Goal-Eclipse User Manual

Vincent J. Koeman, Wouter Pasman, and Koen V. Hindriks

June 21, 2021

Contents

Chapter 1

Introduction

This user manual describes and explains how to install and use the GOAL plug-in for the Eclipse IDE. This includes creating, running, debugging, and testing multi-agent systems. This document does not introduce the agent programming language GOAL itself, nor does it discuss how to write tests, for which we refer the reader to the GOAL Programming Guide [1].

GOAL and the GOAL-Eclipse plug-in are open source software, available at `https://bitbucket.org/goalhub`. A general technical overview of the IDE can be found in [2].

Chapter 2

Installing the GOAL Platform

This chapter explains how to install the GOAL plug-in for the Eclipse IDE. 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. The GOAL platform requires Java (Sun or OpenJDK) 8 or later; the latest OpenJDK is recommended.

2.2 Installation

GOAL is available as a plug-in for Eclipse Neon and up. Using the latest available Eclipse version is recommended. The plug-in can be installed as follows:

1. *Install Eclipse*: If you do not have Eclipse (for Java developers) installed yet, download and run the installer from `https://www.eclipse.org/downloads`.
2. *Start Eclipse*: If you have not done this yet, select a workspace directory (you will be prompted). Note that special characters in the path of this directory are not recommended.
3. *Add a link to the GOAL repository*: In the Eclipse menu, select 'Help → Install New Software → Add' to add a new repository (with any name you like) using the following URL: `https://goalapl.dev/stable`
4. *Install the GOAL Plugin*: Select the 'GOAL Agent Programming' field (make sure 'Group items by category' is enabled), click Next and follow the installation steps. During installation, ignore (accept) any warnings about unsigned content.
5. *Restart Eclipse*: Restart when the installation is done (you will be prompted).

To use the latest (possibly unstable) development version of the plug-in, replace the URL in step 3 with: `https://goalapl.dev/beta`.

After having installed the plug-in, a new 'GOAL Perspective' is now available . Switch Eclipse to this perspective for the first time through 'Windows → Open Perspective → Other → GOAL'. A shortcut will be placed in the top-right corner. Furthermore, we recommend changing the following preferences in Eclipse ('Window → Preferences'):

- General → Workspace → Refresh using native hooks or polling → Enable
- Install/Update → Automatic Updates → Automatically find new updates and notify me → Enable
- Run/Debug → Console → Console buffer size (characters)→ Set to 999999 (the maximum)

Please note that if you are having performance issues whilst debugging, e.g. with many agents, the console buffer size can be reduced. The consoles for each agents and/or the action history can even be entirely disabled in the 'GOAL → Logging' preference category, where the logging itself can be fully tweaked as well.

2.3 Uninstalling

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

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

2.4 Upgrading

Upgrading the GOAL plug-in to a newer version, when not using Eclipse's automatic update functionality, is possible in Eclipse by selecting: 'Help → Check for Updates', and following the steps in the resulting dialog(s).

2.5 Customizing

Various GOAL platform features can be customized.

1. Select Window → Preferences, and unfold the GOAL menu item.
2. Select one of the three categories (Logging, Runtime, or Templates) to edit the related settings.
3. Save your changing by clicking on the 'Apply' button.

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:

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

It is also possible to start-off with one of the many example projects 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. Warnings or errors are displayed because these sections are initially empty; you should adapt the content of these sections of the MAS file to your needs. Warnings and errors are shown at the line at which they occur, underlining the specific text causing the issue. In addition, the *Problems* tab (at the bottom) 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.

3.1.1 Adding an Agent File to a MAS Project

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

1. 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.
2. Give a (unique) name to the agent file, and press 'Finish'.

Do not forget to add the agent file to the MAS file if required.

A new agent file is automatically loaded with a corresponding template. Again, warnings or errors are displayed for a new agent file as the template needs to be completed.

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.

1. 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.
2. Select 'General → File System', and press 'Next'.
3. Browse for the directory the environment file is in. In the right pane, the desired file(s) can now be selected.
4. After having selected the right file(s), press 'Finish'.

Do not forget to add the environment file to the MAS file if required, including any initialization parameters (check the documentation that comes with the environment).

