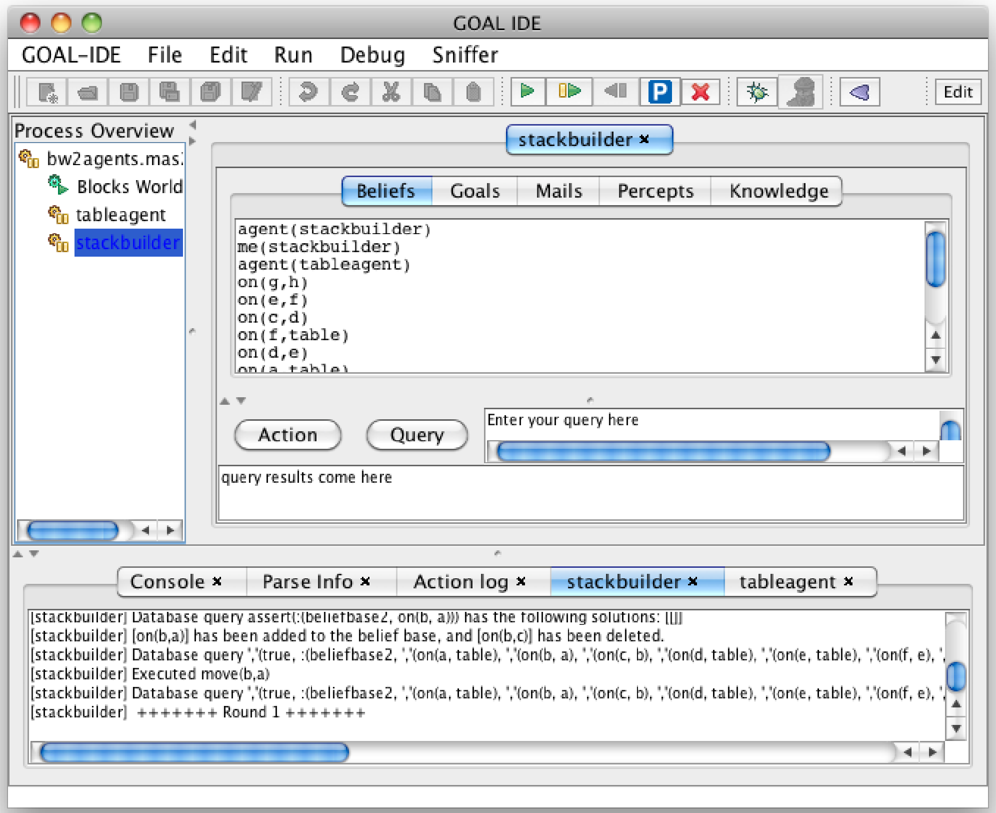
A useful tool to trace what your agents are up to is the *introspector*. An introspector allows you to inspect the mental state of an agent.

How to open an Introspector:

- Open an introspector by double clicking on an agent process in the Process panel, or,
- Alternatively, select the agent process and the Open Introspector item in the Debug menu.

Introspectors are located in the content area of the debug mode. Figure 5.1 shows an introspector for the `stackbuilder` agent that is part of the `bw2agents.mas2g` project.

An agent introspector consists of various tabs that allow you to access the current contents of the agent's beliefs, goals, mails, percepts, and knowledge. In the area showing the contents, you can select text and copy it to somewhere else.



```
agent(stackbuilder)
me(stackbuilder)
agent(tableagent)
on(g,h)
on(e,f)
on(c,d)
on(f,table)
on(d,e)
on(a,table)
```

```
[stackbuilder] Database query assert(:(beliefbase2, on(b, a))) has the following solutions: [[]]
[stackbuilder] [on(b,a)] has been added to the belief base, and [on(b,c)] has been deleted.
[stackbuilder] Database query ','(true, :(beliefbase2, ','(on(a, table), ','(on(b, a), ','(on(c, b), ','(on(d, table), ','(on(e, table), ','(on(f, e), ',
[stackbuilder] Executed move(b,a)
[stackbuilder] Database query ','(true, :(beliefbase2, ','(on(a, table), ','(on(b, a), ','(on(c, b), ','(on(d, table), ','(on(e, table), ','(on(f, e), ',
[stackbuilder]  +++++++ Round 1 +++++++
```

Figure 5.1: An Introspector for the `stackbuilder` agent

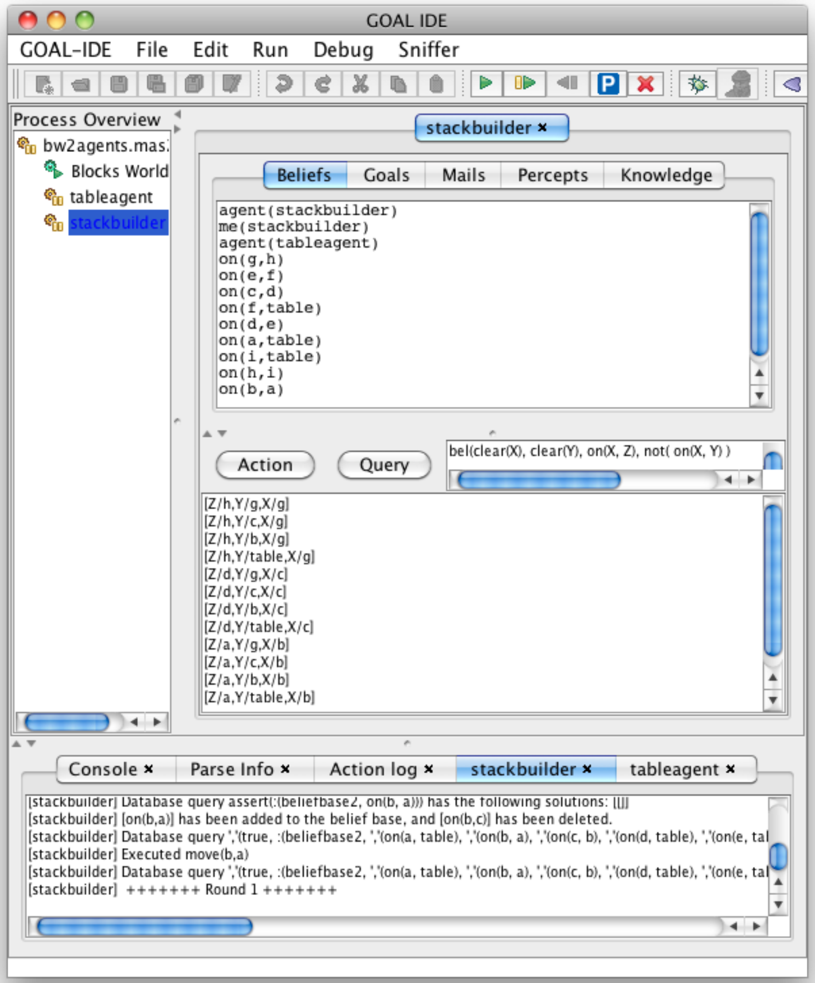
5.2 Querying a Mental State

Apart from being able to view the contents of an agent's mental state when it is paused, an introspector also allows you to pose queries for evaluation on the current mental state of the agent in the Query Area.

