The MCAPL Framework Manual 2024: Covering the Agent Infrastructure Layer (AIL) and Agent Java Pathfinder (AJPF)

Louise A. Dennis

July 11, 2024

Contents

Chapter 1

Introduction

The MCAPL framework is a suite of tools for building interpreters for agent programming languages and model checking programs executing in those interpreters. It consists of the AIL toolkit for building interpreters for rational agent programming languages (BDI languages) as introduced by [Rao and Georgeff, 1992] and the AJPF model checker [Dennis et al., 2012]. AJPF extends the JavaPathfinder (JPF) model checker [Visser et al., 2003] to prove LTL properties of BDI agents. This distribution also contains a number of programming languages implemented in the AIL. Chief among these are Gwendolen [Dennis, 2017], the EASS variant of Gwendolen that can be used to program hybrid autonomous systems and GOAL [Hindriks et al., 2001]. These languages are described in this manual. It also contains the systems outlined in [Dennis et al., 2015a] (`ethical_gwen`), [Dennis et al., 2015c] (`actiononly` and `ethical_governor`), [Ferrando et al., 2018] (`monitor`), [Bremner et al.,] (`pbdi`) and a reimplemention of the HERA system [Lindner et al., 2017] (`hera`) together with a BDI-style wrapper for it (`juno`). These systems are not documented in this manual.

The high-level concepts behind the MCAPL framework, including discussion of many examples and extensions can be found in [Dennis and Fisher, 2023].

This manual consists primarily of basic installation instructions and then a set of tutorials covering various aspects of using the system and some of the languages shipped with it. These tutorials can also be found in the tutorials sub-directory. Chapter 2 provides installation instructions and Chapter 3 provides simple instructions for running and model-checking a program in the framework. Chapter 4 describes the use of the AIL with particular reference on its use in creating environments for multi-agent systems – where users of particular languages will need to engage with the AIL. Chapter 5 provides a tutorial based description of the use of AJPF for model checking. Chapter 6 is a tutorial introduction to the Gwendolen programming language and chapter 8 is a tutorial introduction to its EASS variant that can be used for programming hybrid autonomous systems . Chapter 7 provides an operational semantics for Gwendolen which may, among other things, be useful for people considering

implementating their own language in the AIL toolkit. Chapter 9 consists of a discussion of the differences between using the AIL implementation of the GOAL programming language and the version described in [Hindriks, 2014] and a discussion on the use of AJPF to model check GOAL programs. [Hindriks, 2014] is included in the distribution so that full instructions on programming agents in GOAL are available.

How to read this manual If you are installing the MCAPL framework then we recommend you start by reading chapters 2 and 3 and follow the instructions to install the system and run some basic programs to check it is functioning correctly. The other chapters are mostly stand alone and will refer you to the relevant parts of this manual where they are not. The exception is chapter 8 which assumes familiarity with programming in Gwendolen (chapter 6). If your interest is in learning BDI programming in Gwendolen or its EASS variant then we recommend you start with chapter 6 and proceed to chapter 8 if desired. Chapter 4 is also useful for learning to use this system since it covers the creation of environments for multi-agent systems. If your interest is primarily in using the AJPF model checker then we recommend you start by reading chapter 5. If you want to model check GOAL programs then you should read chapter 9.

Note This release comes packaged with Jar files for NASA's Java Pathfinder (JPF) tool and a number of other libraries. The relevant jars, zip files of source code and the NASA license can be found in `lib/3rdparty`. The release is configured to run using the version of JPF packaged with it, but this means that some adaptation, particularly of configuration files, may be required to run it on systems where JPF is already installed.

The development of the MCAPL Framework would have been impossible without the financial support of the EPSRC via several grants: Model-Checking Agent Programming Languages (EP/D052548), Engineering Autonomous Space Software (EP/F037201/1), Reconfigurable Autonomy (EP/J011770), Verifiable Autonomy (EP/L024845/1), Robotics and AI for Nuclear (EP/R026084/1), Future AI and Robotics for Space (EP/R026092/1), and Trustworthy Autonomous Systems Verifiability Node (EP/V026801/1). Thanks are also owed to Jomi Hübner, Koen Hindriks, Matt Webster, Angelo Ferrando, Vincent Koeman, Fatma Faruq, and Mengwei Xu.

Chapter 2

Installation Instructions

2.1 Requirements

The MCAPL software requires Java 8.

IMPORTANT: It does have to be Java 8. The version of Java PathFinder currently packaged with the system does not work with later versions of Java.

2.2 Installation

2.2.1 Installation of MCAPL 2024

- Download the 2024 release from Github `https://github.com/mcapl/mcapl/releases/`.
- Create a file `.jpf/site.properties` in your home directory with the path to mcapl2024 assigned to the value `mcapl` (e.g. `mcapl = ${user.home}/mcapl`).

  If you have put the `mcapl` download in your home directory, you should be able to do this at the UNIX command line by running the following commands in your home directory.

  ```
  mkdir -p ~/.jpf && touch ~/.jpf/site.properties
  ```

  followed by:

  ```
  echo "mcapl = ${user.home}/mcapl" >> ~/.jpf/site.properties
  ```

- Within the `mcapl` directory you should find `build.xml` to which you can apply ant to build ajpf. (e.g., at the command line, `ant compile` to just compile the files and `ant build` to build and run regression tests (takes just under fifteen minutes on a 3GHz Macbook with 16 GB memory))
- Make sure you have the bin sub-directory of `mcapl` on your java class path.

- We recommend you set the `AJPF_HOME` environment variable to give the path to your MCAPL directory.

2.2.2 Installation of Development Branch

You will need an installation of the Git version control system. It is worth taking some time before you start to understand the basics of Git, particularly checking out versions, switching between branches, commiting changes and managing local and remote repositories. There are several introductions to Git and Github on the web.

- Clone the git repository from github.

  ```
  git clone https://github.com/mcapl/mcapl.git
  ```

- Create a file `.jpf/site.properties` in your home directory with the path to mcapl assigned to the value `mcapl` (e.g. `mcapl = ${user.home}/mcapl`).

  If you have put the `mcapl` download in your home directory, you should be able to do this at the Unix command line by running the following commands in your home directory:

  ```
  mkdir -p ~/.jpf && touch ~/.jpf/site.properties
  ```

  followed by:

  ```
  echo "mcapl = ${user.home}/mcapl" >> ~/.jpf/site.properties
  ```

- Within the mcapl directory you should find `build.xml` to which you can apply ant to build ajpf. (e.g., at the command line, `ant compile` to just compile the files and ant build to build and run regression tests (takes just under fifteen minutes on a 3GHz Macbook with 16 GB memory))

- Make sure you have the bin sub-directory of mcapl on your java class path.

- We recommend you set the `AJPF_HOME` environment variable to give the path to your mcapl directory.

2.3 IntelliJ Installation

We also supply an IntelliJ Module file so you should be able to import mcapl2024 (or mcapl if using the development version) into IntelliJ. You will need a Java 8 SDK and JRE installed and will probably also want to install the ant plugin in order to run the tests easily. You may need to configure the Project Settings manually to use your Java 8 SDK.

- If you have the ant plugin installed, you should then be able to build ajpf by clicking on build in the ant window (ant symbol – often on the right of the IDE). This will build the project and run the "quick" test suite. You can see the progress of this in the Messages window (usually at the bottom on the left of the IDE).
- Create a file `.jpf/site.properties` with the path to mcapl2024 (or mcapl if using the development version) assigned to the value `mcapl` (e.g. `mcapl = ${user.home}/IdeaProjects/mcapl2024`).
- We recommend you set the `AJPF_HOME` environment variable to give the path to your ajpf directory.

2.4 Eclipse Installation

We also supply an eclipse `.project` file so you should be able to import mcapl2024 (or mcapl if using the development version) into eclipse. You need to be using a version of Eclipse for Java Development. We have tested on the Eclipse IDE for Java Developers. You will need to configure Eclipse to use Java 1.8 as both its compiler and runtime environment.

- In theory, you should then be able to build ajpf by clicking on the build.xml file, however recent versions of Eclipse force the build process to use Java 11 and don't send clear error messages when they do this. If the build process isn't working, you may still nevetheless be able to use the MCAPL framework by using Eclipse's built-in build process bypassing the `ant` file. It is possible to force Eclipse to use a version of `ant` that is compatible with Java 8, but this is non-trivial.
- Create a file `.jpf/site.properties` with the path to mcapl2024 (or mcapl if using the development version) assigned to the value `mcapl` (e.g. `mcapl = ${user.home}/Eclipse/mcapl2024`).
- We recommend you set the `AJPF_HOME` environment variable to give the path to your ajpf directory. If necessary you can do this within the Environment tab for individual Run Configurations.

2.5 Testing the Installation

Chapter 3 describes how to manually run and model-check a simple example program.

There are also number of JUnit tests in the subdirectory `/src/tests` and examples in `/src/examples`.. Using ant, the build target in `build.xml` runs a set of tests.

Part I

Agent JavaPathfinder and the Agent Infrastructure Layer

Chapter 3

Running and Model Checking an Agent Program

All of the languages implemented using the Agent Infrastructure Layer (AIL) come provided with a parser which allows files written in that language to be read in and executed. Examples of programs can be found in the `src/examples` directory and the tutorials for some of the languages can be found in Part 2 of this manual.

However these languages only describe the agents and these agents must execute within an multi-agent system consisting of an environment and one or more agents. Therefore, any specific example needs to first construct such a multi-agent system. The languages implemented in the AIL all come with classes for parsing input files to sets of agents and many use the `DefaultEnvironment` class that come with the AIL – there is more information on creating multi-agent systems and environments in Chapter 4. Configuration files can be used to describe the classes necessary for a given multi-agent system and the class `ail.mas.AIL` will build and run a multi-agent system from a configuration file. This can also be invoked using the `run-AIL` IntelliJ or Eclipse Run Configurations.

Model Checking an agent system uses standard JPF configuration files which can be supplied to the `gov.nasa.tool.RunJPF` command at the command line, or to the `run-JPF (MCAPL)` Run Configuration in IntelliJ or Eclipse. These configuration files should incorporate the line `@using = mcapl` at the top which should ensure that the correct listeners, etc., for AJPF are configured. This assumes that the MCAPL project has been added to the JPF `site.properties` as described in the Installation Instructions (Chapter 2). A general support class for model checking agent systems configured using an AIL configuration file has been provided. This is `ail.util.AJPF_w_AIL`.

3.1 Example: Executing a Multi-Agent-System (UNIX Based Systems)

Important: Make sure that `mcapl2024/bin` is in your CLASSPATH. You also need to make sure that the following jar files appear in your CLASSPATH

```
lib/3rdparty/antlr-4.7-complete.jar
lib/3rdparty/commons-io-2.4.jar
lib/3rdparty/eis-0.5.0.jar
lib/3rdparty/ev3classes.jar
lib/3rdparty/ev3tools.jar
lib/3rdparty/guava-31.1-jre.jar
lib/3rdparty/java-prolog-parser.jar
lib/3rdparty/jpf.jar
lib/3rdparty/jpf-annotations.jar
lib/3rdparty/jpf-classes.jar
lib/3rdparty/jpl.jar
lib/3rdparty/json-simple-1.1.1.jar
lib/3rdparty/junit-4.10.jar
lib/3rdparty/system-rules-1.16.0.jar
```

We also recommend setting the environment variable AJPF_HOME to be the path to `mcapl2024` (NB. throughout this chapter, if you are using the development version `mcapl2024` should be `mcapl`).

You can run the simple agent `pickupagent.gwen` whose code you will find in `/src/examples/gwendolen/simple/PickUpAgent` by calling

```
> java ail.mas.AIL {$path_to}/src/examples/gwendolen/simple/PickUpAgent/PickUpAgent.ail
```

where `$path-to` is the path to ajpf in your system.

You should see output similar to:

```
MCAPL Framework 2024
done
```

`PickUpAgent.ail` is a Configuration File which describes how AIL should build the relevant multi-agent system.

NB If you do not have AJPF_HOME set then you will need to run the example from the `mcapl2024` directory.

3.1.1 In IntelliJ

If you have successfully imported the project into IntelliJ then you should have two run configurations `run-AIL` and `run-JPF (MCAPL)`.

Select the configuration file you want to run, and then select `run-AIL` from the Run Configuration drop-down.

3.1.2 In Eclipse

If you have successfully imported the project into Eclipse then you should have two run configurations `run-AIL` and `run-JPF (MCAPL)`. If you do not these can be found in the `eclipse` sub-directory.

Select the configuration file you want to run, and then select `run-AIL` from Eclipse's Run Configuration menu.

3.2 Example: Model Checking a Multi-Agent System (UNIX Systems)

To verify a multi-agent system, you will need to run JPF which uses a class contained in `lib/3rdparty/jpf.jar`. Make sure this is on your class path. Call *in the mcapl2024* directory:

```
> java gov.nasa.jpf.tool.RunJPF ${path-to}/src/examples/gwendolen/simple/PickUpAgent/PickUpAgent.jpf
```

where `$path-to` is the path to mcapl2024 in your system.

You should see output similar to:

```
------------------------------------------ search started
      [skipping static init instructions]
JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights


========================================================== system under test
ail.util.AJPF_w_AIL.main("/src/examples/gwendolen/simple/PickUpAgent/PickUpAgent.ail

========================================================== search started: 30/03/23 11:5
                # choice: gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT" ,1/
MCAPL Framework Version 2023
ANTLR Tool version 4.4 used for code generation does not match the current runtime v
# garbage collection
[INFO] Adding 0 to []
------------------------------------ [1] forward: 0 new
                # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="NewAgentProgr
                # garbage collection
[INFO] Adding 1 to [0]
------------------------------------ [2] forward: 1 new
                # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="NewAgentProgr
                # garbage collection
[INFO] Adding 2 to [0, 1]
------------------------------------ [3] forward: 2 new
                # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="NewAgentProgr
                # garbage collection
[INFO] Adding 3 to [0, 1, 2, 3]
[INFO] Always True from Now On
```

```
------------------------------------------ [4] forward: 3 visited
------------------------------------------ [3] backtrack: 2
------------------------------------------ [3] done: 2
------------------------------------------ [2] backtrack: 1
------------------------------------------ [2] done: 1
------------------------------------------ [1] backtrack: 0
------------------------------------------ [1] done: 0
------------------------------------------ [0] backtrack: -1
------------------------------------------ [0] done: -1
------------------------------------------ search finished

========================================================== results
no errors detected

========================================================== statistics
elapsed time:       00:00:01
states:             new=3,visited=1,backtracked=4,end=0
search:             maxDepth=3,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,resch
heap:               new=6113,released=3792,maxLive=2265,gcCycles=4
instructions:       365022
max memory:         491MB
loaded code:        classes=301,methods=4831

========================================================== search finished: 30/
```

3.2.1 In IntelliJ

In IntelliJ you should be able to select `run-JPF (MCAPL)` from the Run Configurations menu while you have `src/examples/gwendolen/simple/PickUpAgent/PickUpAgent.jpf` selected. This should generate similar output to the above.