Note that you can also manually place the environment file(s) 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.

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 [1]. 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.

3.1.3 Creating Knowledge, Module and Action Specification Files

When you start writing larger agent programs, it is often useful to add more structure to the MAS project. An important reason for moving code to separate files and importing those files is to facilitate **reuse**. Below we detail the procedure for creating (Prolog) *knowledge files* with the extension `.pl`.

How to create an external knowledge file:

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

Do not forget to reference the knowledge file in the module(s) where this is required.

A procedure very similar to that for knowledge files can also be used to create *module files* with the extension `.mod2g`.

How to create a module file:

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

Do not forget to reference the module file in the MAS (if it is top-level) or any module file where this is required.

Again, a similar procedure can be used to create *action specification files* with the extension `.act2g`.

How to create an action specification file:

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

Do not forget to reference the action specification file in any module file where this is required.

3.2 Importing an Existing Multi-Agent Project

Importing an existing multi-agent system project can be done by means of the following steps:

How to import an existing project into Eclipse:

1. Choose 'File → Import', select 'Existing GOAL Project' (in the 'GOAL Agent Programming' category), and press 'Next'.
2. Browse for the `.mas2g` file that defines the multi-agent system that is to be imported.
3. 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.
4. Press 'Finish'.

Please note 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 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 by using the 'Refactor' menu. Deleting a file can simply be done by right-clicking on it and selecting 'Delete'. Please note, however, that *changing a file name or location that is part of a MAS project does not modify the contents of the file itself.* When changing a module file name, or moving or deleting it, for example, you need to manually change the references to that file. All of these basic file operations can also be executed on the file system itself, after which you may need to refresh the related projects in Eclipse (right-click the project and select 'Refresh').

3.2.2 Automatic Completion

Automatic code completion is fully supported.

Using auto-completion:

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

Chapter 4

Running a Multi-Agent System

This chapter explains how to run a MAS project. 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.

4.1 Launching a Multi-Agent System

Launching a MAS means:

1. Launching an environment referenced in the MAS file, if any, and
2. Launching the agents referenced in the MAS file (when applicable).

How to launch a MAS:

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

'Debug As' will automatically switch to the GOAL Debug Perspective, and place a shortcut to this perspective in the top-right corner (if it was not there yet). If you use the GOAL Debug Perspective for the first time, you will be asked if you want to open it. You should select 'Remember my decision' and 'Yes' in this case in order to debug properly. In the debug perspective, the **Debug panel** is shown at the top . This panel shows the processes that have been created by launching the MAS. In the Blocks World example, an environment process is started labelled *Blocks World*, as well as two agent processes named *stackbuilder* and *tableagent*. All processes that are created are *paused* initially, which is indicated by the label just after the process names.

Before launching a MAS, make sure to check that the MAS project is clean. That is, make sure no 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.

Note 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 (at the bottom of the screen). At the top-right corner of the console, actions like terminating the run are available. You can search in any console as well ('Right click → Find/Replace' or 'Left click → CTRL+F'). This is useful for quick, light-weight testing of desired functionalities. Note that with this method, the environment and all agents are automatically started, and thus the following sections do not apply.

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:

1. Select the top node in the Debug panel ('GOAL Debugging Engine').
2. Press the 'Run' button in the tool bar (or use F8).

All processes, including the environment and the agents, must be in *running mode* to be able to run the *complete* multi-agent system.

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).

4.3 Terminating a MAS

How to terminate a MAS:

1. Select the top node in the Debug panel.
2. Press the 'Terminate' button in the tool bar.
   Alternatively, press the red stop button of the console tab (at the bottom). Yet another way is right-clicking the top node in the Debug panel, and selecting one of the termination options there.

Terminating a MAS terminates the environment and all agents. An agent will also be *automatically killed* if the entity that it is controlling in an environment has been terminated, and an agent can also have terminated itself. If a terminated agent is put back in running mode, the agent is restarted (as if it were just launched).

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' (F8), 'Step' (into/over/out, F5/6/7), 'Pause', and 'Terminate' buttons in the tool bar. You can *control all processes* that are part of the MAS at once by selecting the top process node and using the relevant buttons. This is particularly useful when starting, stepping or killing all agents and/or the environment. Holding `shift` or `ctrl` whilst clicking on the nodes will allow you to hand-pick multiple processes. Note that environments can be in running, paused or killed mode, but not in stepping mode.