How to evaluate mental state conditions:

- Enter a mental state condition in the Query Area.
- Press the Query button.

All solutions for the query performed are shown in the query window. They are shown in the query result area as a list of substitutions for variables. Figure 5.2 shows the results of performing the query `bel(clear(X), clear(Y), on(X, Z), not( on(X, Y) ))`. This mental atom corresponds to the precondition of the `move` action in the Blocks World environment and shows all the options to move blocks that are available in the present state. Consult the GOAL Programming Guide [3] for a detailed explanation of mental state conditions.


GOAL IDE
GOAL-IDE File Edit Run Debug Sniffer
Process Overview
bw2agents.mas
Blocks World
tableagent
stackbuilder
Beliefs Goals Mails Percepts Knowledge
Action
Query
Console Parse Info Action log stackbuilder tableagent

```
agent(stackbuilder)
me(stackbuilder)
agent(tableagent)
on(g,h)
on(e,f)
on(c,d)
on(f,table)
on(d,e)
on(a,table)
on(i,table)
on(h,i)
on(b,a)
```

```
bel(clear(X), clear(Y), on(X, Z), not( on(X, Y) )
```

```
[Z/h,Y/g,X/g]
[Z/h,Y/c,X/g]
[Z/h,Y/b,X/g]
[Z/h,Y/table,X/g]
[Z/d,Y/g,X/c]
[Z/d,Y/c,X/c]
[Z/d,Y/b,X/c]
[Z/d,Y/table,X/c]
[Z/a,Y/g,X/b]
[Z/a,Y/c,X/b]
[Z/a,Y/b,X/b]
[Z/a,Y/table,X/b]
```

```
[stackbuilder] Database query assert(:(beliefbase2, on(b, a))) has the following solutions: [[]]
[stackbuilder] [on(b,a)] has been added to the belief base, and [on(b,c)] has been deleted.
[stackbuilder] Database query ','(true, :(beliefbase2, ','(on(a, table), ','(on(b, a), ','(on(c, b), ','(on(d, table), ','(on(e, tal
[stackbuilder] Executed move(b,a)
[stackbuilder] Database query ','(true, :(beliefbase2, ','(on(a, table), ','(on(b, a), ','(on(c, b), ','(on(d, table), ','(on(e, tal
[stackbuilder]  +++++++ Round 1 +++++++
```

Figure 5.2: A sample query.

5.3 Executing an Action using the Introspector

You can also execute actions from the Query area within the Introspector. The actions that can be executed include (i) mental state actions which change the mental state of the agent (**adopt**, **drop**, **insert**, **delete**) and (ii) user specified environment actions that are executed in the environment. You cannot execute other actions such as combined actions, sending messages, or modules.

How to execute an action using the Introspector:

- Enter the action into the Query area.
- Press the Action button.

Make sure that the action is closed, i.e., has no free variables. To execute an environment action, make sure that the environment is running. You may also need to figure out whether the agent that should execute the action is allowed to perform an action at the time you want to perform the action, e.g., it is the turn of that agent.

Note that an environment action is sent to the environment without checking its preconditions. The postcondition of the action also is not processed nor are any percepts related to the action passed to the agent. Finally, no feedback is provided whether an action was successfully performed or not.

5.4 Debugging Output

In debug mode, the feedback area in the runtime contains various tabs for inspecting agents and the actions they perform, see Figure 4.1. Besides the console tab, an action log tab is present that provides an overview of actions that have been performed by all (running) agents. In addition, when an agent is launched a *debug tracer* tab is added to the feedback area for inspecting various aspects of the agent program during runtime.

It is sometimes useful to clean the contents of a tab. This can be done by right-clicking (ctrl-click on OSX) and selecting the Clear option. This is particularly useful for the action log, which is never automatically cleared.

Console

The console shows all important messages, warnings and errors that may occur. These messages include those generated by Goal at runtime, but possibly also messages produced by environments and other components. When something goes wrong, it is always useful to inspect the console tab for any new messages. If a new message is present in the console tab, the color of the tab is changed to indicate this.

Action Log

The action log tab shows the actions that have been performed by running agents that are part of the multi-agent system. These actions include both user-defined actions that are sent to the environment as well built-in actions provided by Goal.

Apart from an overview of what has been done by agents, the action log can also be used as a *simple means for debugging*. It can be used, for example, to check whether a particular action added for debugging has been performed. The idea is that you can add a dummy action with a particular content message to a rule and verify in the action log whether that action has been performed. To do so, include an action in the action specification of your agent named e.g. `debug(<yourMessage>)` with `true` as pre- and post-condition to ensure the action always can be performed and will not change the state of the agent but still will be printed in the action log output.

Debug Tracer

Upon launching an agent, a debug tracer tab is added for that agent in the feedback area. This tab typically contains more detailed information about your agent. To avoid flooding the tab, it is only filled, however, when the multi-agent system is paused. To be useful, the contents of the tab need to be analyzed which cannot be done while new content would be added continuously.

The exact contents of the tab can be customized by means of the Help/Preferences/Breakpoints settings menu. Various items that are part of the reasoning cycle of an agent can be selected for viewing. If checked, related reports will be produced in the debug tracer of an agent. A more detailed explanation of these breakpoints can be found in Section 5.5.