3.2.2 In Eclipse

In eclipse you should be able to select `run-JPF (MCAPL)` from the Run menu while you have `src/examples/gwendolen/simple/PickUpAgent/PickUpAgent.jpf` selected. This should generate similar output to the above.

3.3 Executing and Model Checking Multi-Agent Systems on Windows Systems

AJPF and the AIL have not been extensively tested on Windows systems. In particular all the examples assume UNIX conventions for path names. *In theory* however, it should be possible to adapt these to Windows systems simply by converting paths to use Windows style paths.

Chapter 4

Creating Multi-Agent Systems with the Agent Infrastructure Layer

This chapter contains tutorials on the use of the Agent Infrastructure Layer (AIL). This primarily concerns the construction of multi-agent systems out of a set of agents programmed in specific languages and the environments in which those systems run.

4.1 Tutorial 1 – Configuration Files

This is the first in a series of tutorials on the use of the Agent Infrastructure Layer (AIL). This tutorial covers the basics of configuring a multi-agent system in the AIL which can then be used to run the system and/or for model checking. This duplicates material that appears in section 6.1 and in section 6.8 but involves a more thorough discussion of configuration options.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ail_tutorials/tutorial1`.

This tutorial assumes some familiarity with the basics of running Java programs either at the command line or in Eclipse.

4.1.1 Agent Java PathFinder (AJPF) and the Agent Infrastructure Layer (AIL)

Agent Java PathFinder (AJPF) is primarily designed to work with agent programming languages which are implemented using the *Agent Infrastructure Layer* (AIL). The first language implemented in the AIL was Gwendolen so the examples in these tutorials will use Gwendolen agents. It isn't necessary

to understand GWENDOLEN to use these tutorials but it is important to understand a little bit about the AIL. In particular it is important to understand AIL configuration files and how they are used to construct a multi-agent system for model checking.

4.1.2 An Example Configuration Files

You will find an AIL configuration file tutorial directory called `hello_world.ail`. Its contents is shown in figure 4.1.

```
mas.file = /src/examples/gwendolen/ail_tutorials/tutorial1/hello_world.gwen
mas.builder = gwendolen.GwendolenMASBuilder

env = ail.mas.DefaultEnvironment

log.warning = ail.mas.DefaultEnvironment
```

Figure 4.1: An AIL Configuration File

This is a very simple configuration consisting of four items only.

mas.file gives the path to the GWENDOLEN program to be run.

mas.builder gives a java class for building the file. In this case `gwendolen.GwendolenMASBuilder` parses a file containing one or more GWENDOLEN agents and compiles them into a multi-agent system.

env provides an environment for the agent to run in. In this case we use the default environment provided by the AIL.

log.warning sets the level of output for the class `ail.mas.DefaultEnvironment`. This is a pretty minimal level of output (warnings only). We will see in later tutorials that it is often useful to get more output than this.

You will notice that the GWENDOLEN MAS file, `hello_world.gwen` is also in the tutorial directory.

Running the Program

To run the program you need to run the JAVA program `ail.mas.AIL` and supply it with a the configuration file as an argument. You can do this either from the command line or using the IntelliJ or Eclipse `run-AIL` configuration (with `hello_world.ail` selected in the Project Files/Package Explorer window) as detailed in chapter 3

Run the program now.

4.1.3 Configuration Files

Configuration files all contain a list of items of the form `key=value`. Particularly agent programming languages, and even specific applications may have their own specialised keys that can be placed in this file. However the keys that are supported by all agent programs are as follows:

env This is the Java class that represents the environment of the multi-agent system. The value should be a java class name – e.g., `ail.mas.DefaultEnvironment`.

mas.file This is the name of a file (including the path from the MCAPL home directory) which describes all the agents needed for a multi-agent system in some agent programming language.

mas.builder This is the Java class that builds a multi-agent system in some language. For GWENDOLEN this is `gwendolen.GwendolenMASBuilder`. To find the builders for other languages consult the language documentation.

mas.agent.*N*.file This is the name of a file (including the path from the MCAPL home directory) which describes the Nth agent to be used by some multi-agent system. This allows individual agent code to be kept in separate files and allows agents to be re-used for different applications. It also allows a multi-agent system to be built using agents programmed in several different agent programming languages.

mas.agent.*N*.builder This is the Java class that is to be used to build the Nth agent in the system. In the case of GWENDOLEN individual agents are built using `gwendolen.GwendolenAgentBuilder`. To find the builders for other languages consult the language documentation.

mas.agent.*N*.name All agent files contain a default name for the agent but this can be changed by the configuration (e.g., if you want several agents which are identical except for the name – this way they can all refer to the same code file but the system will consider them to be different agents because they have different names).

log.severe, log.warning, log.info, log.fine, log.finer, log.finest These all set the logging level for Java classes in the system. `log.finest` prints out the most information and `log.severe` prints out the least. By default most classes are set to `log.warning` but sometimes, especially when debugging, you may want to specify a particular logging level for a particular class.

log.format This lets you change the format of the log output from Java's default. At the moment the only value for this is `brief`.

ajpf.transition_every_reasoning_cycle This can be `true` or `false` (by default it is true). It is used during model checking with AJPF to determine whether a new model state should be generated for every state in the agent's reasoning cycle. This means that model checking is more thorough, but at the expense of generating a lot more states.

ajpf.record This can be `true` or `false` (by default it is false). If it is set to true then the program will record its sequence of choices (all choices made by the scheduler *and* any choices made by the special `ajpf.util.choice.Choice` class). By default (unless `ajpf.replay.file` is set) these choices are stored in a file called `record.txt` in the `records` directory of the MCAPL distribution.

ajpf.replay This can be set to `true` or `false` (by default it is false). If it is set to true then the system will execute the program using a set of scheduler and other choices from a file. By default (unless `ajpf.replay.file` is set) this file is `record.txt` in the `records` directory of the MCAPL distribution.

ajpf.replay.file This allows you to set the file used by either `ajpf.record` or `ajpf.replay`.

ail.store_sent_messages This can be `true` or `false` (by default it is true). If it is false then AIL's built-in rules for message sending will not store a copy of the message that was sent. This can be useful to reduce the number of states when model checking, but obviously only if it isn't important for the agent to know about sent messages.

4.1.4 Exercises

In the tutorial directory you will find three further Gwendolen files (`simple_mas.gwen`, `lifter.gwen` and `medic.gwen`) and an environment (`SearchAndRescueMASEnv.java`).

simple_mas.gwen Is a simple multi-agent system consisting of a lifter agent and a medic agent. The lifter explores a location (5, 5). If it finds a human it will summon the medic to assist the human. If it finds some rubble it will pick up the rubble.

lifter.gwen Is a single lifting agent much like the one in `simple_mas.gwen`. It explores first location (5, 5) and then (3, 4) and will ask one of two medics, `medic` or `nurse` for help assisting any humans it finds.

medic.gwen Is a medic agent that assists humans if it gets sent a message requesting help.

SearchAndRescueMASEnv.java is an environment containing two injured humans, one at (5, 5) and one at (3, 4).

Exercise 1

Write a configuration file to run `simple_mas.gwen` with `gwendolen.ail_tutorials.tutorial1.SearchAndRescueMASEnv` as the environment. Set the log level for `ail.mas.DefaultEnvironment` to `info`.

You should see output like

```
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: lifter done move_to(5,5)
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: lifter done send(1:human(5,5), medic)
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: medic done move_to(5,5)
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: lifter done lift_rubble
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: medic done assist
Jan 29, 2015 5:17:42 PM ajpf.util.AJPFLogger info
INFO: Sleeping agent lifter
```

Although the order of the actions may vary depending on which order the agents act in.

You can find sample answers for all the exercises in this tutorial in the `answers` directory.

Exercise 2

Write a configuration file to run `lifter.gwen` and two copies of `medic.gwen` with `gwendolen.ail_tutorials.tutorial1.SearchAndRescueMASEnv` as the environment. One of the medic agents should be called `medic` and one should be called `nurse`.

4.2 Tutorial 2 – Extending the Default Environment

This is the second in a series of tutorials on the use of the Agent Infrastructure Layer (AIL). This tutorial covers creating environments for agent programs by extending the `ail.mas.DefaultEnvironment` class. Sometimes this will not be possible because of the complexity of the environments involved, or the requirements of the programming language interpreters but this is the simplest way to create an environment for an agent program to run in.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ail_tutorials/tutorial2`.

The tutorial assumes a good working knowledge of Java programming and an understanding of how unification works in Prolog programs.

4.2.1 The Default Environment and the AILEnv Interface

All environments for use with language interpreters created using the AIL must implement a java interface `ail.mas.AILEnv`. This specifies some minimal functionality agents will expect the environment to provide such as the ability to deliver a set of perceptions, deliver messages and calculate the outcomes of agent actions. It also requires certain methods be implemented for use with the AJPF property specification language.

`ail.mas.DefaultEnvironment` provides a basic level implementation of all these methods, so any environment that extends it only has to worry about those aspects particular to that environment. Typically this is just the way that actions performed by the agents are to be handled. `ail.mas.DefaultEnvironment` also provides a set of useful methods for handling changing the perceptions available to the agent that can then be used to program these action results.

4.2.2 A Survey of some of Default Environment's Methods

We note here some of the more useful methods made available by the Default Environment before we talk about implementing the outcomes of agent actions.

public void addPercept(Predicate per) This adds a percept which is perceivable by all agents in the environment. The percept has to be an object of class `ail.syntax.Predicate` (see section 4.2.3).

public void addPercept(String agName, Predicate per) As above but the percept is perceivable only by the agent called `agName`.

public boolean removePercept(Predicate per) This removes a percept which is perceivable by all agents in the environment. It returns `true` if the percept existed.

public void removePercept(String agName, Predicate per) As above but the percept is perceivable only by the agent called `agName`.

public boolean removeUnifiesPercept(Predicate per) Sometimes we don't know the exact logical formulae that we want removed only that it unifies with some term. This method allows us to remove any percept that unifies with the argument.

public void removeUnifiesPercept(String agName, Predicate per) As above but the percept is perceivable only by the agent called `agName`.

public synchronized void clearPercepts() Removes all percepts.

public void clearPercepts(String agName) Removes all percepts perceivable only by `agName`.

public void clearMessages(String agName) Removes all messages available for `agName`.

public void addMessage(String agName, Message m) This adds a message to an agent's inbox. In general this should only be used by agent's invoking `SendAction`'s but their may be circumstances when a system requires messages to be added at other times.

4.2.3 Classes for Logical Formulae

Logical Formulae in AIL are handled by a complex hierarchy of classes. Here we will concern ourselves only with the `ail.syntax.Predicate`, `ail.syntax.VarTerm`, `ail.syntax.NumberTerm`, `ail.syntax.NumberTermImpl` and `ail.syntax.Action` classes.

The Predicate Class

`ail.syntax.Predicate` is a basic work horse class for handling logical formulae.

- You create a `Predicate` object by calling the constructor `Predicate` with a string argument that is the name of the predicate.

  So, for instance, `Predicate red = new Predicate("red")` creates a constant, *red*.

- You can add arguments to predicates using `addTerm`.

  So for instance, `red.addTerm(new Predicate("box"))` changes the constant, *red* in to the predicate $red(box)$.

  `addTerm` always adds new terms to the end of the the predicate. So, for instance `red.addTerm(new Predicate("train"))` changes $red(box)$ into $red(box, train)$.

- If you want to change an argument then you need to use `setTerm(int i, Term t)`. So for instance, `red.setTerm(0, new Predicate("book"))` changes $red(box, train)$ to $red(book, train)$.

  NB. Predicate arguments count up from zero.

- You can access the arguments of a predicate using `getTerm(int i)`. So `red.getTerm(0)` applied to $red(book, train)$ returns *book*.

  book will be returned as an object of the `ail.syntax.Term` interface. Most of the classes for logical terms subclass objects (usually `ail.syntax.DefaultTerm`) that implement this interface. Depending on the situation, a programmer may therefore need to cast the `Term` object into something more specific.

- `getTerms()` returns a list of the arguments to a term. `getTermsSize()` returns an integer giving the number of arguments.

- `getFunctor()` returns a predicate's functor as a string. So, for instance, `red.getFunctor()` applied to $red(book, train)$ returns "red".

The VarTerm Class

`ail.syntax.VarTerm` is used to create variables in terms. Following Prolog conventions, all variables start with capital letters.

- You create a `VarTerm` object by calling the constructor `VarTerm` with a string argument that is the name of the variable.

  So, for instance, `VarTerm v1 = new VarTerm("A")` creates a variable, A.

- Since variables may be instantiated by unification to any logical term they subclass `ail.syntax.Predicate` and implement interfaces for other sorts of term, e.g., numerical terms via the `ail.syntax.NumberTerm` interface mentioned below. Once instantiated to some other sort of term a variable should behave like the relevant term object.

The NumberTerm interface and the NumberTermImpl Class

`ail.syntax.NumberTerm` and `ail.syntax.NumberTermImpl` are used to work with numerical terms. `NumberTermImpl` implements the `NumberTerm` interface.

- You create a `NumberTermImpl` object by calling the constructor `NumberTermImpl` with either a string argument that is the name of the number or a double.

  So, for instance, `NumberTermImpl value = new NumberTermImpl(2.5)` creates a numerical term, 2.5.

- You convert a `NumberTerm` into a value (e.g. to be used in a simulator) using the method `solve()` which returns a double. So, for instance, `value.solve()` applied to the numerical term 2.5 returns 2.5 as a double.

  When working with predicates that have numerical arguments – e.g., $distance(5.4)$ you may want to extract the argument (e.g, using `getTerm(0)`), cast it into a `NumberTerm` and then call `solve()` to get the actual number you want to work with.