More information about the stepping will be provided in the next chapter.

Chapter 5

Debugging a Multi-Agent System

This chapter explains how debug a MAS project. Debugging is the process of detecting, locating, and correcting faults in a program. Compared to other programming paradigms, agent-oriented programming introduces several specific challenges. Cognitive agent programs (e.g. GOAL agents) repeatedly execute a **decision cycle** which not only *controls the choice of action* of an agent (e.g., which rules are applied) but also specifies how and when particular *updates of an agent's state* are performed (e.g., how and when percepts and messages are processed). In this introduction, we briefly discuss the relevant background for debugging cogntive agents, based on [3]. Next, in the subsections, all practical details are presented (i.e., how to use the debugger for stepping through a program, including how to record and navigate the history of a program's execution).

An agent's decision cycle provides a set of points that the execution can be suspended at, i.e. **breakpoints**. These points do not necessarily have a corresponding code location. For example, receiving a message from another agent is an important state change that is not present in an agent's source, i.e., there is no code in the agent program that makes it check for new messages. Thus, two types of breakpoints can be defined: *code-based* breakpoints and (decision) *cycle-based* breakpoints. Code-based breakpoints have a clear location in an agent program. Cycle-based breakpoints, in contrast, do not always need to have a corresponding code location. Together, these are referred to as the set of *pre-defined* breakpoints. When single-stepping through a program, these points are traversed. A user is also be able to mark specific locations in an agent's source at which execution will always be suspended, even when not explicitly stepping. To facilitate this, the debugger identifies such a marker (e.g., a line number) with the nearest code-based breakpoint. These markers are referred to as *user-defined* breakpoints. A user is also be able to suspend execution upon specific decision cycle events, especially when those do not have a corresponding location in the source. Such an indication is referred to as a *user-selectable* breakpoint.

A user is able to **control the granularity** of the debugging process. In other words, a user can navigate the code in such a way that a specific fault can be investigated conveniently. For example, skipping parts of an agent program that are (seemingly) unrelated in order to examine (seemingly) related parts in more detail. This is supported by three different step actions: *step into*, *step over*, and *step out*. At any breakpoint, a detailed **inspection of an agent's cognitive state** is facilitated. In addition, support for **evaluable cognitive state expressions** is provided, allowing a user to pose queries about specific rule parts to identify which part fails. Modifying the agent's cognitive state is supported as well.

5.1 Stepping an Agent

A paused agent in the can be single-stepped. This means that the agent will execute until the next pre-defined breakpoint is reached. For each such breakpoint, we need to determine the result of a stepping action, i.e., the flow of stepping.

The result of a stepping action (see Figure 5.1 as well):

- **Into (F5):** traverse downward from the current node in the diagram until we hit the next breakpoint. In other words, follow the edges going down in the tree's levels until an indicated node is reached. If the current node is a leaf (i.e., we cannot go down any further), perform an over-step.
- **Over (F6):** traverse to the next node (i.e., to the right) on the current level until we hit the next breakpoint. If there are none, perform an out-step.
- **Out (F7):** traverse upward from the current node until we hit the next breakpoint, whilst remaining in the current context. In other words, the edges going back up in the tree's levels should be traced until any applicable node, and then from there back down again until any indicated node is reached (like an into-step). Here, applicable refers to a 'one-to-many' edge of which not all cases have been processed yet.

In Figure 5.1, this flow for the step into and step over actions on each breakpoint has been illustrated. For readability, the step out action has been left out. Note that the broken edge indicates a link to the `event module`. After the event module has been processed, depending on the rule evaluation order, either the first rule in the module or the rule after the performed action will be evaluated. In addition, a module's exit conditions might have been fulfilled at this point as well, which means that the flow may return to the `action combo` in which the call to the exited module was made. Note that the stepping flow is heavily influenced by the rule evaluation order and/or exit conditions in general.

5.1.1 User-defined breakpoints

User-defined breakpoints are line-based. They can be set in Eclipse at any time (i.e. also whilst running a MAS) by *double-clicking on the line number you want to break at*, on which a red marker will appear. When double-clicking on this red marker again, a yellow marker appears. Finally, when double-clicking on this yellow marker, the marker and thus the breakpoint will be removed entirely.