If you step a multi-agent system using the Step button ▶, the runtime by default switches to the debug tracer of the agent that is being stepped. This allows you to automatically keep track of what happens without having to manually switching between agent debug tracer tabs. The default switching between debug tracers can be turned of by means of the Help/Preferences/GUI menu; uncheck the option at the bottom labeled *Automatically switch to Debug Tracer tab.*

Debug tracer tabs can be closed by clicking the close icon on of the tab. You can re-open a closed tab by selecting the agent in the process panel and righ-clicking (ctrl-click on OSX) and selecting Open Debug Tracer.

Customizing the Logging

The logging behaviour of the runtime can be modified from the preferences in Eclipse. Access them through opening the GOAL category in Window → Preferences →, and selecting the Logging page.

A timestamp with millisecond accuracy can be added to each printed log by checking the "Show log time" checkbox.

The 'write logs to file' checkbox determines if GOAL writes all logs to separate files. Logs for each agent, the action log, warnings, results from GOAL log actions etc are all written to separate files. Log files are written to the logs directory inside the directory where GOAL was installed. You can change it by clicking on the Browse button of the "Write log files here" field.

Most logs are in XML format, and each log entry contains a timestamp with millisecond accuracy. Debug messages for each ¡agent¿ are written to <agent>.log. All actions are written to historyOfActions.log. General info is written to goal.tools.logging. InfoLog.log. Warnings are written to goal.tools.errorhandling. Warning.log. Sytem bug reports are written to goal.tools.errorhandling. SystemBug.log. Result of log actions are written to <agent>_<timestamp>.txt files. Be careful with this option, as many megabytes per second can be written. A new run generates new log files (made unique by adding a number to the names). By selecting 'Overwrite old log files', a new run uses the same file names, thus overwriting previously generated files.

The type of debug messages that are logged can be determined with the view checkboxes in the global breakpoints preferences panel (Figure 5.3).

The warnings print format allows you to select the information that is printed along with any warning. You can turn on the stack trace and the detailed java message. Also you can set the maximum number of times the same message is printed, to avoid flooding of the console.

5.5 Setting Breakpoints

A useful way to trace what your agents are doing is to step the multi-agent system and check how and which action rules are evaluated, which actions are performed, which percepts are received, etc. Stepping assumes natural breakpoints to stop an agent at. For GOAL agents, a number of such breakpoints have been pre-defined. These built-in breakpoints facilitate stepping and tracing of what happens when an agent is executed.

A second, alternative method to set breakpoints is by explicitly adding them to code lines in agent program files. A user can set breakpoints at various places in a program where the agent should stop when such a breakpoint is hit.

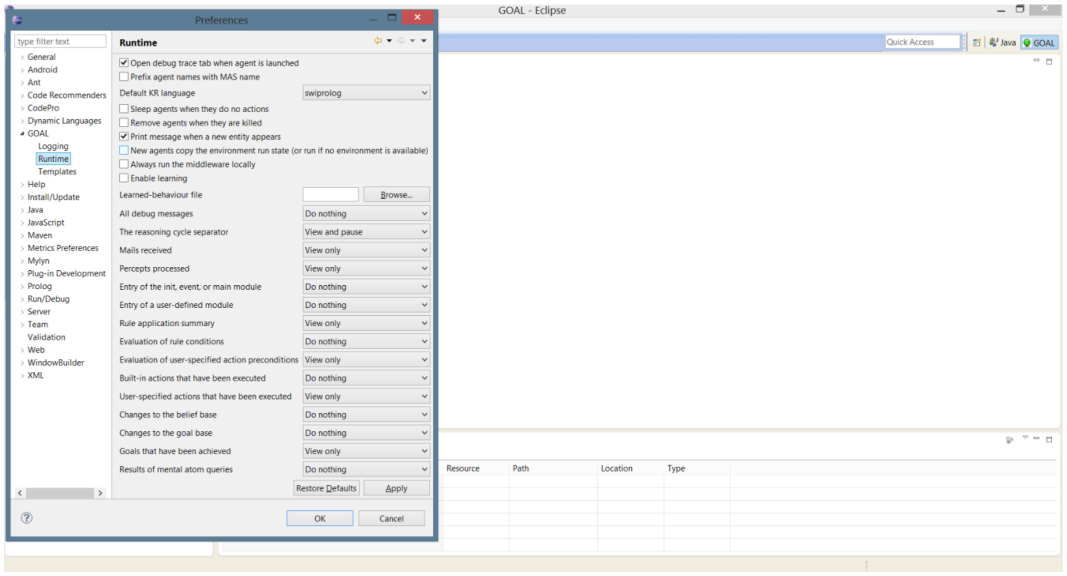
5.5.1 Stepping and Global Breakpoints

The first method to step an agent and halt at breakpoints is based on a set of built-in breakpoints that relate to the reasoning cycle of an agent. A list of these breakpoints is available in Eclipse by opening the GOAL category in Window → Preferences →, and selecting the Logging page.5.3.

For each breakpoint, a selection box with multiple options is available. If View Only is selected, a report related to the breakpoint will be made available in the Debug Tracer of an agent. If View and pause is selected, in addition the creation of a report, all agents will be paused when such a breakpoint is hit. The Do nothing option skips the breakpoint completley.

We briefly discuss each of the available built-in breakpoints:

- The first item *Steps in the reasoning cycle of an agent* relates to e.g. the start of the agent's reasoning cycle. By default a *round separator* is added to the output in the Debug Tracer



Runtime

- [✓] Open debug trace tab when agent is launched
- [ ] Prefix agent names with MAS name

Default KR language	swiprolog

- [ ] Sleep agents when they do no actions
- [ ] Remove agents when they are killed
- [✓] Print message when a new entity appears
- [ ] New agents copy the environment run state (or run if no environment is available)
- [ ] Always run the middleware locally
- [ ] Enable learning

Learned-behaviour file	
All debug messages	Do nothing
The reasoning cycle separator	View and pause
Mails received	View only
Percepts processed	View only
Entry of the init, event, or main module	Do nothing
Entry of a user-defined module	Do nothing
Rule application summary	View only
Evaluation of rule conditions	Do nothing
Evaluation of user-specified action preconditions	View only
Built-in actions that have been executed	Do nothing
User-specified actions that have been executed	View only
Changes to the belief base	Do nothing
Changes to the goal base	Do nothing
Goals that have been achieved	View only
Results of mental atom queries	Do nothing

Figure 5.3: Setting Global Breakpoints

to indicate which reasoning cycle (round) has been executed. This round separator is also available in the action log.