Example 1

```
if (act.getFunctor().equals("move")) {                    1
   NumberTerm distance = (NumberTerm) act.getTerm(0);     2
   double d = distance.solve();                           3
   this.move(d);                                          4
}                                                         5
```

Example 1 shows some sample code that takes an action such as `move(2.5)` requested by the agent extracts the distance to be moved and then calls some internal method to perform the action in the environment passing in the double as an argument.

The Action class

Agents use `ail.syntax.Action` objects to request actions in the environment. `ail.syntax.Action` subclasses `ail.syntax.Predicate` and can generally be used just like a predicate.

Unifiers

Lastly we will briefly look at the use of unifiers with logical terms. Unifiers are represented by objects of the class `ail.syntax.Unifier`. We will use the syntax $Var - value$ to indicate that a unifier unifies the variable Var with the value $value$. We represent a unifier as a list of such variable-value pairs.

- Unifiers can be applied to any logical term (indeed to any object that implements the `ail.syntax.Unifiable` interface) by using the method `apply(Unifier u)`.

  So, for instance, suppose the Unifier, `u` is $[A - box]$, and `VarTerm a` is A. Then `a.apply(u)` will instantiate `a` as the term box.

- We can unify two terms using the `unifies(Unifiable t, Unifier u)` method.

  So, for instance, if predicate `p1` is $red(A)$ and predicate `p2` is $red(box)$ then `p1.unifies(p2, new Unifier u)` will turn `u` into the unifier $[A - box]$.

- You can extend an existing unifier in the same way.

  So, for instance, suppose `u` is $[A - box]$, `p1` is $red(A)$ and predicate `p2` is $red(B)$. Then `p1.unifies(p2, u)` will turn `u` into the unifier $[A - box, B - box]$.

- You can combine two unifiers using the `compose()` method (e.g., `u1.compose(u2)`. However you should be very careful about doing this unless you are certain that there is no variable unified with one term in the first unifier and a different term in the second.

4.2.4 The Message Class

This should only be relevant if you want to change the default handling of messages. This should only rarely be needed.

The message class is `ail.syntax.Message`. It has a number of fields which allow a message to specify the sender (a String), receiver (a String), propositional content (a Term), an *illocutionary force* or *performative* (an int), a message identifier (a StringTerm) and a thread identifier (a StringTerm).

It isn't necessary to use all these fields when creating a message and the simpliest constructor takes four arguments, the illocutionary force, sender, reciever and content, in that order. So for instance,

`Message m = new Message(1, "ag1", "ag2", new Predicate("red"))` sends the message "red" from `ag1` to `ag2` with illocutionary force 1.

Each language implemented in the AIL specifies its own meanings for illocutionary force. For instance the GWENDOLEN language (and its EASS variant) define 1 as *tell*, 2 as *perform* and 3 as *achieve*. So a GWENDOLEN agent would interpret the above example message as a *tell* message..

Messages come with a set of getter methods, `getIlForce()`, `getPropCont()`, `getReceiver()`, `getSender()` etc., for accessing the messages field.

Message objects also have a method `toTerm()` which will convert the message to a `Predicate` object of the form: `message(msgId, threadID, sender, receiever, ilFroce, propCont)`. Note that sender, receiver and ilforce are all converted to predicates (not to StringTerms and a NumberTerm) in this representation.

4.2.5 Extending executeAction

`ail.mas.DefaultEnvironment` implements a method called `executeAction`

```
public Unifier executeAction(String agName, Action act) throws AILexception {
```

As can be seen, `executeAction` takes the agent name and an action as parameters. The method returns a unifier. Sometimes part of the result of executing an action can be the instantiation of one of the arguments to the action predicate. This instantiation is provided by the unifier that is returned. It is this method that is called by agents when they want to perform an action.

In `DefaultEnvironment`, `executeAction` implements the default actions (discussed in section 6.9), message sending actions, updates fields relevant to model checking, and generates appropriate logging output. All these are important functions and so we *strongly recommend* that when overwriting `executeAction` you include a call to it (`super.executeAction(agName, act)`) at the end, outside of any conditional expressions. The method returns an empty unifier so this can be safely ignored or composed in subclassing environments.

Normally `executeAction` will need to handle several different actions. An easy way to do this is to use conditional statements that check the functor of the `Action` predicate (see Example 2)

Example 2

```
if (act.getFunctor().equals("red_button")) {                    1
   addPercept(agName, new Predicate("red_button_pressed");      2
   removePercept(agName, new Predicate("green_button_pressed"); 3
```

```
} else if (act.getFunctor().equals("green_button")) {
   addPercept(agName, new Predicate("green_button_pressed");
   removePercept(agName, new Predicate("red_button_pressed");
}
```

4.2.6 Initialising and Cleaning Up

Environments get created *before* any agents are created or added to them. This can sometimes cause problems if you want the environment to be configured in some way relating to the agents (e.g., setting up a location for each agent in the environment) before everything starts running.

After environments are created they can be configured using the AIL configuration file for the multi-agent system. The key/value pairs used will be specific to the environment.

The method `public void init_before_adding_agents()` is called on environments after configuration but before any agents are added to them. This is rarely used but occasionally there is some aspect of initialisation that has to happen *after* the use of any user supplied configuration files but before agents are added.

The method `public void init_after_adding_agents()` is called after the agents have been created and added to the environment but before the environment starts running. Therefore overriding this method can be a good way in which to perform any configuration that involves agents.

Similarly `public void cleanup()` is called at the end of a run of the multi-agent system and so can be used for any final clean up of the environment or to print out reports or statistics.

4.2.7 Exercises

Exercise 1

In the tutorial directory you will find an AIL configuration file, `PickUpAgent.ail`. This is a configuration for a simple multi-agent system consisting of one GWENDOLEN agent, `pickupagent.gwen`, and the Default Environment.

The agent is programmed to continue making `pickup` actions until it believes *holding_block*. If you run the multi-agent system you will observe it making repeated actions. Because the default environment does nothing with the `pickup` action the agent sees no outcomes to its efforts and so keeps trying.

Create a new environment for the agent that subclasses `ail.mas.DefaultEnvironment` and makes the `pickup` action result in the perception that the agent is *holding_block*.

A sample answer can be found in the `answers` directory.

Exercise 2

In the tutorial directory you will find a second GWENDOLEN agent, `lucky_dip_agent.gwen`. This agent is searching for a toy in three bins which are red, green and yellow. If it doesn't find a toy in any of the bins it will throw a tantrum. The agent can perform three actions.

search(Colour, A) It searches in the bin of colour, *Colour* and expects *A* to be unified to whatever it finds.

drop(A) It drops *A* (which it will have unified with something) and then waits until it sees *A* (e.g., if it does `drop(book)` it then waits until it perceives *see*(*book*) before it continues).

throw_tantrum

Create an environment for the agent that subclasses `ail.mas.DefaultEnvironment` and implements the five actions in a sensible way – i.e., unifying *A* appropriately for the search actions (e.g., to `book` or `toy` depending which bin the agent searches in), and adding an appropriate *see*(*A*) predicate. It isn't really necessary for anything to happen as a result of the tantrum action but it can if you want.

A sample answer can be found in the `answers` directory.

4.3 Tutorial 3 – Dynamic and Random Environments

This is the third in a series of tutorials on the use of the Agent Infrastructure Layer (AIL). This tutorial covers creating environments for agent programs which contain dynamic or random behaviour. Dynamic behaviour is behaviour that may occur without the agents doing anything to cause it. Random behaviour is when the outcome of an action, the input to a sensor, or the dynamic behaviour of the environment has some element of chance to it.

It should be noted that the EASS variant of the GWENDOLEN language is intended for use with dynamic and random environments and has its own customised support for them. If you are working with the EASS variant you may wish to skip this tutorial and use the EASS tutorial on environments instead.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ail_tutorials/tutorial3`.

The tutorial assumes a good working knowledge of Java programming and an understanding of the basics of constructing AIL environments as discussed in section 4.2

4.3.1 Dynamic Environments

A dynamic environment is one that gets to change in some way, typically to effect the percepts available in the system, without any agent taking any particular action. To do this the environment needs to be included in the scheduler that is used to decide which agent gets to act next.

The schedulers all expect to pick between objects that implement the `ajpf.MCAPLJobber` interface so the first thing a dynamic environment needs to do is implement this interface. This interface includes implementing a `do_job()` method which should contain the changes to be made when the environment runs. Once this is done the environment needs to be added to the scheduler and set up to receive notifications from the scheduler. Example 3 shows a simple dynamic environment for an agent that searches a grid in order to find a human. The agent program searches a grid by performing move_to actions. When the scheduler calls the the environment it inserts the perception that the robot sees a human to rescue.

Example 3

```
public class RobotEnv extends DefaultEnvironment                    1
                                        implements MCAPLJobber {     2
                                                                     3
  public RobotEnv() {                                                4
    super();                                                         5
    getScheduler().addJobber(this);                                  6
  }                                                                  7
                                                                     8
  @Override                                                          9
  public int compareTo(MCAPLJobber o) {                              10
    return o.getName().compareTo(getName());                         11
  }                                                                  12
                                                                     13
  @Override                                                          14
  public void do_job() {                                             15
    addPercept(new Predicate("human"));                              16
  }                                                                  17
                                                                     18
  @Override                                                          19
  public String getName() {                                          20
    return "gwendolen.ail_tutorials.tutorial3.RobotEnv";             21
  }                                                                  22
                                                                     23
  @Override                                                          24
  public Unifier executeAction(String agName, Action act)            25
                                          throws AILexception {      26
    if (act.getFunctor().equals("move_to")) {                        27
      Predicate robot_position = new Predicate("at");                28
      Predicate old_position = new Predicate("at");                  29
      robot_position.addTerm(act.getTerm(0));                        30
      robot_position.addTerm(act.getTerm(1));                        31
```

```
        old_position.addTerm(new VarTerm("X"));                    32
        old_position.addTerm(new VarTerm("Y"));                    33
        removeUnifiesPercept(old_position);                        34
        addPercept(robot_position);                                35
      }                                                            36
      return super.executeAction(agName, act);                    37
    }                                                              38
}                                                                  39
```

Line 2 shows that the class implements `MCAPLJobber`. At line 6 the environment is added as a jobber to the scheduler. Lines 9-23 show the three methods that need to be implemented for the `MCAPLJobber` interface. Schedulers generally compare jobbers by their names so `compareTo` implements this while `getName()` returns a name for the jobber. `do_job()` implements adding the perception of a human.

Lastly `executeAction` implements the result of the robot moving by changing the perceptions of its coordinates. This uses `removeUnifiesPercept` to remove the old robot position before it then asserts the new one.

The AIL configuration file, `searcher.ail`, executes a search and rescue agent in this environment.

A Note on Schedulers

The default scheduler used by `DefaultEnvironment` is `ail.mas.ActionScheduler`. This makes a random scheduling choice from among all its jobbers each time perceptions change in the environment. In general this works well but can become a problem if one of the jobbers (either an agent or the environment) gets stuck in a run in which it never changes any perceptions – e.g., an agent never takes an action or only does print actions (or similar) which don't alter perceptions – in these situations that jobber can run indefinitely without the scheduler ever being prompted to make another choice.

One situation where this may commonly arise is if the environment modifies some underlying fields or data structures but this information only becomes available as perceptions when an agent does something (e.g., moves close enough to see the change). In this case the line.

```
getScheduler().perceptChanged();
```

Can be asserted at the end of the `do_job()` method. This will prompt the scheduler to make a new choice even though no explicit perceptions have changed.

There are other three other schedulers in the current distribution:

NActionScheduler This functions as `ActionScheduler` except every n times it is invoked it forces a choice irrespective of whether perceptions have changed. This can be particularly useful if the environment is connecting to an external system and its use is discussed in the EASS tutorials since

the language is intended to work in this way. It isn't advisable to use the `NActionScheduler` in verification since it contains counters that will increase the number of model-checking states.

RoundRobinScheduler This scheduler acts like `ActionScheduler` except that instead of making a random choice between jobbers, it selects each in turn.

SingleAgentScheduler This is for situations when there is only one jobber and it effectively just returns that one jobber each time it is called.

If you wish to use a different scheduler in an environment then create the relevant scheduler object, add it to the environment (using `setScheduler(MCAPLScheduler s)` and add it as a percept listener to the environment (using `addPerceptListener(MCAPLPerceptListener s)`). If your environment subclasses `ail.mas.DefaultEnvironment` then you can call the method `setup_scheduler(AILEnv env, MCAPLScheduler s)` when the environment is constructed to do this for you.

4.3.2 Adding Randomness

Often we want an environment with some random behaviour to model, for instance, unreliable sensors or actuators.

It is tempting to add random behaviour to an environment simply through use of JAVA's `Random` class. However this will break the system's ability to record and replay runs through the program which can be very useful in debugging. The simplest way to add some random behaviour to an environment is to subclass `ail.mas.DefaultEnvironmentwRandomness` rather than `ail.mas.DefaultEnvironment`. This provides two `Choice` objects which are the mechanism the AIL uses to manage random behaviour for recording and replaying.

The `random_booleans` object has one method, `nextBoolean()` which will return either true or false. The `random_ints` object has one method, `nextInt(int i)`, which will return a random integer between `0` and `i`. Example 4 shows a sample environment for a search and rescue robot. This has a human at (1, 1) in the grid and the robot has a 50% chance of spotting the human if it is in the same grid square. If you run this program several times you will see that sometimes the robot finds the human quickly and sometimes it has to search the grid several times.

Example 4

```
public class RandomRobotEnv extends DefaultEnvironmentwRandomness 1{
  int human_x = 1;                                                 2
  int human_y = 1;                                                 3
                                                                   4
```

```
  public Unifier executeAction(String agName, Action act)          5
                                                 throws AILexception 6{
    if (act.getFunctor().equals("move_to")) {                       7
      Predicate robot_position = new Predicate("at");               8
      Predicate old_position = new Predicate("at");                 9
      robot_position.addTerm(act.getTerm(0));                       10
      robot_position.addTerm(act.getTerm(1));                       11
      old_position.addTerm(new VarTerm("X"));                       12
      old_position.addTerm(new VarTerm("Y"));                       13
      removeUnifiesPercept(old_position);                           14
      addPercept(robot_position);                                   15
      if (((NumberTerm) act.getTerm(0)).solve() == human_x          16
            && ((NumberTerm) act.getTerm(1)).solve() == human_y ) 17{
        if (random_booleans.nextBoolean()) {                        18
          addPercept(new Predicate("human"));                       19
        }                                                           20
      }                                                             21
    }                                                               22
    return super.executeAction(agName, act);                       23
  }                                                                 24
                                                                    25
}                                                                   26
```

In example 4 lines 17-19 add the percept, human, if `random_booleans.nextBoolean()` returns true.