The two different types of user-defined breakpoints:

- **Regular breakpoint:** indicated by a red marker. A user-defined breakpoint will be set on the first module entry, rule condition, or pre-condition that can be found after the indicated line. When this code-based breakpoint is reached during execution, the agent will always be paused (even when not explicitly stepping).
- **Conditional breakpoint:** indicated by a yellow marker. A user-defined breakpoint will be set on the first action(combo) that can be found after the indicated line. When this code-based breakpoint is reached, the agent will always be paused (even when not explicitly stepping). This means that if the condition of the action (i.e. of the rule) does not hold, the breakpoint will not be reached.

5.1.2 User-selectable breakpoints

GOAL has a single user-selectable breakpoint: the achievement of a goal. When stepping an agent, by default, the execution will be paused when a user-selectable breakpoint occurs. This behaviour can be modified (i.e., turned on or off) in the preferences (see Section 2.5) The stepping flow after a user-selectable breakpoint is dictated by the existing (surrounding) node. For example,

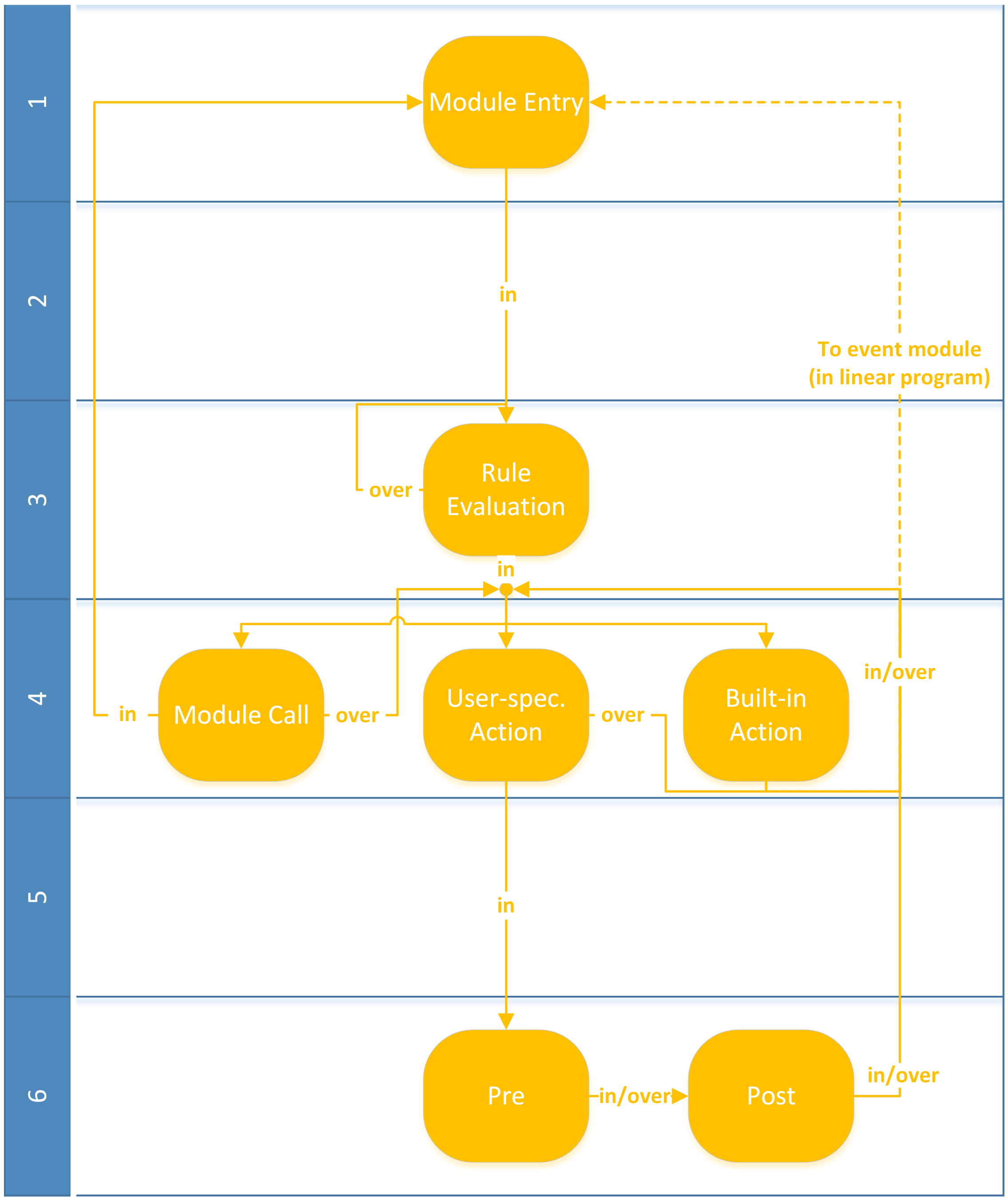


Figure 5.1: The flow of step into and step over actions for a GOAL agent. A 'User-spec. Action' indicates an action that has been defined in an action specification (`act2g` file), whilst a 'Built-in Action' is something like `insert` or `adopt`.

achieving a goal is only possible after either executing an action or applying a post-condition, so the stepping actions from the relevant node will be used when stepping away from a goal-achieved breakpoint.

5.2 State Inspection

Each time the execution is suspended (i.e., a breakpoint is reached), the code that is about to be executed is highlighted, and any relevant evaluations (i.e., the possible values of variables referenced in a rule) of this highlighted code are displayed (on the right side). Additional messages are displayed in the evaluation window as well, for example when an agent has been terminated. Note that an empty substitution (`[]`) implies that the condition is true as well.

In addition, when paused, *the cognitive state of an agent* is displayed at the top-right, split in four tabs: beliefs, goals, messages, and percepts. Searching is possible in all of these views ('Right click → Find' or 'Left click → CTRL+F'). Note that both the displayed cognitive state and the interactive console (see the next paragraph) use the agent that is currently selected in the overview (at the top-left).

5.2.1 Interactive Console