- The items *Mails received* and *Percepts received* concern the events of receiving new mails and percepts. Mails and percepts are buffered and collected at the start of an agent's reasoning cycle. A report containing the percepts and/or messages received is provided as output in the Debug Tracer.

- *Evaluation of action rules* concerns the evaluation of the mental state condition and applicability of action rules. Depending on the rule evaluation order, more or less rules will be evaluated each reasoning cycle. The debug output will show all substitutions for which the rule is applicable, if any.

- *Evaluation of action preconditions* relates to the evaluation of the preconditions of actions. In case this breakpoint is selected to pause on, the agent will halt just before executing the action itself. If this breakpoint is viewed, reports of the evaluation of all action preconditions are provided.

- Besides the action log, which actions are executed can be provided as debug output in the Debug Tracer as well by checking the View check box related to either the *Built-in actions* or *Actions sent to the environment* breakpoints.

- The *Changes to the belief base* and *changes to the goal base* breakpoints provide output related to or pause when a change to either base is made. The *Changes to the belief base* breakpoint usually is a too fine-grained breakpoint to pause on as many changes to a belief base are made.

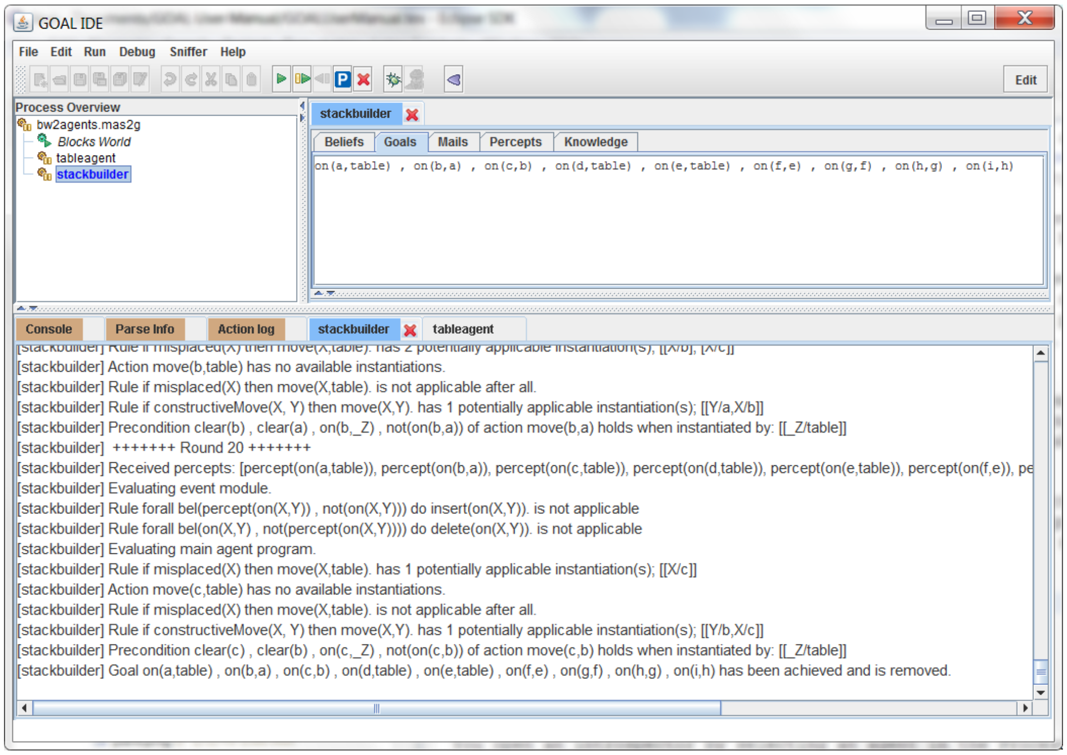
- The *Goals that have been achieved* breakpoint can be quite useful to verify whether a goal has been achieved, and to halt an agent whenever this happens.

- The last two breakpoints are the most finegrained-level breakpoints available and typically are not very useful. They provide detailed output on basic queries at the level of mental atoms and/or queries performed using the knowledge representation language (e.g. Prolog).

You can modify the selections at any time, also during the execution of a mas. If you are stepping a multi-agent system, agents will continue to run until they hit the next breakpoint selected to pause on. If such a breakpoint is hit, the agent is switched to pause mode.

Note that the Step button has the same effect as the Run button if no breakpoints have been selected. In that case, the agent progress will be run without halting the agent, with the exception of explicitly set breakpoints which we discuss next.

As an example, clear all Pause check boxes in the Breakpoints panel, and only select to pause on the *Goals that have been achieved* breakpoint. Launch the [bw2agents.mas2g] mas and press the Step button (several times to get things started). The agents run until the `stackbuilder` agent achieves its goal, which demonstrates the agent is able to achieve it even in the presence of the `tableagent` agent. Figure 5.4 shows an example run where this happens in round 20.


GOAL IDE
File Edit Run Debug Sniffer Help
Edit
Process Overview
bw2agents.mas2g
Blocks World
tableagent
stackbuilder
stackbuilder
Beliefs Goals Mails Percepts Knowledge
Console Parse Info Action log stackbuilder tableagent

```
on(a,table) , on(b,a) , on(c,b) , on(d,table) , on(e,table) , on(f,e) , on(g,f) , on(h,g) , on(i,h)
```

```
[stackbuilder] Action move(b,table) has no available instantiations.
[stackbuilder] Rule if misplaced(X) then move(X,table). is not applicable after all.
[stackbuilder] Rule if constructiveMove(X, Y) then move(X,Y). has 1 potentially applicable instantiation(s); [[Y/a,X/b]]
[stackbuilder] Precondition clear(b) , clear(a) , on(b,_Z) , not(on(b,a)) of action move(b,a) holds when instantiated by: [[_Z/table]]
[stackbuilder]  +++++++ Round 20 +++++++
[stackbuilder] Received percepts: [percept(on(a,table)), percept(on(b,a)), percept(on(c,table)), percept(on(d,table)), percept(on(e,table)), percept(on(f,e)), pe
[stackbuilder] Evaluating event module.
[stackbuilder] Rule forall bel(percept(on(X,Y)) , not(on(X,Y))) do insert(on(X,Y)). is not applicable
[stackbuilder] Rule forall bel(on(X,Y) , not(percept(on(X,Y)))) do delete(on(X,Y)). is not applicable
[stackbuilder] Evaluating main agent program.
[stackbuilder] Rule if misplaced(X) then move(X,table). has 1 potentially applicable instantiation(s); [[X/c]]
[stackbuilder] Action move(c,table) has no available instantiations.
[stackbuilder] Rule if misplaced(X) then move(X,table). is not applicable after all.
[stackbuilder] Rule if constructiveMove(X, Y) then move(X,Y). has 1 potentially applicable instantiation(s); [[Y/b,X/c]]
[stackbuilder] Precondition clear(c) , clear(b) , on(c,_Z) , not(on(c,b)) of action move(c,b) holds when instantiated by: [[_Z/table]]
[stackbuilder] Goal on(a,table) , on(b,a) , on(c,b) , on(d,table) , on(e,table) , on(f,e) , on(g,f) , on(h,g) , on(i,h) has been achieved and is removed.
```