Random Doubles

The AIL doesn't have support for random doubles (in part because model checking requires a finite state space) but it does let you specify a probability distribution over a set of choices. To do this you need to create your own `Choice` object. Say, for instance, in the above example the human is moving between the squares and could be at (0, 1), (1, 1) or (2, 1) with a 50% chance of being at (1, 1), a 30% chance of being at (2, 1) and a 20% chance of being at (0, 1).

Example 5 shows an environment with this behaviour. An integer `Choice` object, `human_location` is declared as a field in line 4. This is then instantiated by the `setMAS` method in lines 25-31. This method overrides the implementation in `DefaultEnvironmentwRandomness` so first we call the super-method, then we create the `Choice` object and lastly we add the choices to it – the humans x-coordinate is 1 with a probability of 0.5, 2 with a probability of 0.3 and 0 with a probability of 0.2. It is important to note that the `Choice` object can't be created when the class is created since it needs to be instantiated by a `MCAPLController` object [1]. Any instantiation of an Environment class that

[1]This is the object that governs the overrall behaviour of the system but which isn't available when the class it created. As part of setting up a multi-agent system in the AIL the `setMAS(MCAPLController m)` from the environment will be invoked at a suitable moment after the controller has been created.

involves choice methods (e.g., placing objects at random places within the environment) should be done in the `setMAS` method after the `Choice` objects have been instantiated and *not* in the class's constructor or initialisation methods.

In line 17 you can see the call to the `Choice` object's `get_choice()` method being invoked to return the correct integer.

Example 5

```
1  public class RandomRobotEnv2 extends DefaultEnvironmentwRandomness1 {
2    int human_x = 1;
3    int human_y = 1;
4    Choice<Integer> human_location;
5
6    public Unifier executeAction(String agName, Action act)
7                                        throws AILexception {
8      if (act.getFunctor().equals("move_to")) {
9        Predicate robot_position = new Predicate("at");
10       Predicate old_position = new Predicate("at");
11       robot_position.addTerm(act.getTerm(0));
12       robot_position.addTerm(act.getTerm(1));
13       old_position.addTerm(new VarTerm("X"));
14       old_position.addTerm(new VarTerm("Y"));
15       removeUnifiesPercept(old_position);
16       addPercept(robot_position);
17       human_x = human_location.get_choice();
18       if (((NumberTerm) act.getTerm(0)).solve() == human_x
19             && ((NumberTerm) act.getTerm(1)).solve() == human_y ) {
20         addPercept(new Predicate("human"));
21       }
22     }
23     return super.executeAction(agName, act);
24   }
25
26   public void setMAS(MAS m) {
27     super.setMAS(m);
28     human_location = new Choice<Integer>(m.getController());
29     human_location.addChoice(0.5, 1);
30     human_location.addChoice(0.3, 2);
31     human_location.addChoice(0.2, 0);
32   }
33
34 }
```

`Choice` objects can be created to return any object – integers, `Predicates`, `AILAgents`, etc.,[2] by being given the correct type and instantiated correctly. It is important to remember that the probabilities of the choices added by the `addChoice` method should add up to 1.

[2]The sample answer to the exercise at the end of this tutorial has an example of a `Choice` object for a JAVA `enum` type created just for the example.

If you genuinely need random doubles in an AIL environment then you can use JAVA's `Random` class but be aware that there may be issues with model checking search space and that the record and replay functionality will no longer work.

4.3.3 Record and Replay

When debugging a multi-agent program you sometimes want to replay the exact sequence of events that occurred in the problem run. To do this you first need to record the sequence. You can get an AIL program to record its sequence of choices (in this case choices about whether or not the agent perceives the human) by adding the line

```
ajpf.record = true
```

to the program's AIL configuration file. There is an example of this in the configuration file `searcher_random_record.ail` in the tutorial directory. By default this records the current path through the program in a file called `record.txt` in the directory, `records`, of the MCAPL distribution. You can change the file using `ajpf.replay.file =`.

To play back a record you include

```
ajpf.replay = true
```

in the program's AIL configuration file. The configuration file `searcher_random_replay.ail` is set up to replay runs generated by `searcher_random_record.ail`. Again, by default, this will replay the sequence from `record.txt`, but will use a different file if `ajpf.replay.file =` is set.

4.3.4 Exercise

Obviously for complex systems you often want to combine dynamic environments with randomness.

Adapt the various search and rescue environments so that the human moves one square in a random direction each time the environment's `do_job` method is called, to simulate a human moving independently around the search grid (NB. the search grid is 3x3 with coordinates ranging from (0, 0) to (2, 2) – you may assume it wraps if you wish).

You may use either `random_booleans` or `random_ints` to generate movement, however the sample answer (in the `answers` directory) creates a probability distribution over a custom JAVA `enum` type with the human most likely to remain stationary and least likely to move diagonally.

Since you are altering the position of the human in this environment, not the perceptions available, you will find that the scheduler will loop infinitely when selecting the enviroment unless you include the line

```
getScheduler().perceptChanged();
```

at the end of `do_job` or you use a different scheduler.

Check that your solution works with record and replay.

4.4 Tutorial 4 – Logging and Configuring Environment Behaviour

This is the fourth in a series of tutorials on the use of the Agent Infrastructure Layer (AIL). This tutorial covers using configuration and logging when programming with the AIL. These are particularly relevant for constructing environments but can be useful elsewhere.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ail_tutorials/tutorial4`.

You can find sample answers for all the exercises in this tutorial in the `answers` directory.

The tutorial assumes a working knowledge of Java programming and the implementation of logging in Java.

4.4.1 Logging

JAVA has a flexible API for implementing logging within programs. Unfortunately this does not work seamlessly within JPF and hence within AJPF . In order to enable logging to be used in AIL programs we have therefore provided a class `ajpf.util.AJPFLogger` which uses the native JAVA logging capabilities when not executed within AJPF but uses JPF's logging support when it is.

`AJPFLogger` supports the six logging levels of the JAVA logging framework, namely, `SEVERE`, `WARNING`, `INFO`, `FINE`, `FINER` and `FINEST`. If you want to print a log message at a particular log level, say `info`, you call the method `AJPFLogger.info(String logname, String message)` and similarly for `severe`, `warning` etc.

When logging, JAVA/JPF will print out all messages for a log at the set logging level and higher. So if you have set a log at level `FINE` in your AIL or JPF configuration file you will get all messages for `FINE`, `INFO`, `WARNING` and `SEVERE`.

The `logname` can be any string you like, though in general people use the class name for the logname. However there is no reason not to use log names associated with particular tasks your program performs or other groupings if that seems more sensible.

It is worth noting that Java's string manipulation is not particularly efficient. If you are constructing a complex string for a log message it can be worth putting the message within an if statement in order to prevent the string being constructed if it won't be printed. This can improve the speed of model checking, in particular. To help with this `AJPFLogger` provides four helper methods:

- `public boolean AJPFLogger.ltFinest(String logname)`,

- `public boolean AJPFLogger.ltFiner(String logname)`,
- `public boolean AJPFLogger.ltFine(String logname)`,
- `public boolean AJPFLogger.ltInfo(String logname)`

These return true if `logname` is set at or below a particular logging level. Therefore you can use the construction:

```
if (AJPFLogger.lfFine(logname)) {
    String s = ....
     .... code for constructing your log message ...
    AJPFLogger.fine(logname, s);
}
```

in order to ensure the string construction only takes place if the message will actually get logged.

Exercise

In the tutorial directory you will find a simple environment for a search and rescue robot (`RobotEnv.java`) together with code and a configuration file for the robot (`searcher.ail`, `searcher.gwen`). The robot moves around a 3x3 square searching for a human which may randomly appear in any square. If the human appears the robot sends a message to some lifting agent and then stops. If you run this program with the supplied AIL configuration you will see it printing out the standard messages from `ail.mas.DefaultEnvironment` noting when the robot moves and when it sends a message.

Adapt the environment with the following log messages

- If logging is set to `INFO` it prints out a message when the system first decides a human is visible,
- If logging is set to `FINE` it prints a message every time the agent is informed the human is visible (not just when the human first appears) and,
- If logging is set to `FINER` it prints a message every time the agent checks its percepts.

Experiment with setting log levels in the AIL configuration file. Note that by default anything at `INFO` or higher gets printed. If you don't want to see `INFO` level log messages then you need to configure the logger to a higher level e.g. `log.warning = gwendolen.ail_tutorials.tutorial4.RobotEnv`.

4.4.2 Customized Configuration

You can use the `key = value` mechanism inside AIL configuration files to create customisation for your own AIL programs. When the configuration file is parsed all the properties are stored in an `ail.util.AILConfig` object which is itself an extension of the JAVA `java.util.Properties` class.

The `Properties` class has two methods of particular note:

- `public boolean containsKey(String key)` tells you if a particular key is contained in the configuration.
- `public Object get(String key)` returns the value stored for the key. If parsed from an AIL configuration file this will return a `String`.

You can then use JAVA methods such as `Boolean.valueOf(String s)` and `Integer.valueOf(String s)` to convert that value into a boolean, integer or other type if desired.

Obviously in order to add your own key/value pairs to the configuration you need to be able to access the `AILConfig` object. The easiest way to do this is via your environment. During system initialisation a method `public void configure(AILConfig config)` is called on any AIL environment. The default implementation of this method does nothing, but it is easy enough to override this in a customised environment and check any keys you are interested in.

This can be particularly useful in environments used for verification where you may wish to have a range of slightly different behaviours in the environment for efficiency reasons. Listing 6 shows (a slightly shortened version of) the `configure` method used by `eass.verification.leo.LEOVerificationEnvironment` which was used for the Low Earth Orbit satellite verifications described in [Dennis et al., 2014]. Most of the values used here are `true`/`false` values parsed into booleans but in line 34 you can see a value that is being treated as a string (where the target formation can be either `line` or `square`). In that paper you can see that different properties were proved against different sets of percepts. The `configure` method was used in conjunction with AIL configuration files for each example in order to tweak the environment to use the correct settings. You can find all the configuration files in the `examples/eass/verification/leo` directory.

Example 6

```
1  public void configure(AILConfig configuration) {
2    if (configuration.containsKey("testing_thrusters")) {
3      testing_thrusters = Boolean.valueOf((String) configuration.get("testing_thrusters"));
4    }
5
6    if (configuration.containsKey("allthrusters")) {
7      allthrusters = Boolean.valueOf((String) configuration.get("allthrusters"));
8    }
9
10   if (configuration.containsKey("allpositions")) {
11     allpositions = Boolean.valueOf((String) configuration.get("allpositions"));
12   }
13
14   if (configuration.containsKey("formation_line")) {
15     formation_line = Boolean.valueOf((String) configuration.get("formation_line"));
16   }
```

```
                                                                    17
    if (configuration.containsKey("formation_square")) {           18
      formation_square = Boolean.valueOf((String) configuration.get("formation_square");
    }                                                               20
                                                                    21
    if (configuration.containsKey("all_can_break")) {              22
      all_can_break = Boolean.valueOf((String) configuration.get("all_can_break"));
    }                                                               24
                                                                    25
    if (configuration.containsKey("changing_formation")) {         26
      changing_formations = Boolean.valueOf((String) configuration.get("changing_formatio
      if (changing_formations) {                                    28
        formation_line = true;                                      29
        formation_square = true;                                    30
      } else {                                                      31
        if (configuration.containsKey("initial_formation")) {       32
          if (configuration.get("initial_formation").equals("line")) {
            formation_line = true;                                  34
            formation_square = false;                               35
          } else {                                                  36
            formation_line = false;                                 37
            formation_square = true;                                38
          }                                                         39
        } else {                                                    40
          formation_line = true;                                    41
          formation_square = false;                                 42
        }                                                           43
      }                                                             44
    }                                                               45
  }                                                                 46
```

Exercise

Adapt `RobotEnv` so it has a configuration option in which the robot always sees a human when it moves rather than there being a random chance of seeing a human.

Chapter 5

Verifying Programs Using AJPF

This chapter consists of tutorials on the use of AJPF to verify multi-agent sytems.

5.1 Tutorial 1 – The Property Specification Language

This is the first in a series of tutorials on the use of the AJPF model checking program. This tutorial covers the basics of configuring a model-checking run and writing properties in AJPF's property specification language.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ajpf_tutorials/tutorial1`.

The tutorials assume some familiarity with the basics of running Java programs either at the command line or in Eclipse and some familiarity with the syntax and semantics of Linear Temporal Logic.

5.1.1 Setting up Agent Java Pathfinder

Before you can run AJPF it is necessary to set up your computer to use Java Pathfinder (JPF). There are instructions for doing this in chapter 2.

The key point is that you need to create a file called `.jpf/site.properties` in your home directory on the computer you are using. In this file you need to put one line which assigns the path to the MCAPL distribution to the key `mcapl`. For instance if you have your MCAPL distribution in your home directory as folder called `mcapl` then `site.properties` should contain the line.

```
mcapl = ${user.home}/mcapl
```

We strongly recommend that you also set up an environment variable, `$AJPF_HOME`, set it to the path to the MCAPL directory and add this to your `.bashrc` or equivalent start-up files.

5.1.2 A Simple Model Checking Attempt

To run AJPF you need to run the program `gov.jpf.tool.RunJPF` which is contained in `lib/3rdparty/RunJPF.jar` in the MCAPL distribution. Alternatively you can use the `run-JPF (MCAPL)` Run Configuration in IntelliJ or Eclipse.