A single 'interactive console' is provided in which both cognitive state queries (e.g. `bel` or `goal`) and actions (e.g. `delete` or `drop`) can be performed in order to respectively *inspect* or *modify* the displayed cognitive state. All solutions for a query are shown in the interactive console as a list of substitutions for variables. For actions, one needs to make sure that they are closed, i.e., have no free (unbound) variables. It is possible to execute a user-defined (environment) action as well when the environment is running. Consult the GOAL Programming Guide [1] for a more detailed explanation of cognitive state queries and actions.

5.2.2 Watch Expressions

It is also possible to continuously evaluate and thus inspect one or more cognitive state queries whilst a system is running by using so-called 'Watch Expressions'. The evaluation of each expression will be shown for all agents, and is updated every second (i.e., even when they are running).

How to continuously evaluate cognitive state queries:

1. Open the Expressions panel
2. Click on 'Add new expression' and enter a cognitive state query.
3. Press Enter.

5.3 Logging

In debug mode, the bottom area contains various tabs for inspecting agents and the actions they perform.Besides the main console tab, an action history 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 dedicated console is added for that agent for inspecting various aspects of the agent program during runtime.

5.3.1 Main Console

The main 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.

5.3.2 Action History

The action history tab shows the actions that have been performed by all 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.

5.3.3 Agent Consoles

Upon launching an agent, a dedicated tab is added for that agent. This tab typically contains more detailed information about your agent. The exact contents of the tab can be customized through the preferences (see Section 2.5). 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 console of an agent. A more detailed explanation of these settings can be found in the next section.

Note that in debug mode, as an optimization and to keep the logs more readable, empty cycles of an agent are not printed. This means that an agent might be running, but not producing any logging output. Pausing the agent will make it print a message about which cycle it is currently in.

5.3.4 Customizing the Logging

The logging can be customized in the preferences (see Section 2.5). A timestamp with millisecond accuracy can be added to each printed log. The action history and dedicated logging tabs for agents can also be turned on or off here. It is also possible to write all logs to separate files. These files will be stored in a directory within the currently executed project. Logs for each agent, the action log, warnings, results from GOAL log actions etcera are all written to separate files.

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. System 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 might be written, degrading performance significantly. 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.