Figure 5.4: Pausing on a Goal Achieved Breakpoint

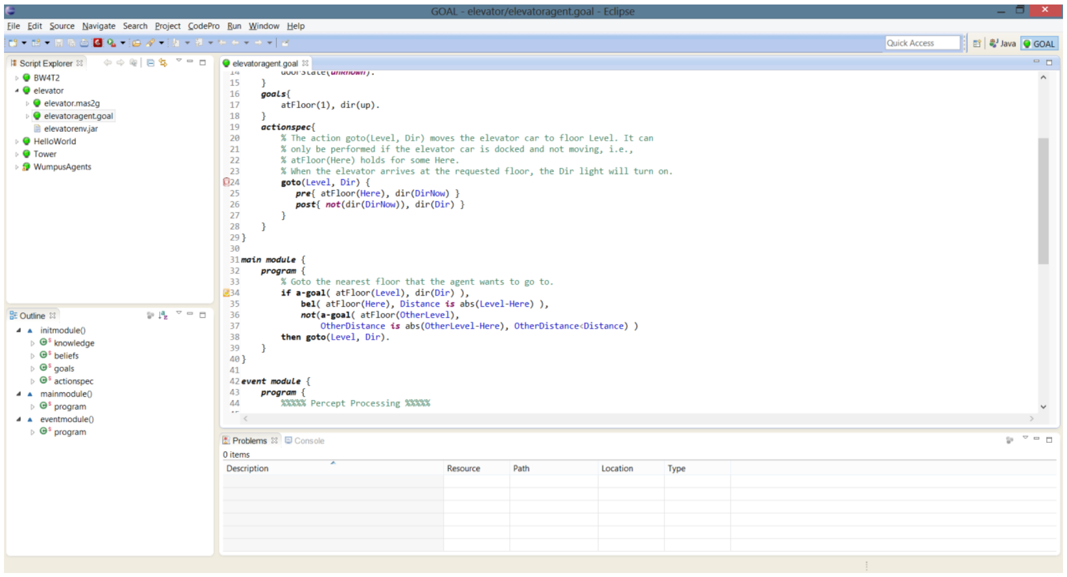
5.5.2 Setting Code Line Breakpoints

A second, more flexible way of setting breakpoints, is by adding code line breakpoints. In GOAL, breakpoints can be associated with four constructs in an agent program: a module, the condition of an action rule, the action part of an action rule, and an action specification. Breakpoints can be added to these constructs by opening a mas project file in Eclipse, and adding breakpoints at *the first line associated with the construct*. That is, put a breakpoint next to the line with the module name, the first line associated with an action rule, or the the line with the action name. You can do this by double-clicking in the left border of the editor.

GOAL will automatically allocate placed breakpoints to their appropriate position when the MAS is started by attaching any placed breakpoint to the first suitable location after its current position.

There are two types of breakpoints: the ALWAYS and the CONDITIONAL type. A CONDITIONAL breakpoint shows as an orange icon in the left border. It can only attach to the action part of an action rule. The ALWAYS type shows as a red icon in the left border, and attaches to the other three types.

An example of two breakpoints that were added to the `move` action specification is illustrated in Figure5.5.



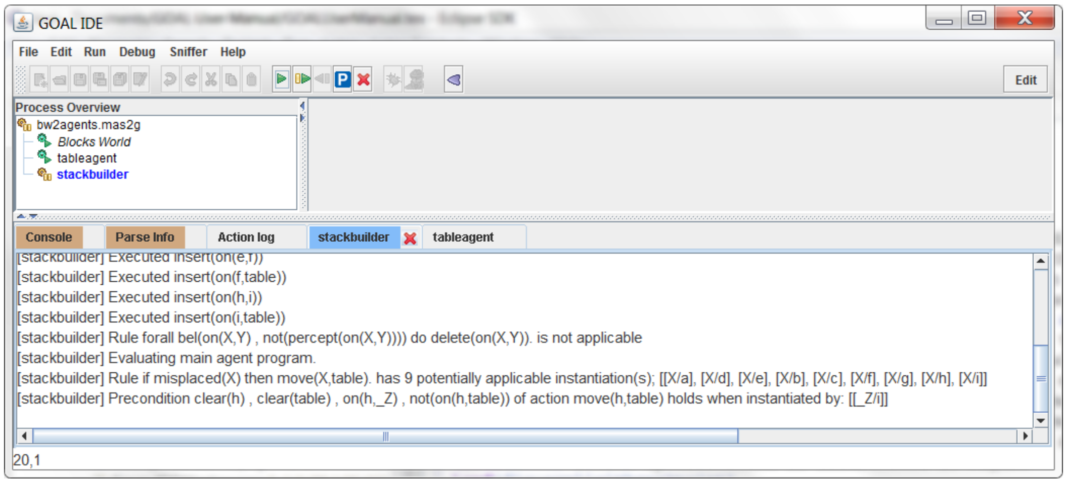
```
    }
    goals{
        atFloor(1), dir(up).
    }
    actionspec{
        % The action goto(Level, Dir) moves the elevator car to floor Level. It can
        % only be performed if the elevator car is docked and not moving, i.e.,
        % atFloor(Here) holds for some Here.
        % When the elevator arrives at the requested floor, the Dir light will turn on.
        goto(Level, Dir) {
           pre{ atFloor(Here), dir(DirNow) }
           post{ not(dir(DirNow)), dir(Dir) }
        }
    }
}

main module {
    program {
        % Goto the nearest floor that the agent wants to go to.
        if a-goal( atFloor(Level), dir(Dir) ),
           bel( atFloor(Here), Distance is abs(Level-Here) ),
           not(a-goal( atFloor(OtherLevel),
               OtherDistance is abs(OtherLevel-Here), OtherDistance<Distance) )
        then goto(Level, Dir).
    }
}

event module {
    program {
        %%%%% Percept Processing %%%%%
```