You need to supply a JPF Configuration file as an argument. You will find a sample file in `src/examples/gwendolen/ajpf_tutorials/tutorial1/lifterandmedic.jpf`. If you run this you should see output like the following:

```
loading property file: /Users/louisedennis/.jpf/site.properties
loading property file: /Users/louisedennis/eclipse-workspace/mcapl/jpf.properties
loading property file: /Users/louisedennis/eclipse-workspace/mcapl/src/examples/gwendol
collected native_classpath=/Users/louisedennis/eclipse-workspace/mcapl/bin,/Users/louis
collected native_libraries=null
JPF build information:
 os.arch = x86_64
 date.tip = 2012-01-13 13:30 -0800
 java.version = 1.6.0_26
 user.country = US
 author = Peter Mehlitz <Peter.C.Mehlitz@nasa.gov>
 os.version = 10.5.8
 os.name = Mac OS X
 upstream = http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
 repository = file://flyer/Users/pmehlitz/projects/eclipse/jpf-core
 revision = 647:b8b86ac8f503
JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights re


====================================================== system under test
ail.util.AJPF_w_AIL.main("/Users/louisedennis/eclipse-workspace/mcapl/src/examples/gwen

====================================================== search started: 30/03/23 12:09
MCAPL Framework Version 2023
ANTLR Tool version 4.4 used for code generation does not match the current runtime vers
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:14
states:             new=2055,visited=210,backtracked=2265,end=3
search:             maxDepth=111,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), da
```

```
heap:               new=1179555,released=1172517,maxLive=3965,gcCycles=2265
instructions:       116768553
max memory:         2442MB
loaded code:        classes=308,methods=5043

============================================================ search finished: 30/03/23 12:09
```

Note this will take several seconds to generate. We will discuss in future tutorials how to get more detailed output from the model checker.

At the moment the key point is the fact that it states `no errors detected`. This means that the property supplied to the model checker was true for this program.

5.1.3 JPF Configuration Files

`lifterandmedic.jpf` is a JPF Configuration file. There are a large number of configuration options for JPF. JPF documentation is currently being updated but you can find some information in its github repository *https://github.com/javapathfinder/jpf-core*. We will only discuss a handful of these options. If you open `lifterandmedic.jpf` you should see the following:

```
@using = mcapl

target = ail.util.AJPF_w_AIL
target.args = ${mcapl}/src/examples/gwendolen/ail_tutorials/tutorial1/answers/ex2.ail,${mcapl}/src/examples/
```

We explain each line of this below.

@using = mcapl Means that the proof is using the home directory for `mcapl` that you set up in `.jpf/site.properties`.

target = ail.util.AJPF_w_AIL This is the Java file containing the main method for the program to be model checked. By default when model checking a program implemented using the AIL, you should use `ail.util.AJPF_w_AIL` as the target. If you are familiar with running programs in the AIL, this class is very similar to `ail.mas.AIL` but with a few tweaks to set up and optimise model checking.

target.args =... This sets up the arguments to be passed to `ail.util.AJPF_w_AIL`. `ail.util.AJPF_w_AIL` takes three arguments. In the configuration file these all have to appear on one line, separated by commas (but *no spaces*). This means you can not see them all in the file print out above. In order the arguments are:

1. The first is an AIL configuration file. In this example the file is `${mcapl}/src/examples/gwendolen/ail_tutorials/tutorial1/answers/ex2.ail`

which is a configuration file for a multi-agent system written in the GWENDOLEN language.

2. The second argument is a file containing a list of properties in AJPF's property specification language that can be checked. In this example this file is `lifterandmedic.psl` in the directory for this tutorial.
3. The last argument is the name of the property to be checked, `1` in this case.

5.1.4 The Property Specification Language

Syntax The syntax for property formulæ ϕ is as follows, where ag is an "agent constant" referring to a specific agent in the system, and f is a ground first-order atomic formula (although it may use _, as in Prolog, to indicate variables which may match any value):

$$\begin{array}{rl} \phi ::= & \mathcal{B}_{ag}\, f \mid \mathcal{G}_{ag} f \mid \mathcal{A}_{ag} f \mid \mathcal{I}_{ag} f \mid \mathcal{ID}_{ag} f \mid \mathcal{P}(f) \mid \\ & \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \phi\ \mathbf{U}\, \phi \mid \phi \mathcal{R} \phi \mid \diamond\phi \mid \Box\phi \end{array}$$

Here, $\mathcal{B}_{ag}\, f$ is true if ag believes f to be true, $\mathcal{G}_{ag} f$ is true if ag has a goal to make f true, and so on (with $\mathcal{A}$ representing actions, $\mathcal{I}$ representing intentions, $\mathcal{ID}$ representing the intention to take an action, and $\mathcal{P}$ representing percepts, i.e., properties that are true in the environment).

The following representation of this syntax is used in AJPF's property specification files:

$$\begin{array}{rl} \phi ::= & \texttt{B(}ag\texttt{, }f\texttt{)} \mid \texttt{G(}ag\texttt{, }f\texttt{)} \mid \texttt{D(}ag\texttt{, }f\texttt{)} \mid \texttt{I(}ag\texttt{, }f\texttt{)} \mid \texttt{ItD(}ag\texttt{, }f\texttt{)} \mid \texttt{P(}f\texttt{)} \mid \sim \phi \\ \phi' ::= & \phi \mid \phi' \texttt{ || } \phi' \mid \phi' \texttt{ \& } \phi' \mid \phi' \texttt{ U } \phi' \mid \phi' \texttt{ R } \phi' \mid \texttt{<>}\phi' \mid \texttt{[]}\phi' \end{array}$$

Note, in particular, that in property specification files "not" ($\sim$) must always appear in an innermost position next to one of the BDI agent properties such as `B(`ag`, `f`)`.

It is also possible to use ϕ`->`ψ as shorthand for $\neg\phi \vee \psi$ in property specification files.

Semantics We summarise semantics of property formulæ. Consider a program, P, describing a multi-agent system and let MAS be the state of the multi-agent system at one point in the run of P. MAS is a tuple consisting of the local states of the individual agents and of the environment. Let $ag \in MAS$ be the state of an agent in the MAS tuple at this point in the program execution. Then

$$MAS \models_{MC} \mathcal{B}_{ag}\, f \quad \text{iff} \quad ag \models \mathcal{B}_{ag}\, f$$

where $\models$ is logical consequence as implemented by the agent programming language. The semantics of $\mathcal{G}_{ag} f$, $\mathcal{I}_{ag} f$ and $\mathcal{ID}_{ag} f$ similarly refer to internal im-

plementations of the language interpreter [1]. The interpretation of $\mathcal{A}_{ag}f$ is:

$$MAS \models_{MC} \mathcal{A}_{ag}f$$

if, and only if, the last action changing the environment was action f taken by agent ag. Finally, the interpretation of $\mathcal{P}(f)$ is given as:

$$MAS \models_{MC} \mathcal{P}(f)$$

if, and only if, f is a percept that holds true in the environment.

The other operators in the AJPF property specification language have standard PLTL semantics [Emerson, 1990] and are implemented as Büchi Automata as described in [Gerth et al., 1996, Courcoubetis et al., 1992]. Thus, the classical logic operators are defined by:

$$\begin{array}{rcl} MAS \models_{MC} \varphi \vee \psi & \text{iff} & MAS \models_{MC} \varphi \text{ or } MAS \models_{MC} \psi \\ MAS \models_{MC} \neg\phi & \text{iff} & MAS \not\models_{MC} \phi. \end{array}$$

The temporal formulæ apply to runs of the programs in the JPF model checker. A run consists of a (possibly infinite) sequence of program states MAS_i, $i \geq 0$ where MAS_0 is the initial state of the program (note, however, that for model checking the number of *different* states in any run is assumed to be finite). Let P be a multi-agent program, then:

$MAS \models_{MC}$	$\varphi \mathbf{U} \psi$	iff	in all runs of P there exists a state MAS_j such that $MAS_i \models_{MC} \varphi$ for all $0 \leq i < j$ and $MAS_j \models_{MC} \psi$.
$MAS \models_{MC}$	$\varphi \mathcal{R} \psi$	iff	either $MAS_i \models_{MC} \varphi$ for all i or there exists MAS_j such that $MAS_i \models_{MC} \varphi$ for all $0 \leq i \leq j$ and $MAS_j \models_{MC} \varphi \wedge \psi$.

Conjunction $\wedge$ and the common temporal operators $\diamond$ (eventually) and $\Box$ (always) are, in turn, derivable from $\vee$, $\mathbf{U}$ and $\mathcal{R}$ in the usual way [Emerson, 1990].

5.1.5 Exercises

If you look in `lifterandmedic.psl` you should see the following:

```
1: [] (~B(medic, bad))
```

So this file contains one formula, labelled, `1` and the formula is equivalent to $\Box\neg\mathcal{B}_{\texttt{medic}}\, bad$ – which means it is always the case the medic agent doesn't believe the formula `bad` (or alternatively that the medic agent never believes *bad*.

As noted previously the multi-agent system in `ex2.ail` is a GWENDOLEN program and is, in fact, the one described in section 6.8. It isn't necessary, for this tutorial, to understand the implementation of the BDI modalities (belief,

[1] We briefly cover the GWENDOLEN implementation in section 6.10.1.

goal, intention etc.) in the GWENDOLEN interpreter but a brief discussion is included in section 6.10.1.

Adapt the JPF configuration file and extend the property specification file in the tutorial directory in order to verify the following properties of the multi-agent system. You can find sample answers in the `answers` directory.

1. Eventually the lifter believes $human(5, 5)$.

$$\diamond\mathcal{B}_{\texttt{medic}}\, human(5, 5)$$

2. Eventually the medic has the goal $assist\_human(5, 5)$.

$$\diamond\mathcal{G}_{\texttt{medic}} human(5, 5)$$

3. Eventually the lift believes $human(3, 4)$ and eventually the lifter believes $holding(rubble)$.

$$\diamond\mathcal{B}_{\texttt{lifter}}\, human(3, 4) \wedge \diamond\mathcal{B}_{\texttt{lifter}}\, holding(rubble)$$

4. If the lifter has the intention to $goto55then34$ then eventually the medic will have the goal $assist\_human(5, 5)$.

$$\mathcal{I}_{\texttt{lifter}} goto55then34 \Rightarrow \mathcal{G}_{\texttt{medic}} assist\_human(5, 5)$$

5. It is always the case that if the lifter does $move\_to(5, 5)$ then $human(5, 5)$ becomes perceptible.

$$\Box(\mathcal{A}_{\texttt{lifter}} move\_to(5, 5) \Rightarrow \mathcal{P}(human(5, 5))$$

6. Eventually the lifter intends to move to (5, 5).

$$\diamond\mathcal{ID}_{\texttt{lifter}} move\_to(5, 5)$$

7. Eventually the lifter intends to send the medic a perform request to assist the human in some square.

$$\diamond\mathcal{ID}_{\texttt{lifter}} send(\texttt{medic}, 2, assist\_human(\_, \_))$$

   For this you should consult the section on intending to send messages in section 6.10.1.

5.2 Tutorial 2 – JPF Configuration Files: Troubleshooting Model Checking

This is the second in a series of tutorials on the use of the AJPF model checking program. This tutorial covers JPF configuration files in more detail as well as techniques for troubleshooting model checking.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ajpf_tutorials/tutorial2`.

The tutorials assume some familiarity with the basics of running Java programs either at the command line or in IntelliJ or Eclipse and some familiarity with the syntax and semantics of Linear Temporal Logic, and the use of Büchi Automata in model checking.

5.2.1 JPF Configuration Files

As mentioned in section 5.1, JPF has an extensive set of configuration options. We only examined the most basic in section 5.1 but in this tutorial we will cover a few more that are useful, particularly when debugging a program you are attempting to model check.

In the tutorial directory you will find a simple GWENDOLEN program, `twopickupagents.gwen`. This contains two agents, one holding a block and one holding a flag. Each agent puts down what they are holding. If the agent with the block puts it down before the agent with the flag puts the flag down, then the agent with the flag will pick up the box. The agent with the flag also performs an action with random consequences after it puts down the flag.

TwoPickUpAgents_basic.jpf

`TwoPickUpAgents_basic.jpf` is a minimal configuration file containing only options discussed in section 5.1. This generates the following output (ignoring some initial system information):

```
JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.


============================================================ system under test
ail.util.AJPF_w_AIL.main("/Users/louisedennis/eclipse-workspace/mcapl/src/examples/gwendolen/ajpf

============================================================ search started: 15/03/19 11:45
MCAPL Framework 2020
ANTLR Tool version 4.4 used for code generation does not match the current runtime version 4.7ANT
============================================================ results
no errors detected

============================================================ statistics
elapsed time:       00:00:05
states:             new=31,visited=32,backtracked=63,end=0
search:             maxDepth=7,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=31
heap:               new=412761,released=409463,maxLive=3827,gcCycles=63
instructions:       28180407
max memory:         437MB
loaded code:        classes=326,methods=5084
```

```
====================================================== search finished: 15/03/19 11:45
```

This is obviously fine as output in situations where the model checking completes quickly and with `no errors detected` but gives the user very little to go on if there is a problem or the model checking is taking a long time and they are not sure whether to kill the attempt or not.

TwoPickUpAgents_ExecTracker.jpf

`TwoPickUpAgents_ExecTracker.jpf` adds the configuration option:

```
listener+=,.listener.ExecTracker
et.print_insn=false
et.show_shared=false
```

Adding `listener.ExecTracker` to JPF's listeners means that it collects more information about progress as it goes and then prints this information out. The next two lines suppress some of this information which is generally less useful in AJPF. With these settings the following output is generated (only the start is shown):

```
   # choice: gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT" ,1/1,isCascaded:false}
   # garbage collection
---------------------------------- [1] forward: 0 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [2] forward: 1 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [3] forward: 2 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [4] forward: 3 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [5] forward: 4 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [6] forward: 5 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>
   # garbage collection
---------------------------------- [7] forward: 6 visited
---------------------------------- [6] backtrack: 5
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0
   # garbage collection
---------------------------------- [7] forward: 7 visited
---------------------------------- [6] backtrack: 5
---------------------------------- [6] done: 5
```

```
---------------------------------- [5] backtrack: 4
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0,>1]
   # garbage collection
---------------------------------- [6] forward: 8 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>0,1]
   # garbage collection
---------------------------------- [7] forward: 6 visited
---------------------------------- [6] backtrack: 8
```

Every time JPF generates a new state for model checking it assigns that state a number. In the output here you can see it generating new states 0 through to 7 and advancing forward to each state. You then see it backtracking back to state 5 (which is fully explored `done`) and then state 4 at which point it finds a branching point in the search space and advances to state 8 and then again to state 6 which it has visited already and so backtracks to 8.

Typically search space branching is caused either whenever a random value is generated. This happens most often when the multi-agent system scheduler must choose between several agents.