You can also turn stack traces and/or detailed Java messages on or off (i.e. for exception handling). You can also set the maximum number of times the same message is printed, to avoid flooding the console(s).

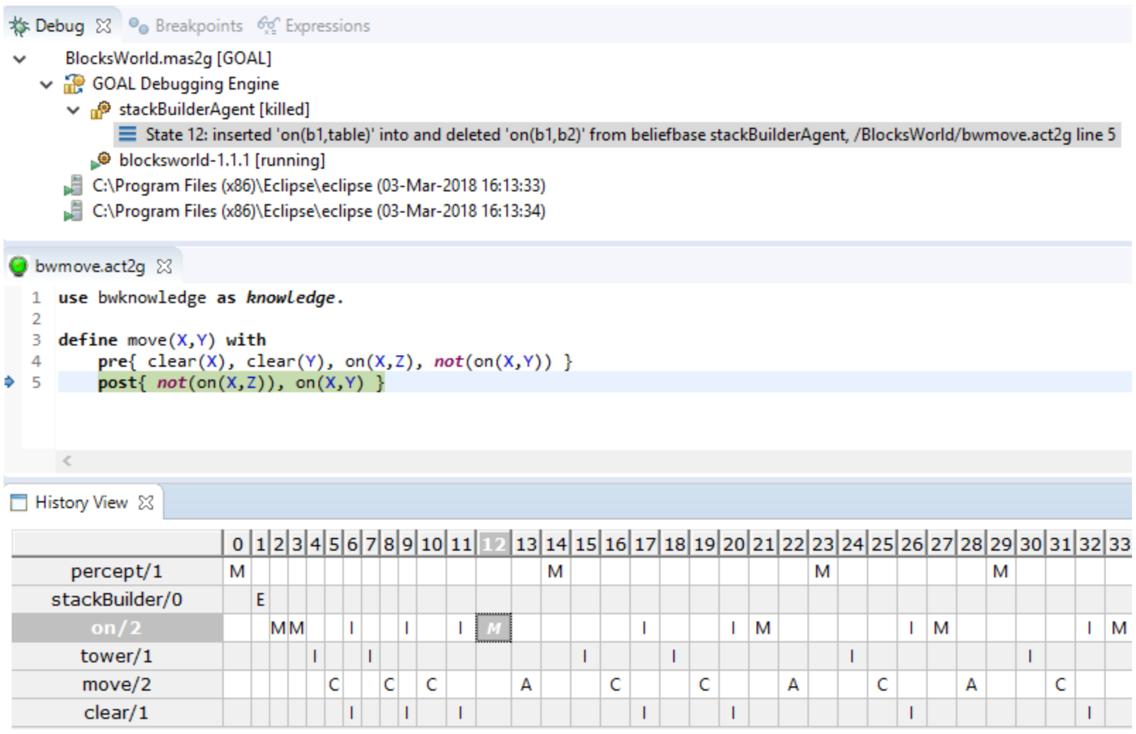
5.4 Back-in-time Debugging

Running the same agent system again more often than not results in a different program run or trace, which complicates the iterative process of debugging. In order to deal with this issue, GOAL contains a tracing mechanism that supports *omniscient debugging for cognitive agents* [4]. Omniscient (or back-in-time) debugging allows a developer to move backwards in time through a program's execution. In the debugger of the GOAL-Eclipse plug-in, a so-called *space-time view of the execution history* is provided. Using this visualisation, a developer can also *single-step through a program's execution history.*

Because of its performance impact (of about 10%), the recording of a trace is not enabled by default, and can be toggled on or off in Eclipse through the 'Enable trace recording (history)'[1] (see Section 2.5). When enabled, the execution of any run of a MAS will be stored to a file (one per agent) in the indicated logging directory (again see Section 2.5). Note that these files are not automatically removed. The trace for any agent (i.e., which is selected in the 'Debug' tab on the top) can be inspected and navigated through in the 'History View' of the debugger.