0 items

Description	Resource	Path	Location	Type

Figure 5.5: An ALWAYS and a CONDITIONAL Breakpoint added to the`elevator`

Different from the built-in breakpoints, even if a multi-agent system is put in run mode, the system will halt on any explicit breakpoints that have been set. Figure 5.6 shows that the `stackbuilder` is paused when the breakpoint set on the action specification is hit.



```
[stackbuilder] Executed insert(on(e,f))
[stackbuilder] Executed insert(on(f,table))
[stackbuilder] Executed insert(on(h,i))
[stackbuilder] Executed insert(on(i,table))
[stackbuilder] Rule forall bel(on(X,Y) , not(percept(on(X,Y)))) do delete(on(X,Y)). is not applicable
[stackbuilder] Evaluating main agent program.
[stackbuilder] Rule if misplaced(X) then move(X,table). has 9 potentially applicable instantiation(s); [[X/a], [X/d], [X/e], [X/b], [X/c], [X/f], [X/g], [X/h], [X/i]]
[stackbuilder] Precondition clear(h) , clear(table) , on(h,_Z) , not(on(h,table)) of action move(h,table) holds when instantiated by: [[_Z/i]]
```

Figure 5.6: The `stackbuilder` agent paused on an explicit breakpoint

Breakpoints that are associated with action specifications and action rules provide useful output in the Debug Tracer immediately. This is not the case, however, for breakpoints set on modules. The output produced in this case only provides a clue that the module is about to be evaluated. Take it from here to step further through the module using the Step button to produce more relevant debug output.

Explicit breakpoints that have been added to a file are not permanently saved. Upon closing Eclipse, all breakpoints set are lost. This is an easy way to clear all breakpoints. Individual breakpoints can be cleared by double-clicking them again; Eclipse will toggle between the two different types of breakpoint, and no breakpoint at all.

5.6 Prolog Level Debugging

GOAL does not support tracing of Prolog queries inside the Prolog inference engine. If you need this level of detail while you are debugging an agent, it is useful to be able to export the contents of a mental state to a separate file that can be imported into Prolog. Then use the Prolog tracer to analyze the code.

5.6.1 Exporting a Mental State to a File

To export the mental state of a GOAL agent, select the agent process and then select Debug/Export mental state. This allows you to save the contents of a mental state as a Prolog file.

The exported file(s) can then be imported into a Prolog debugger. SWI Prolog can be used for this, provided that you installed SWI Prolog on your machine. The version of SWI Prolog that is distributed with GOAL cannot be used for this purpose.

A detailed set of instructions on how to do this follows:

1. select and pause the agent just before the point where you would like to debug a call to Prolog

2. select menu Debug/Export Mental State contents and save the database contents as a `.ms` (MentalState) file. You can save beliefs and goals to a separate file and opening the exported file in the IDE, but for opening in Prolog, export to a single file seems most useful. After clicking OK a file browser appears; select a good place to save the file.

3. Manually edit the `.ms` files:

   (a) They need to have the extension `.pl` to import in prolog

   (b) All predicates that you use but do not have explicit values in the prolog program may need to be declared dynamic. See the SWI Prolog Manual for details.

   (c) Goals are dumped to the `.ms` files as a conjunct of predicates, and this is not a legal notation in Prolog; the least you have to do is to replace all commas ',' within a single goal with dots '.'.

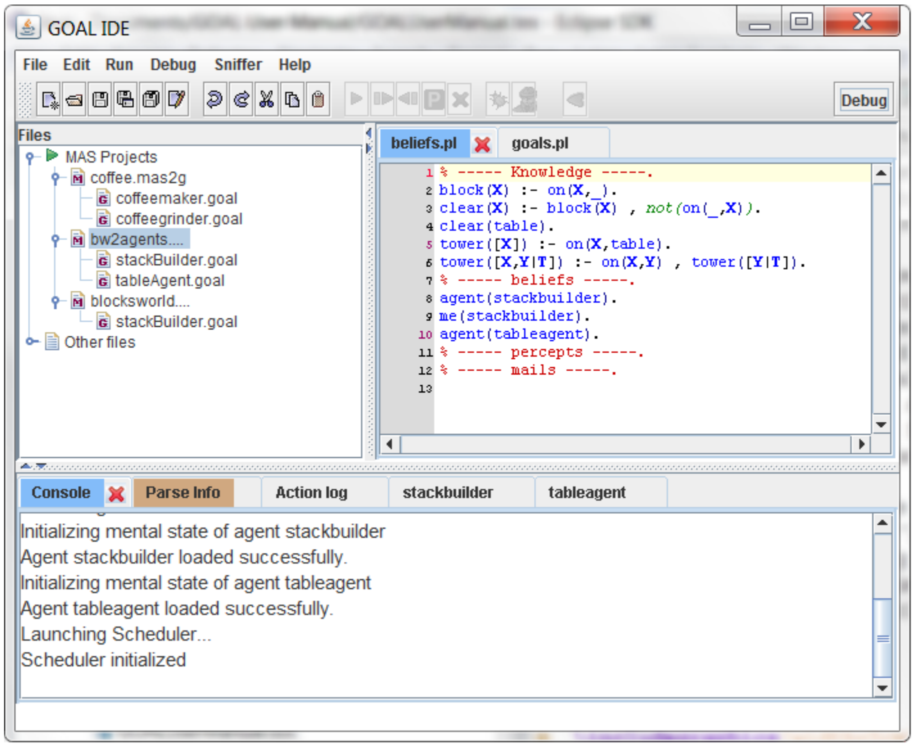
   (d) You may need to fix the goals. Several goals may be in conflict when seen from SWI Prolog and therefore you may have to remove part of the goals.