Random value generation activates an `IntChoiceFromSet` choice generator (which picks a random integer from a set – usually picking one number from a range). The scheduler keeps track of the agents which are awake and assigns an integer to them. Since there are only two agents, there is no choice if one is asleep, but you can see when the choice between `0` and `1`.

The numbers in square brackets – `[7]`, `[6]` etc. indicate the depth that model checking has reached in the search tree. If these numbers become very large without apparent reason then it may well be the case that the search has encountered an infinite branch of the tree and needs to be killed.

Logging

JPF suppresses the logging configuration you have in your AIL configuration files so you need to add any logging configurations you want to the JPF configuration file. Useful classes when debugging a model checking run are

ail.mas.DefaultEnvironment At the `info` level this prints out any actions the agent performs. Since the scheduler normally only switches between agents when one sleeps or performs an action this can be useful for tracking progress on this model checking branch.

ajpf.MCAPLAgent At the `info` level this prints information when an agent sleeps or wakes. Again this can be useful for seeing what has triggered a scheduler switch. It can also be useful for tracking which agents are awake and so deducing which one is being picked from the set by the `IntChoiceFromSet` choice generator.

ajpf.product.Product At the `info` level this prints out the current path through the search tree being explored by the agent. This can be useful just to get a feel for the agents' progress through the search space.

It can also be useful, when an error is thrown and in conjunction with some combination of logging actions, sleeping and waking behaviour and (if necessary) internal agent states, to work out why a property has failed to hold.

It also prints the message `Always True from Now On` when exploration of a branch of the search tree is halted because the system deduces that the property will be true for the rest of that branch. This typically occurs when the property is something like $\diamond\phi$ (i.e., ϕ will eventually occur) and the search space is pruned once ϕ becomes true.

ajpf.psl.buchi.BuchiAutomaton At the `info` level this prints out the Büchi Automaton that has been generated from the the property that is to be proved. Again this is useful, when model checking fails, for working out what property was expected to hold in that state.

ail.semantics.AILAgent At the `fine` level this prints out the internal agent state once every reasoning cycle. Be warned that this produces a lot of output in the course of a model checking run.

In general, when working on a program for model checking it is useful to have the ExecTracker listener enabled and `ajpf.MCAPLAgent`, `ajpf.product.Product` and any environment loggers (so typically `ail.mas.DefaultEnvironment` and any sub-classes of that you are using) set at info. This provides a useful starting point for accessing information about model checking.

`TwoPickUpAgents_Logging.jpf` has this set up. It's output starts

```
[INFO] Adding 0 to []
---------------------------------- [1] forward: 0 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[INFO] ag2 done putdown(flag)
   # garbage collection
[INFO] Adding 1 to [0]
---------------------------------- [2] forward: 1 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
   # garbage collection
[INFO] Adding 2 to [0, 1]
---------------------------------- [3] forward: 2 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[INFO] Block 1 is visible
   # garbage collection
[INFO] Adding 3 to [0, 1, 2]
---------------------------------- [4] forward: 3 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[INFO] Block 2 is visible
[INFO] ag2 done random
   # garbage collection
```

```
[INFO] Adding 4 to [0, 1, 2, 3]
---------------------------------- [5] forward: 4 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[INFO] Sleeping agent ag2
[INFO] Waking agent ag2
[INFO] ag1 done putdown(block)
   # garbage collection
[INFO] Adding 5 to [0, 1, 2, 3, 4]
---------------------------------- [6] forward: 5 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[INFO] Sleeping agent ag2
   # garbage collection
[INFO] Adding 6 to [0, 1, 2, 3, 4, 5, 6]
[INFO] Always True from Now On
---------------------------------- [7] forward: 6 visited
---------------------------------- [6] backtrack: 5
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0,
[INFO] Sleeping agent ag1
   # garbage collection
[INFO] Adding 7 to [0, 1, 2, 3, 4, 5, 7]
[INFO] Always True from Now On
---------------------------------- [7] forward: 7 visited
---------------------------------- [6] backtrack: 5
---------------------------------- [6] done: 5
---------------------------------- [5] backtrack: 4
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0,
[INFO] ag1 done putdown(block)
   # garbage collection
[INFO] Adding 8 to [0, 1, 2, 3, 4]
---------------------------------- [6] forward: 8 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[INFO] Sleeping agent ag2
   # garbage collection
[INFO] Adding 6 to [0, 1, 2, 3, 4, 8]
---------------------------------- [7] forward: 6 visited
---------------------------------- [6] backtrack: 8
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0,
[INFO] Sleeping agent ag1
   # garbage collection
[INFO] Adding 7 to [0, 1, 2, 3, 4, 8]
---------------------------------- [7] forward: 7 visited
---------------------------------- [6] backtrack: 8
---------------------------------- [6] done: 8
---------------------------------- [5] backtrack: 4
---------------------------------- [5] done: 4
---------------------------------- [4] backtrack: 3
```

```
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[INFO] Block 2 is not visible
[INFO] ag2 done random
   # garbage collection
[INFO] Adding 9 to [0, 1, 2, 3]
---------------------------------- [5] forward: 9 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[INFO] Sleeping agent ag2
[INFO] Waking agent ag2
[INFO] ag1 done putdown(block)
   # garbage collection
[INFO] Adding 10 to [0, 1, 2, 3, 9]
---------------------------------- [6] forward: 10 new
```

You can see the additional information provided by the loggers here, in terms of printing out the current path through the search tree, reporting on sleeping and waking behaviour, etc.,

Important Note: While the additional output information can be very useful for understanding what is happening during a model checking run, printing output slows down the computation. If speed of model checking is important then it is best to turn off all logging and the `ExecTracker`.

Saving the log to a file

It is possible to save log message generated by the AIL to a file by including `log.output = filename` (where `filename` is the name of the file you want to use) in your JPF configuration file. Unfortunately this does not save the output of the `ExecTracker` to the file but may nevertheless be useful.

5.2.2 What to do when Model Checking Fails

`TwoPickUpAgents_FalseProp.jpf` attempts to prove the property $\diamond\mathcal{B}_{ag2}\, hold(block)$ which isn't true. The configuration file uses the normal loggers but doesn't have the `ExecTracker` listener[2]. The following output is generated.

```
====================================================== system under test
ail.util.AJPF_w_AIL.main("/Users/lad/Eclipse/mcapl/src/examples/gwendolen/ajpf_tutorials/tutorial2/TwoPickU

====================================================== search started: 28/04/17 15:58
[INFO] Adding 0 to []
[INFO] ag2 done putdown(flag)
[INFO] Adding 1 to [0]
[INFO] Adding 2 to [0, 1]
[INFO] Block 1 is visible
[INFO] Adding 3 to [0, 1, 2]
[INFO] Block 2 is visible
```

[2]Largely to keep the output compact.

```
[INFO] ag2 done random
[INFO] Adding 4 to [0, 1, 2, 3]
[INFO] Sleeping agent ag2
[INFO] Waking agent ag2
[INFO] ag1 done putdown(block)
[INFO] Adding 5 to [0, 1, 2, 3, 4]
[INFO] Sleeping agent ag2
[INFO] Adding 6 to [0, 1, 2, 3, 4, 5]

====================================================== error 1
ajpf.MCAPLListener
An Accepting Path has been found:
[MS: 0, BS: 2, UN: 0], [MS: 1, BS: 2, UN: 0], [MS: 2, BS: 2, UN: 0], [MS: 3, BS: 2, UN: 0], [MS: 4, BS: 2, UN: 0],
[MS: 5, BS: 2, UN: 0], [MS: 6, BS: 2, UN: 0],

====================================================== snapshot #1
no live threads

====================================================== results
error #1: ajpf.MCAPLListener "An Accepting Path has been found:  [MS: 0, BS: 2, ..."
```

As can be seen at the end of the failed run this prints out the accepting path that it has found that makes the property false. This path is a sequence of triples consisting of the state in the model, `MS`, the state in the Büchi automaton generated from the negation of the property, `BS`, and lastly a count of the number of until statements that have been passed in this branch/loop of the search space (This counter is explained in [Gerth et al., 1996] – it isn't normally useful for debugging properties but is included for completeness).

So we can see that the accepting path through the model is 0,1,2,3,4,5,6 and we can work out what happens on that path from the logging output: ag2 puts down the flag, both blocks becomes visible, ag2 does random and then sleeps, ag1 puts down the block, waking ag2 which then sleeps again. All these states in the model are paired with state 2 in the Büchi Automaton. To see the Büchi Automaton you have to add `ajpf.psl.buchi.BuchiAutomaton` to the logging.

If you do this you get the following print out at the start:

```
[INFO] Number: 2
Incoming States: 0,2,
True in this State: ~B(ag2,hold(block())),~T R ~B(ag2,hold(block())),
True in next State: ~T R ~B(ag2,hold(block())),
```

The property has created a very simple Büchi Automaton. It has been given the number 2 in the automaton generation process. It has two incoming states 0 (which is the start state) and 2 (i.e., itself). It has two properties that hold in that state $\neg\mathcal{B}_{ag2}\, hold(block)$ (ag2 doesn't believe it is holding the block) and $\neg\top\mathcal{R}\neg\mathcal{B}_{ag2}\, hold(block)$ (false ($\neg\top$) released by ag2 doesn't believe it is holding the block – which under standard LTL transformations means $\Box\neg\mathcal{B}_{ag2}\, hold(block)$ (it is always the case that ag2 doesn't believe it is holding the block)). In the next state this should also hold. For debugging failed model checking runs it is normally safe to ignore the properties that should hold in the

next state, and any temporal properties that should hold in the current state, so this automaton can be visualised as in figure 5.1.

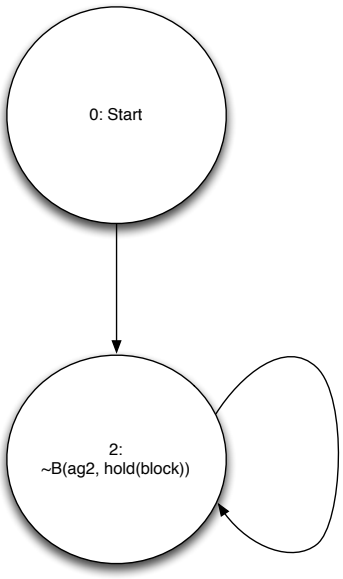

Figure 5.1: The Property Automaton for $\neg \diamond \mathcal{B}_{ag2}\, hold(block)$

I.e. a single state automaton in which $\mathcal{B}_{ag2}\, hold(block)$ is never true. The model checking has failed because this state is true for every state in the model along the path 0,1,2,3,4,5,6 (you can look in the program to se why).

5.2.3 Replaying a Counter-example

When model-checking fails the branch it has failed on has essentially generated a counter-example for the property. Sometimes you will want to replay this counter-example in the AIL without performing model-checking. AJPF has *record* and *replay* functionality to assist with this.

To obtain a record of a model checking run you will need to set *logging mode* in the *AIL configuration file* (using `ajpf.record = true`) and then set the the logging level to fine in `ajpf.util.choice.ChoiceRecord`).

`TwoPickupAgents_Recording.txt` has this setup. Its output starts:

```
   # choice: gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT" ,1/1,isCascaded:fals
   # garbage collection
---------------------------------- [1] forward: 0 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascade
[FINE] Record: [0]
```

```
[FINE] Record: [0, 0]
   # garbage collection
---------------------------------- [2] forward: 1 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[FINE] Record: [0, 0, 0]
[FINE] Record: [0, 0, 0, 0]
   # garbage collection
---------------------------------- [3] forward: 2 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[FINE] Record: [0, 0, 0, 0, 0]
   # garbage collection
---------------------------------- [4] forward: 3 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[FINE] Record: [0, 0, 0, 0, 0, 0]
   # garbage collection
---------------------------------- [5] forward: 4 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[FINE] Record: [0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   # garbage collection
---------------------------------- [6] forward: 5 new
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,>C
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   # garbage collection
---------------------------------- [7] forward: 6 visited
---------------------------------- [6] backtrack: 5
   # choice: gov.nasa.jpf.vm.choice.IntChoiceFromSet[id="probabilisticChoice",isCascaded:false,0,
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0]
[FINE] Record: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0]
   # garbage collection
```

The lines starting `[FINE] Record:` show the record of choices at that point. To replay a particular branch through the search tree in AIL without model checking do the following:

1. Paste the relevant record list, e.g. `[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]` into the file `record.txt` in the `records` directory of the MCAPL distribution.

2. Replace the line `ajpf.record = true` in your AIL configuration file with

the line `ajpf.replay = true`.

3. Run the program in AIL as normal.

If you want to use a different file to `record.txt` to store the record for replay you can, but you will need to set `ajpf.replay.file` in the AIL configuration file appropriate in order to replay that record.

(Use of Random) Please note: that it is important for record and replay to work correctly that all choice points in the program are used in the record. Among other things this means that Java's `Random` class **can not** be used in constructing environments and AJPF's `Choice` class should be used instead.

5.2.4 Forcing Transitions in the Agent's Reasoning Cycle

In the examples considered so far in this tutorial, AJPF has only generated new states for the model when JPF would generate a state. This has been when there has been a scheduling choice between the two agents. While this is often sufficient for many model checking problems it does mean that the property is only checked when the agent program has done significant processing. This means that states of interest can sometimes be ommitted for checking – particularly in the case of the ($\mathcal{ID}_{ag}f$ properties where you are interested in whether the agent ever has an intention that contains a particular action, the action may well have been removed from the intention before the property is checked).

By default, in fact, AJPF generates a new model state every time the agent advances its reasoning cycle. `TwoPickUpAgents_EveryTransition.jpf` runs the above examples in this mode. If you run it you will notice that there are a lot more states in the model and that most of them are generated by a `NewAgentProgramState` choice generator that doesn't actually cause any branching. This behaviour can be switched off by including `ajpf.transition_every_reasoning_cycle = false` in the AIL configuration file (Note this has to be the *AIL configuration file* not the *JPF configuration file*).

5.3 Tutorial 3 – Using AJPF to create Models for other Model-Checkers

This is the third a series of tutorials on the use of the AJPF model checking program. This tutorial covers the use of AJPF in conjunction with other model-checkers, specifically SPIN and Prism. AJPF is used to create a model of the program which is then verified by another tool. The main purpose of this is to enable model checking with more expressive logics (as can be done with the Prism implementation), but there may also be efficiency gains in outsourcing property checking to another tool.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/ajpf_tutorials/tutorial3`.