For the cognitive agents in GOAL, the elements in the space-time view (i.e., that are traced) are the agent's events, beliefs, goals, actions, and modules. We use the corresponding *signatures* as the rows in the space dimension. For example, the signature `move/2` in Fig. 5.2 represents a `move` action with two parameter. Each point in a trace represents a step (column) in the time dimension. Multiple space elements (signatures) can be used in a single step, e.g., evaluating a query may require accessing several beliefs and goals. The cells in the space-time view contain information about how an element was used at a particular step, which differs per type of element (e.g., a belief can be modified or inspected, an action can be called and performed, a module can be entered or exited). Empty cells indicate the element was not used. A single letter is used in each cell: **E**ntered a module, **L**eft a module, **M**odified of a belief or goal, **I**nspected of a belief or goal, **C**alled an action, or **A**ction executed. Note that calling an action is different from executing an action, as the action's precondition might fail after calling it (after which the action is not executed). Moreover, note that the 'percept/1' note indicates when the agent's percept were updated (and thus when a new decision cycle was started).

By *double-clicking on any cell in the table*, the debugger will reverse the agent to the state matching the column, allowing a developer to use all debugging tools (e.g., inspecting an agent's beliefs and goals) in that specific historic state. The 'Debug' tab also shows information about the state that was navigated to. Note that a running agent will first be killed before navigating its history, and cannot be restarted afterwards (i.e., it goes into 'post-mortem inspection' mode).