4. open SWI Prolog (see the SWI Prolog manual) and import the Prolog file (e.g. `consult('<filename>').`)

5. start the SWI Prolog tracer (e.g. type `trace.` in its console)

6. run and trace the query (see SWI Prolog manual, typically the 'n', 'a' and 'enter' are used for tracing)

If you selected opening of files while exporting the mental state contents, these files are opened in edit mode. Switch back to edit mode in order to view the contents. Figure 5.7 shows the results for the `stackbuilder` agent.



```
% ----- Knowledge -----.
block(X) :- on(X,_).
clear(X) :- block(X) , not(on(_,X)).
clear(table).
tower([X]) :- on(X,table).
tower([X,Y|T]) :- on(X,Y) , tower([Y|T]).
% ----- beliefs -----.
agent(stackbuilder).
me(stackbuilder).
agent(tableagent).
% ----- percepts -----.
% ----- mails -----.
```

```
Initializing mental state of agent stackbuilder
Agent stackbuilder loaded successfully.
Initializing mental state of agent tableagent
Agent tableagent loaded successfully.
Launching Scheduler...
Scheduler initialized
```

Figure 5.7: Export dialog for mental state contents

5.6.2 Prolog Exceptions

When an exception occurs in the Prolog engine (SWI Prolog), error messages may get a bit hard to decipher. This section explains how to read such messages.

A typical PrologException will show up typically like this:

```
WARNING:  jpl.PrologException: PrologException:
          error(instantiation_error, context(:(system, /(>=, 2)), _1425))
query=true,beliefbase6: (openRequestOnRoute(L,D,T))
```

The first of the two lines indicates the message that Prolog gave us. The second line shows the query that caused the warning. To read the first line, notice the part `error(CAUSE,DETAILS)` inside it.

The CAUSE part gives the reason for the failure, the DETAILS gives the context of the error. Several types of CAUSE can occur, amongst others:

1. type_error(X,Y): an object of type X was expected but Y was found. For instance, in the context of arithmetic X can be 'evaluable' and Y a non-arithmic function.

2. instantiation_error: the arguments for an operator are insufficiently instantiated. For example, if we ask X>=8 with X a free variable, we will get an error like the example above. The context in such a case would be `context(:(system, /(>=, 2)), _1425)`. The first part, `system:>=/2` refers to the >= operator (which has arity 2 and is part of the system module) and the _1425 refers to the free variable (which apparently was even anonymous hence is not of much help). Note that these messages commonly contain pure functional notation where you might be more used to the infix notation. For instance `system:X` appears as `:(system,X)` and >=/2 appears as /(>=,2).

3. representation_error: the form of the predicate was not appropriate, for instance a bound variable was found while an unbound variable was expected

4. permission error: you have insufficient permissions to execute the predicate. Typically this occurs if you try to redefine system predicates.

5. resource_error: there are not enough resources of the given type.

5.7 Runtime Preference Settings

Most settings can be modified from the preferences in Eclipse. Access them through opening the GOAL category in Window → Preferences →, and selecting the Runtime page. This section discusses those settings.

5.7.1 Platform Management

The tab "PlatformManager" in the Help/Preferences menu offer a number of options to set the base path for the environment and middleware.

This preference window offers the option to always run the middleware locally. This checkbox is by default disabled and therefore when starting a mas you will always be prompted for the location of the middleware host. If you turn on this checkbox, the default "localhost" will be used as middleware host and the dialog will not be shown.

You can adjust the directory where browsing for `.mas2g` and `.goal` files will start, and whether your last directory should be used as the starting point for browsing.

The 'default KR Language' selection allows you to choose between the default swiprolog implementation and an alternative swiprolog2 implementation. swiprolog2 attempts to compile mental state queries into a single query to the underlying swi prolog engine. This implementation is not fully tested, and it is recommended to use swiprolog.

Selecting the 'sleep agents when they repeat same action' enables GOAL to skip agents that seem to be idle, which relocates CPU time to other agents that really are working on something. GOAL guesses that an agent is idle if the agent is receiving the same percepts and mails as last step on the input, and picks the same actions as output. The sleep is cancelled when the percepts/mails of the agent change. This setting is on by default.

IMPORTANT: with 'sleep agents' selected, your agent may go to sleep when it is repeating actions. This may be counterintuitive, and may cause GOAL programs to malfunction. In doubt, you should turn off this option.

To remove agents automatically after they have been killed, select the "Remove agents when they are killed" option. To run new agents automatically as their entities appear, select "Run agents automatically when the environment is already running" These two options are useful in environments where large numbers of agents appear and disappear while GOAL is running.

5.7.2 Warnings

Warnings and errors are shown depending on how you set the preferences. Warnings generally appear in the Console area (see Section 5.4).

By default, warnings are plain messages. For development purposes, however, the format of warnings can be adjusted using the Logging tab in the Preferences menu. Select to *Show Java details* to include the Java-specific part of a warning message. In some cases this can help to see what the platform was attempting to do. Select *Show Java stack traces* to add a full stack dump to warning messages. This is mainly intended to aid debugging of the platform itself but in some cases may also help you to understand what the platform was trying to do.

Reporting Issues:
When reporting an issue, it is very useful if your issue report **includes Java details and stack traces**. See Section 8.1 for more about reporting an issue.

When an identical problem re-occurs, a warning message is re-printed only a limited number of times to prevent the Console from being flushed with identical messages. When the Console shows the message for the last time, it will add the message “New occurrences of this warning will not be shown any more”. You can adjust the number of times that a message will be re-printed in the Logging tab in the Preferences menu.

Chapter 6

Running in Stand-Alone Mode

You can run a multi-agent system containing GOAL agent programs stand-alone, that is without starting the IDE or Eclipse. That way you can use GOAL from scripting languages like a shell script or bat script, or from your Java program.

Stand-alone running always involves starting up a Java Virtual Machine (JVM) with the right environment settings. We first describe the required environment and then describe how to use this from a Java program or from a script.

6.1 Setting the JVM environment

The variables to be set up properly for the environment are the `PATH`, the `JAVA_LIBRARY_PATH`, `CLASSPATH` and the `SWI_HOME_DIR`. Below we use the tag `GOALHOME` to point to the absolute directory where you installed GOAL. The exact way to do this is machine dependent; we assume you have the knowledge how to do this on your particular machine. For example, the ';' used as separator in the `CLASSPATH` below should be replaced with a ':' on Linux and OSX.