This tutorial assumes familiarity with the operation of AJPF as described in section 5.1 and in section 5.2 and familiarity with the theory of model checking. Unlike most tutorials, this tutorial is not standalone and assumes the user has access to both SPIN and Prism.

This tutorial explains how to use the tools described in [Dennis et al., 2015b].

5.3.1 Separating out Model and Property

In normal operation, the AJPF system is performing two tasks at once. Firstly it is building a *model* of the program execution. This is a graph (or Kripke structure) of states. These are numbered and labelled with the facts that are true in each state (e.g., "agent 1 believes `holding_block`", "agent 2 has a goal `pickup_block`" and so on). At the same time it is checking this graph against a property (e.g., "eventually agent 2 believes `holding_block`"). It does this by converting the property into an automaton, combining the property automaton with the Kripke structure on-the-fly (following [Gerth et al., 1996, Courcoubetis et al., 1992]) and then checking for accepting paths through this product automaton.

In this tutorial we demonstrate how AJPF can be used to produce just the Kripke structure without creating the property automaton or the product automaton.

5.3.2 Using AJPF with SPIN

SPIN [Holzmann, 2004] is a popular model checking tool originally developed by Bell Laboratories in the 1980s. It has been in continuous development for over thirty years and is widely used in both industry and academia (e.g., [Havelund et al., 2000, Kars, 1996, Kirsch et al., 2011]). SPIN uses an input language called Promela. Typically a model of a program and the property (as a "never claim" — an automaton describing executions that violate the property) are both provided in Promela, but SPIN also provides tools to convert formulae written in LTL into never claims for use with the model-checker. SPIN works by automatically generating programs written in C which carry out the exploration of the model relative to an LTL property. SPIN's use of compiled C code makes it very quick in terms of execution time, and this is further enhanced through other techniques such as partial order reduction. The examples in this tutorial were checked using SPIN version 6.2.3 (24 October 2012).

To complete this tutorial you will need to download, install and run SPIN. SPIN can be downloaded from *http://spinroot.com* where you can also find documentation in its use.

Why use SPIN

SPIN and AJPF are both LTL model-checkers so it may seem odd to use AJPF only to produce the model and then use SPIN to create the property automaton. There are a couple of advantages to this however. Firstly SPIN has more powerful tools for producing property automata and so there are some properties that AJPF can not handle which SPIN can. Secondly SPIN's LTL model checking algorithms are more efficient than AJPF's so in theory the whole process could be quicker by using SPIN. In practice it has been demonstrated [Dennis et al., 2015b] that the major cause of slow performance in AJPF is in generating the Kripke structure of the program model and any gains in efficiency from using SPIN are often lost in converting AJPF's program model into Promela. However there may nevertheless be situations where efficiency gains can be made.

In terms of this tutorial, looking at the process of exporting models to SPIN forms a useful preliminary first step before we turn our attention to Prism.

Configuring AJPF to output SPIN models

In order to configure AJPF to use another model checker you need to tell it:

1. to produce *only* a program model,
2. which other model checker to target and,
3. where to output the program model.

This is done in the AJPF configuration file.

We will use the same program that was used in section 5.2. In the directory for tutorial 3 you will find the configuration file `TwoPickUpAgents_Spin.jpf` that is shown in figure 5.2.

This configuration file tells AJPF to produce a model only (`ajpf.model_only = true`), to target the SPIN model checker (`ajpf.target_modelchecker = spin`), and to print the model to standard out (`ajpf.model.location = stdout`). If you execute it in AJPF you get a print out of the model after AJPF has finished executing. This start of this print out is shown in figure 5.3. In this, the first state (state 0) in the AJPF model has become `state0` in the Promela model. This state can transition to either state 1 (`state1`) or state 37 (`state77`) and so on.

This model also records one proposition `bag1holdblock` which is true in both states 0 and 1. If we look further into the model (shown in figure 5.4) we see that `bag1holdblock` is false in states 8 and end_state9 which is an end state in the model.

The property The property is, in fact, the AJPF property $\mathcal{B}_{ag1}\, hold(block))$ and this has been stated in the property specification language file as property 1.

```
@using = mcapl

target = ail.util.AJPF_w_AIL
target.args = ${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial2/TwoPickUpAgents.ail,
${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial3/PickUpAgent.psl,1

ajpf.model_only = true
ajpf.target_modelchecker = spin
ajpf.model.location = stdout

listener+=,.listener.ExecTracker
et.print_insn=false
et.show_shared=false
```

Figure 5.2: A Configuration file for use with Spin

When using AJPF to generate only a program model, the property in the property specification language file should be a conjunction of the atomic properties that will appear in the property checked by the external system. In the property specification language, the atomic properties are those about the mental state of the agent, or the perceptions in the environment, i.e., those of the form $\mathcal{B}_{ag}\, f$, $\mathcal{G}_{ag} f$, $\mathcal{A}_{ag} f$, $\mathcal{I}_{ag} f$, $\mathcal{ID}_{ag} f$ and $\mathcal{P}(ag) f$.

Printing the Output to a file You can obviously cut and paste the Promela model from the AJPF output into a file for use with SPIN. Alternatively you can set `ajpf.model.location` to the path to an output file. The path should be relative to your `HOME` directory.

If you want to give the absolute file name you need to set `ajpf.model.path` as well as `ajpf.model.location` in the configuration file. The system will then join these to create the absolute path to the file you want to use.

The file `TwoPickUpAgents_SpinToFile.jpf` will print the model to a file `tutorial3_spin.pml` in the tutorial directory.

Model-checking the program in SPIN In the tutorial directory you will find a file `spinprop.pml` which is a Promela file containing the *never claim* for $\neg \Diamond \mathcal{B}_{ag1}\, hold(block)$. SPIN searches for a contradiction, so the model checking succeeds if it can find no path through the model where $\mathcal{B}_{ag1}\, hold(block)$ does not eventually hold.

You can take the file containing your program model, plus `spinprop.pml` and compile them (using `spin -a -N spinprop.pml` *modelfile*) to get a C file, `pan.c`. This needs to be compiled then executed in order to check the program. More details on this process can be found in the SPIN documentation.

```
bool bag1holdblock

active proctype JPFModel()
{
state0:
 bag1holdblock = true;
 if
 :: goto state1;
 :: goto state37;
 fi;
state1:
 bag1holdblock = true;
 if
 :: goto state2;
 :: goto state24;
 fi;
```

Figure 5.3: Model output for SPIN

```
state8:
 bag1holdblock = false;
 goto end_state3;
end_state9:
 bag1holdblock = false;
 printf("end state\n");
```

Figure 5.4: Further states in the SPIN model

Exercise

You will find a second file in the tutorial directory, `spinprop2.pml`, which contains a never claim for the property $\neg(\diamond\mathcal{B}_{ag1}\, hold(block) \wedge \diamond\neg\mathcal{B}_{ag2}\, hold(block))$.

In order to verify this property you will need to adapt the property in `PickUpAgent.psl` so that it contains a conjunction of $\mathcal{B}_{ag1}\, hold(block)$ and $\mathcal{B}_{ag2}\, hold(flag)$ and then regenerate the model and check in SPIN. As usual a solution file can be found in the answers directory.

5.3.3 Using AJPF with Prism

Prism [Kwiatkowska et al., 2011] is a probabilistic symbolic model-checker in continuous development, primarily at the Universities of Birmingham and Oxford, since 1999. Prism provides broadly similar functionality to SPIN but also allows for the model-checking of probabilistic models, i.e., models whose

behaviour can vary depending on probabilities represented in the model. Developers can use PRISM to create a probabilistic model (written in the PRISM language) which can then be model-checked using PRISM's own probabilistic property specification language, which subsumes several well-known probabilistic logics including PCTL, probabilistic LTL, CTL, and PCTL*. PRISM has been used to formally verify a variety of systems in which reliability and randomness play a role, including communication protocols, cryptographic protocols and biological systems. The examples in this tutorial were checked using PRISM version `4.3`.

To complete this tutorial you will need to download, install and run PRISM. PRISM can be downloaded from *http://www.prismmodelchecker.org* where you can also find documentation on its use.

Configuring AJPF to output Prism models

As mentioned in section 5.3.2, in order to configure AJPF to use another model checker you need to tell it:

1. to produce *only* a program model,
2. which other model checker to target and,
3. where to output the program model.

Because PRISM also includes probabilistic information in the model, when using AJPF with PRISM it is also important to use a *listener* that records such information when a choice in the java execution is governed by a probability.

This is done in the AJPF configuration file. Initially we will, once again, use the same program that was used in section 5.2. In the directory for tutorial 3 you will find the configuration file `TwoPickUpAgents_Prism.jpf` that is shown in figure 5.5.

This configuration file tells AJPF to produce a model only (`ajpf.model_only = true`), to target the PRISM model checker (`ajpf.target_modelchecker = prism`), to print the model to standard out (`ajpf.model.location = stdout`), and to use a probability listener (`listener=ajpf.MCAPLProbListener`). If you execute it in AJPF you get a print out of the model after AJPF has finished executing. The start of this print out is shown in figure 5.6

There is no specifically probabilistic behaviour in this example, however there are two agents and the system, by default, assumes each agent has an equal chance of running every time the scheduler makes a decision. We can see here, therefore, that in state 0 there is a 50% change that the system will transition to state 1 and a 50% chance that it will transition to state 37. As with the SPIN example we are interested in one property, `bag1holdblock` ($\mathcal{B}_{ag1}\, hold(block)$)) and this is true in all the initial states of the model but is false after state 5.

The property Just as when using AJPF with SPIN, the property in the AJPF property specification file should be a conjunction of the atomic properties

```
@using = mcapl

target = ail.util.AJPF_w_AIL
target.args = ${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial2/TwoPickUpAgents.ail,
${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial3/PickUpAgent.psl,1

ajpf.model_only = true
ajpf.target_modelchecker = prism
ajpf.model.location = stdout

listener=ajpf.MCAPLProbListener

listener+=,.listener.ExecTracker
et.print_insn=false
et.show_shared=false
```

Figure 5.5: A Configuration file for PRISM

that will be used in the final property to be checked. In the property specification language, the atomic properties are those about the mental state of the agent, or the perceptions in the environment, i.e., those of the form $\mathcal{B}_{ag}\, f$, $\mathcal{G}_{ag} f$, $\mathcal{A}_{ag} f$, $\mathcal{I}_{ag} f$, $\mathcal{ID}_{ag} f$ and $\mathcal{P}(ag) f$.

Printing the Output to a file You can obviously cut and paste the PRISM model from the AJPF output into a file for use with PRISM. Alternatively you can set `ajpf.model.location` to the path to an output file. The path should be relative to your `HOME` directory.

If you want to give the absolute file name you need to set `ajpf.model.path` as well as `ajpf.model.location` in the configuration file. The system will then join these to create the absolute path to the file you want to use.

The file `TwoPickUpAgents_PrismToFile.jpf` will print the model to a file `tutorial3_prism.pm` in the tutorial directory.

Model-checking the program in Prism In the tutorial directory you will find a file `prismprop1.pctl` which is a PRISM file containing the PCTL property for $P^{=?}\Box \diamond \mathcal{B}_{ag1}\, hold(block)$.

You can take the file containing your program model, plus `prismprop1.pctl` and run them in PRISM (using `prism` *model file* `prismprop1.pctl`). This property is actually false and you should get a result of 0 probability:

`Result: 0.0 (value in the initial state)`.

More details on this process can be found in the PRISM documentation.

```
dtmc

 module jpfModel
state : [0 ..89] init 0;
bag1holdblock: bool init true;
[] state = 0 -> 0.5:(state'=1) & (bag1holdblock'= true) + 0.5:(state'=37) & (bag1holdblock'= true);
[] state = 1 -> 0.5:(state'=2) & (bag1holdblock'= true) + 0.5:(state'=24) & (bag1holdblock'= true);
[] state = 2 -> 0.5:(state'=3) & (bag1holdblock'= true) + 0.5:(state'=15) & (bag1holdblock'= true);
[] state = 3 -> 0.5:(state'=4) & (bag1holdblock'= true) + 0.5:(state'=10) & (bag1holdblock'= true);
[] state = 4 -> 0.5:(state'=5) & (bag1holdblock'= true) + 0.5:(state'=9) & (bag1holdblock'= true);
[] state = 5 -> 0.5:(state'=6) & (bag1holdblock'= false) + 0.5:(state'=8) & (bag1holdblock'= false);
[] state = 6 -> 1.0:(state'=89) & (bag1holdblock'= false);
[] state = 8 -> 1.0:(state'=88) & (bag1holdblock'= false);
```

Figure 5.6: A Model for PRISM

5.3.4 Model Checking Agent Systems with Probabilistic Behaviour

We will now look at a program with probabilistic behaviour. This program is a modified version of one used in section 4.3. The program consists of a robot, `searcher.gwen`, which searches a 3x3 grid in order to find a human and an environment, `RandomRobotEnv`, in which a human is moving between the squares and could be at (0, 1), (1, 1) or (2, 1) with a 50% chance of being at (1, 1), a 30% chance of being at (2, 1) and a 20% chance of being at (0, 1). The robot only finds the human if it is in the same square as the robot and it immediately leaves the area once it finds the human (if it has checked every square without finding the human then it checks every square again) therefore there is a chance that the robot will never check the last square (2, 2).

In the directory for tutorial 3 you will find the AJPF configuration file `searcher.jpf` for this program, that is shown in figure 5.7.

This file target's the PRISM model checker and prints the model to standard out. It uses the listener `ajpf.MCAPLProbListener` to record probabilistic information as the model is built. If you execute it in AJPF you get a print out of the model after AJPF has finished executing. The start of this is shown in figure 5.8.

In this state 0 can transition to three states representing the movement of the human: state 1 (probability 0.5), state 72 (with probability 0.3) and state 73 (with probability 0.2) and so on.