BlocksWorld.mas2g [GOAL]
GOAL Debugging Engine
stackBuilderAgent [killed]
State 12: inserted 'on(b1,table)' into and deleted 'on(b1,b2)' from beliefbase stackBuilderAgent, /BlocksWorld/bwmove.act2g line 5
blocksworld-1.1.1 [running]
C:\Program Files (x86)\Eclipse\eclipse (03-Mar-2018 16:13:33)
C:\Program Files (x86)\Eclipse\eclipse (03-Mar-2018 16:13:34)

```
use bwknowledge as knowledge.

define move(X,Y) with
    pre{ clear(X), clear(Y), on(X,Z), not(on(X,Y)) }
    post{ not(on(X,Z)), on(X,Y) }
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
percept/1	M														M									M						M				
stackBuilder/0		E																																
on/2			M	M			I			I		I	M					I			I	M					I	M					I	M
tower/1					I			I								I			I						I						I			
move/2						C			C		C			A			C			C			A			C			A			C		
clear/1							I			I		I						I			I						I						I	

Figure 5.2: An example of a space-time view of the execution of an agent (in State 12).

[1]Note that for practical use this setting is incompatible with the 'Remove agents completely when they are killed' setting, as one will be unable to inspect an agent's history when that agent is removed upon termination.

5.5 Prolog Level Debugging

GOAL does not directly 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 export the contents of a cognitive state to a separate file that can be imported into Prolog, and then use the Prolog tracer to analyze the code.

5.5.1 Prolog Exceptions

In rare cases, when an exception occurs in the Prolog engine (SWI Prolog), error messages are a bit hard to decipher. This section explains how to read such messages. A 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.

5.6 Runtime Settings

Some generic behaviour can be customized in the preferences as well (see Section 2.5). A few options will be discussed here.

Selecting '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 enabled by default.

To remove agents automatically after they have been killed, select the 'Remove agents when they are killed' option. This option is useful in environments where large numbers of agents appear and disappear while GOAL is running.

Chapter 6

Testing a Multi-Agent System

This chapter explains how to run test and how to interpret their results. For creating the tests themselves, we refer to the GOAL Programming Guide [1].

6.1 Adding a Test File to a MAS Project

Test files (extension `.test2g`) can be created in the same fashion as other types of files as mentioned in Chapter 3.

How to create a test file:

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

6.2 Executing Tests

Test files (`.test2g`) can be used in exactly the same manner as MAS (`.mas2g`) files. One could see the test as an addition to the MAS that is referenced in the test file; all running and debugging functionality is the same. Thus, Chapter 4 fully applies here, and should be consulted if needed.

6.3 Test Results

When a test condition fails in debug mode, the debugger wilt halt on the exact piece of code that caused the test condition to fail, allowing inspection of the agent's cognitive state at that time. In run mode (i.e., when not using the debugger), a report will be printed to the console when at the end of the test.

Chapter 7

Profiling

In real-time systems, optimizing the agent performance is important. GOAL provides a profiler to analyze where agents spend their time. The profiler can be configured in the preferences (see Section 2.5).

To enable profiling, check the "Enable profiling" checkbox in the preferences. The profile reports for each agent are shown in the console when the agent is killed. If the agent was killed forcefully (using the kill button), part of the agent code was aborted and therefore will not be fully profiled. If you prefer to have the profile results in a file, enable the "Save profiling data to file(s)" checkbox. In this case a `.csv` file will be written to the logging directory (a "GOAL" directory inside the user's home directory by default). You can open this file with an editor like Excel for a better layout.

If you want to have the profile node IDs logged, check the "Include profile node IDs" checkbox. This is useful when there are for example multiple uses of the same module. The specific details of the profile log can be adjusted with the remaining checkboxes.

The profiler's output is based on a list of 'profile nodes'. Each node accumulates profile information about a certain object in the agent system. The profile report provides details about each profile node:

1. total time(s): the total accumulated time in the given profile node
2. number of calls: the total number of calls to the given profile node
3. source: the description of the profile node. For example 'total run time' profiles the run time from start to end of an agent. 'KR Insert' profiles all calls being made to insert actions. 'IfThenRule' refers to a specific if-then rule in the agent program. References to source line code is only available for specific program rules, not for more general info.
4. source info: the source code and line number related to this profile node
5. this: the ID of the profile node
6. parent: the ID of the parent of this profile node. The parent node is the node that had the current node as part of its process.

For example, profiling the wumpus gives something like this (assuming you enabled all options):

total time(s)	nr. calls	source	source info	this	parent
0.447326	1	total run time	–	@30bcc113	–
0.415127	164	completed rounds	–	@4a4a14c7	–
0.052247	489	Mental State Condition	–	@7566dc55	–
0.031534	489	KR Query	–	@221d68b4	–
0.003125	125	KR Delete events	–	@64df27ec	@6ca530b3
0.002479	128	KR Insert events	–	@48765de9	–
0.109829	163	Module:events	"line 3, position 7 in ....ts/events.mod2g"	@10716124	–
0.096267	163	IfThenRule:if percept(scream) then delete(wumpusIsAlive)	"line 4 , position 4 in .../events.mod2g"	@d8cd481	@10716124
0.093725	163	MentalStateCondition: percept(scream)	"line 4 , position 4 in ...events.mod2g"	@48694d2b	@d8cd481
0.024713	163	"Mental State Condition line 4, position 4 in ...events.mod2"	"line 4, position 4 in ...ents/events.mod2g"	@320ac8da	@48694d2b
0.00256	58	"KR Query line 4, position 1 in ...s/wumpus.act2g"	"line 4, position 1 in ...ts/wumpus.act2g"	@42b0125b	@40a81be5

If a MAS launches a type of agent multiple times, then a `All_XX profile.csv` file will also be written (where XX is the name of the agent definition in the MAS). This file contains accumulated data from all agents of a type, so both the total time and number of calls have been added up for all agents of type XX.

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.

For problems specific with the **Eclipse interface**, these two steps can be taken:

- Reset the GOAL and/or GOAL Debug perspective(s) by selecting 'Window → Reset Perspective' when the perspective is opened.
- Restore the default GOAL preferences through 'Window → Preferences → GOAL → Logging → Restore Defaults → Apply', and do the same for the 'Runtime' category.

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*. Please also include *Java details and stack trace information* if available (see Section 5.3.4 for adjusting settings to obtain this information).
4. If possible, a *screen shot* of the situation you ran into.

Bibliography

[1] Koen V. Hindriks. Programming Cognitive Agents in Goal. https://goalapl.dev/GOALProgrammingGuide.pdf, 2021.

[2] Vincent J. Koeman and Koen V. Hindriks. A fully integrated development environment for agent-oriented programming. In Yves Demazeau, Keith S. Decker, Javier Bajo Pérez, and Fernando de la Prieta, editors, *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, volume 9086 of *LNCS*, pages 288–291. Springer International Publishing, 2015.

[3] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Designing a source-level debugger for cognitive agent programs. *Autonomous Agents and Multi-Agent Systems*, 2016.

[4] Catholijn M. Jonker Vincent J. Koeman, Koen V. Hindriks. Omniscient debugging for cognitive agent programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 265–272, 2017.