- on Windows, `PATH` should be set to include `GOALHOME/swifiles/libs`. On Linux you should set `LD_LIBRARY_PATH` instead, and on OSX you should set `DYLD_LIBRARY_PATH`.
- `SWI_HOME_DIR` should be set to point to `GOALHOME/swifiles/`.
- `JAVA_LIBRARY_PATH` should be set to include `GOALHOME/swifiles/libs`.
- `CLASSPATH` should be set to include `GOALHOME/goal.jar` and to the (EIS)environment jar file that you want to use.

You can inspect the goal.bat (Windows) or goal.sh file (Linux, OSX) to see a running example of the above.

6.2 From Java

Running a stand-alone MAS from Java is very simple: just create the goal.tools.Run object from Java and start it using this call

```
new Run(new File(fileName)).run();
```

where fileName is the MAS file that you want to run.

6.3 From the Command Line

To run GOAL stand-alone from the command line, the typical call looks like this:

```
java -cp goal.jar -Djava.library.path=swifiles/libs
         goal.tools.Run <filename.mas2g> [wait]
```

where `<filename.mas2g>` is the **absolute** path to the MAS file you want to run. You can add the `wait` option so that the runner waits between starting the environment and scheduling the GOAL agents, see Section 6.5 below.

-d,--debug	Display out put from debugger while running agent
-e	Print messages from error
-h,--help	Displays this help
-i	Print messages from info
--license	Shows the license
-p	Print messages from parser
-r,--repeats ¡number¿	Number of times to repeat running all episodes
--recursive	Recursively search for mas files
--rmi ¡host¿	Use RMI messaging middleware. Host is the location of the RMI server. Using ”localhost” will initialize a RMI server
--timeout ¡seconds¿	Timeout of the multi-agent system.
-v,--verbose	Print all messages
--version	Shows the current version
-w	Print messages from warning

6.4 Runtime Settings

Some settings, e.g. used middleware, are used as set in the preferences. Note that these preferences can be set using the GOAL IDE, but also by direct calls. The raw settings.yaml file, that is used for persisting the preferences, can also be edited directly.

Please refer to the JavaDoc for the various preference settings inside the `goal.preferences` directory. There are preference settings for the CorePreferences, PMPreferences, the Logging-Preferences etc. Also notice the -w and -v options of the tool if you are looking for warning printouts.

6.5 Interaction with Standalone

In some cases, user interaction is still required in a stand-alone run. Usually this is because the environment needs additional setup actions before it can really be used..

Termination does not always imply reset of the environment. You may have to set init parameters in the MAS to request reset of the environment for the next run.

6.6 Batch Running

The `Run` command above allows you various variants for batch running. You can specify a directory containing multiple prepared MAS files, and/or you can request multiple runs of the MAS files.

To enable proper batch running, your setup has to meet two requirements:

- Your MAS terminates at some point. This happens when all agents have terminated, e.g. exited their main program section or were killed.
- Your environment resets the entities when it is killed and then restarted. This may seem natural but in some configurations (eg, BW4T2), killing and restarting a local environment does not necessarily reset the real remote environment.

Chapter 7

Creating Your Own Environments

7.1 Environments Distributed with GOAL

GOAL is distributed with several environments for which you can write your own agents. These environments can be found in the `GOALagents` folder in the main `GOAL` folder in which GOAL is installed. Running these environments is explained in Chapter 4.

7.2 Creating an Environment Interface

Environments provide so-called *controllable entities*. Agents can connect via an interface to these entities and control their behaviour. The GOAL Programming Guide [3] explains in more detail how this works.

The interface used by GOAL is the Environment Interface Standard (EIS; [2, 1]). The current version of the GOAL platform is fully compliant with EIS v0.3. EIS is an Open Source project and documentation on how to create an interface for your environment such that it can operate in combination with GOAL is available via `http://sourceforge.net/projects/apleis`. Additional guidelines for creating an environment can be found on the website for GOAL: `http://ii.tudelft.nl/trac/goal/wiki/Developers/Environments/HowToEISifyAnEnvironment`.

Chapter 8

Known Issues and Workarounds

Below we have listed some known issues and possible workarounds when available.

- **Nothing happens when pressing the Step and Run buttons**.
  There are a number of possible explanations for this you can check.

  1. First, make sure that the environment is ready to run. For example, when using the Elevator environment the environment will respond to GOAL only if you selected *GOAL Controller* available in this environment.
  2. You may be stepping an agent that is suspended somehow, e.g. the environment may be refusing to serve that agent, or it may be the turn of another agent.

- **The environment does not work as expected**.
  Check the documentation that comes with the environment. Most environments need to be set up before the agents can connect and perform actions.

- **The Wumpus environment hangs when the agent shoots an arrow**.
  The Wumpus World always has to be enclosed by means of walls. If an agent shoots an arrow into open space, the arrow will continue forever, causing the system to hang.

- **The Reset button does not work**.
  It is not always possible to reset an environment as an environment may not provide the support for doing so.

- **I do not see an environment after launching a MAS**.
  The environment may be hidden behind another window. This is due to a known bug in Java for Windows. You need to install Java 7 or higher to fix this issue. Java 7 is available from `http://download.java.net/jdk7/binaries/`.

8.1 Reporting Issues

If you discover anything you believe to be an error or have any other issues you run into, please report the error or issue to goal@ii.tudelft.nl. Your help is appreciated!

To facilitate a quick resolution and to help us reproduce and locate the problem, please include the following information in your error report:

1. A short *description* of the problem.
2. All relevant *source files* of the MAS project.
3. Relevant warning or error *messages from the Feedback area*, especially from the Console and/or Parse Info tab. Please also include *Java details and stack trace information* if available (see Section 5.7.2 for adjusting settings to obtain this information).
4. If possible, a *screen shot* of the situation you ran into.

Bibliography

[1] Tristan Behrens. Environment interface standard project. `http://sourceforge.net/projects/apleis/`.

[2] Tristan Behrens, Koen V. Hindriks, and Jrgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, pages 1–35, 2010. 10.1007/s10472-010-9215-9.

[3] Koen V. Hindriks. *GOAL Programming Guide*. Delft University of Technology, 2014. `http://ii.tudelft.nl/trac/goal/wiki/WikiStart#Documentation`.

[4] Koen V. Hindriks. The Goal Agent Programming Language. `http://ii.tudelft.nl/trac/goal`, 2014.