This model also records one proposition `bsearcherempty2020` which is false in all the early states but if you look further into the model you will see it becomes true when state 10 transitions to state 11.

```
@using = mcapl

target = ail.util.AJPF_w_AIL
target.args = ${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial3/searcher.ail,
${mcapl}/src/examples/gwendolen/ajpf_tutorials/tutorial3/searcher.psl,1

log.info = ail.mas.DefaultEnvironment,ajpf.product.Product

ajpf.model.location = stdout
ajpf.model_only = true
ajpf.target_modelchecker = prism

listener=ajpf.MCAPLProbListener

listener+=,.listener.ExecTracker
et.print_insn=false
et.show_shared=false
```

Figure 5.7: Configuration File for the searcher program

```
dtmc

 module jpfModel
state : [0 ..80] init 0;
bsearcherempty2020: bool init false;
[] state = 0 -> 0.5:(state'=1) & (bsearcherempty2020'= false) + 0.3:(state'=72) & (bsearcherempt
[] state = 1 -> 0.5:(state'=2) & (bsearcherempty2020'= false) + 0.3:(state'=70) & (bsearcherempt
[] state = 2 -> 0.5:(state'=3) & (bsearcherempty2020'= false) + 0.3:(state'=68) & (bsearcherempt
[] state = 3 -> 0.3:(state'=66) & (bsearcherempty2020'= false) + 0.2:(state'=67) & (bsearcheremp
[] state = 4 -> 0.2:(state'=65) & (bsearcherempty2020'= false) + 0.5:(state'=5) & (bsearcherempt
[] state = 5 -> 1.0:(state'=80) & (bsearcherempty2020'= false);
```

Figure 5.8: PRISM model for the searcher program

The property The property is the AJPF property $\mathcal{B}_{searcher}\, empty(2,2)$ and this has been stated in the property specification language file as property 1.

Model-checking the program in Prism In the tutorial directory you will find a file `prismprop2.pctl` which is a PRISM file containing the PCTL property for $P^{=?} \diamond \mathcal{B}_{searcher}\, empty(2,2)$. If you run your PRISM model with this file you should find that the property has a 35% chance of being true – i.e., the robot has roughly a 35% chance of checking the final square.

In the property specification file there is a second property for $\mathcal{B}_{searcher}\, found$ (that the searcher has found the human). If you generate a model for this property and check it in PRISM you will find its probability is 1, even though there is an infinite loop where the robot never finds the human. However the probability that the robot will remain in this infinite loop forever is infinitesimally small.

A Note on Creating Environments with Probabilistic Behaviour

In order for AJPF's probability listener to work correctly, all randomness (and probabilistic behaviour) should be created using AIL's `Choice` classes as documented in section 4.3.

It is important that probabilistic choices cause *unique* transitions in the model. If, for instance, you generate four choices each with, say, a 25% probability but two of them end up leading to the same next state then AJPF will only annotate the transition with one of the probabilities (not the sum of both) and this will lead to PRISM generating an error. For instance say you have four choices each representing a direction some human could move in, north, east, south or west. If you are working in a grid world and if the selected direction were to take the human off the grid then you might choose to have the human remain in the same place instead. In this situation, when the human is in the corner of the grid, two of those choices will lead to the same result (the human remains in place) however only the probability for one of these occurring will be annotated on the transition and PRISM will warn that the probabilities of the transitions in this state do not sum to 1.

Exercise

You will find, in the tutorial directory an AIL program, `pickuprubble.ail`. This controls a robot (called `robot`) that searches a small 2x3 grid for injured humans. There is one human in the grid who moves around it randomly and one building in the grid that may collapse. If the building collapses onto the human then they will be injured. The robot systematically searches the grid. If it encounters the human it will direct them to safety and if it finds them injured it will assist them. However the once the robot has reached the top corner of the grid it will stop searching. There is therefore, a chance that the robot will never encounter the human and, what is more, that the human will visit the building after the robot has checked and will be injured by the building collapsing.

We are interested therefore in discovering the probability that if a human is injured then, eventually they are assisted by the robot. The file `prismprop_ex.pctl` in the tutorial directory contains the property $P^{=?}\Box(\mathcal{P}(injured\_humans) \Rightarrow \diamond\mathcal{A}_{robot}assist\_humans)$.

Create an AJPF configuration file and property specification file that will generate a PRISM model for this program and property. You should be able to discover that there is an 88% chance of the robot assisting any injured human. Note that the AJPF model build will take several minutes to run (it generates 3,546 states).

As usual solution files can be found in the answers directory.

5.4 Tutorial 4 – Verification Environments

This is the fourth in a series of tutorials on the use of the AJPF model checking program. This tutorial covers the creation of Java Environments specifically for use in the verification of Autonomous Systems – particularly systems that intended to run in some environment, such as the real world, that is not written in Java.

Files for this tutorial can be found in the `mcapl` distribution in the directory

```
src/examples/verifiableautonomoussystems/chapter5
```

The motorway simulator can be found in `src/examples/motorwaysim`

This tutorials assumes some familiarity GWENDOLEN programming and the creation of environments for multi-agent systems as described in section 4.3

5.4.1 Where does the Automaton representing a BDI Agent Program Branch?

A key part of model-checking is the full exploration of the state space of a program (or model). It's value is therefore in situations where there are branching points in the possible execution of a program. A program which simply prints out the numbers from 1 to 10, for instance, needs only to be tested once to see if it actually does this since there is only one possible execution of the program. In general BDI agent programming languages do not implement any randomness within the language itself so branching in the execution of a program generally occurs at two points.

Firstly, in a multi-agent system, individual agents may act in different orders. Consider two agents, a_1 and a_2 each with a simple program which means that a_1 does act_1 and a_2 does act_2. Then there is potentially a run of this system in which act_1 happens before act_2 and another in which act_2 happens before act_1. The AIL-toolkit provides support for different scheduling policies among agents as discussed in section 4.3. These scheduling policies govern which agent gets to make a state transition at any one time and can, for instance, enforce strict turn taking among agents or, alternatively, select the next agent to make a transition entirely at random. Depending upon the policy used then there

may be branching points created in the automata checked by AJPF. At present no language in AJPF allows two agents to make a transition at exactly the same time, but this is not in principle excluded.

The second place in which branching may occur is in the information received by the agent from perception or messages. Sometimes this information is generated by other agents in the system and so branching points are caused by the scheduling policy which dictates when agents perform actions or send messages. However we may also wish to represent non-determinism in the environment within which the agents operate – for instance we might want to introduce the possibility that messages get lost. In that case we can use randomness (as discussed in section 4.3) when we program our JAVA environment to create such branching. The AIL-toolkit provides specific support for this randomness in order both to assist the model-checking process and to allow replay of specific paths through a program execution if a bug is detected.

5.4.2 The Problem with Environments

This desire to represent non-deterministic behaviour in the agents' environment leads us to one of the key features of our approach to model-checking autonomous systems. When we model check an agent in AJPF (or indeed any model-checking system) we *have to* model check it in the context of a purely JAVA environment that we have placed it in. However the reason we may be representing non-determinism in that environment (e.g., message loss) is because we believe that in the 'real' environment in which it will actually be deployed different things may occur and we wish to understand the effect of this on the system behaviour.

So when model-checking an autonomous hybrid agent system in AJPF we have to construct a JAVA environment that represents a simulation of some 'real' world. We can encode assumptions about the behaviour of the 'real' world in this simulation, but we would prefer to minimize such assumptions. For much of our autonomous systems work we try to have minimal assumptions where the environment asserts or retracts percepts and messages on an entirely random basis. By this we mean that we do not attempt to model assumptions about the effects an agent's actions may have on the world, or assumptions about the sequence in which perceptions may appear to the agent. This approach is not without its cost in terms of state space and the efficiency of model-checking. As a result we often do have to build in assumptions about the real world. An approach to mitigating the potential issues introduced by making assumptions is discussed in [Ferrando et al., 2021].

The process for verifying an agent in this way, is to first analyse the agent program in order to identify all the perceptions that have an effect on the program. In multi-agent systems it is also necessary to identify all messages that the agent may receive from other agents in the environment. Once a list of perceptions and messages has been identified, an environment is constructed for the agent alone in such a way that every time the agent takes an action the set of perceptions and messages available to it are created *at random*. When

model checking, the random selection causes the search tree to branch and the model checker to explore all environmental possibilities.

We will illustrate this process with a simple, but hopefully instructive, example. There is a fuller discussion of the methodology in [Dennis et al., 2014].

5.4.3 Example: Cars on a Motorway

We explain our approach to model-checking autonomous systems via an example of two cars on a motorway[3]

Example 7 *We will consider an intelligent cruise control for a car, focusing simply on when to accelerate and when to maintain its speed. The* GWENDOLEN *code for this is show below. There are two cars,* car1 *and* car2 *and, in both cases, when the car has a goal to reach the speed limit, +!* at_speed_limit [achieve]*, it accelerates and then waits until the goal is achieved (The '*' symbol is the* GWENDOLEN *syntax for 'waiting'). The first car then also sends a message to car2. Once the cars have reached the speed limit they perform a* **maintain_speed** *action followed by a* **finished** *action. Car1 gets the goal to be at the speed limit when it perceives that it has started, while Car2 gets the goal only when it receives a message to achieve the goal.*

```
:name: car1                                                          1
                                                                     2
:Initial Beliefs:                                                    3
                                                                     4
:Initial Goals:                                                      5
                                                                     6
:Plans:                                                              7
+started: {True} ← +!at_speed_limit[achieve];                        8
                                                                     9
+! at_speed_limit [achieve] : {True} ←                              10
        accelerate,                                                 11
        *at_speed_limit,                                            12
        .send(car2, :achieve, at_speed_limit);                      13
                                                                    14
+at_speed_limit: {True} ←                                           15
        maintain_speed,                                             16
        finished;                                                   17
                                                                    18
:name: car2                                                         19
                                                                    20
:Initial Beliefs:                                                   21
                                                                    22
:Initial Goals:                                                     23
                                                                    24
:Plans:                                                             25
```

[3]Examples from this tutorial can be found in `src/examples/verifiableautonomoussystems/chapter5` within the MCAPL distribution. The motorway simulator can be found in `src/examples/motorwaysim`.

```
+.received(:achieve, G): {True} ← +!G [achieve];                    26
                                                                    27
+! at_speed_limit [achieve] : {True} ←                              28
        accelerate,                                                 29
        *at_speed_limit;                                            30
                                                                    31
+at_speed_limit: {True} ←                                           32
        maintain_speed,                                             33
        finished;                                                   34
```

Executing the Program

In order to execute the above program, it needs to be connected either to physical vehicles or simulations. Figure 8.2 shows the output in a very simple vehicle simulator. The simulator has two cars each in their own motorway lane. The lanes loop around so when one car reaches the end of its lane it loops back to the start. The simulator reports both the speed of each car and their distance from the start of the motorway.

The agents are connected to the simulator via a JAVA environment which communicates using a standard socket mechanism. It reads the speeds of the cars from the sockets and publishes values for required acceleration to the socket. If a car's speed becomes larger than 5 then the environment adds a perception that the car is at the speed limit. If a car agent performs the accelerate action then the environment publishes an acceleration of 0.1 to the socket. If a car agent performs the maintain_speed action then the environment publishes an acceleration of 0 to the socket.

Verification: Building a model by Examining the Environment

Suppose we wish to use AJPF to verify our agents for controlling the two cars. We cannot include the whole of the motorway simulator program in our formal verification since it is an external program. We need to replace the socket calls to this simulator in our JAVA environment with some model of its behaviour.

A naive way to set about this might be to capture the obvious behaviour of the simulator. We could use a scheduler to alternately execute a method, generally called do_job in AIL-supporting environments, in the simulator to calculate the position of each car and then to execute one step in the reasoning cycle of each car. If a car agent executes accelerate then each call of do_job increases the car's speed by 1. If the car agent executes maintain_speed then the car's speed remains constant. As in the JAVA environment that communicated with the simulator, once the speed has reached 5 this is set as a percept, at_speed_limit, that the agent can receive.

Such a model is, in fact, entirely deterministic (like the program that counted to 10 in Section 5.4.1) because of the turn based control of the environment and

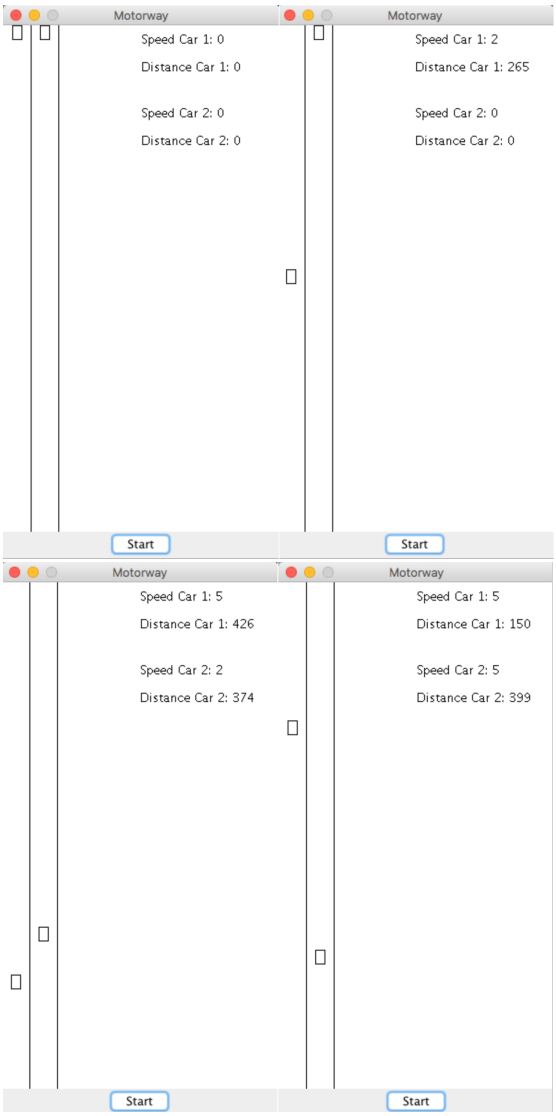

Figure 5.9: Simulating two cars on a motorway. Images from left to right show: (a) two cars waiting at the start; (b) the first car accelerating; (c) as the first car reaches a speed of 5 it messages the second car which begins accelerating; until (d) both cars are moving at a speed of 5.

the two agents. There is no need for more sophisticated model checking since we can simply run the program once and see what happens.

Obviously we can make our model more complex – for instance we could introduce a random element into whether the car detects that it has reached the speed limit, or exactly how much acceleration is created. There are some limitations to this, however. For instance, we need our model to contain a reasonable number of states, so we can not simply vary the acceleration by a random double since that would introduce a very large search space, creating a search branch for each possible double value that could be used at that point.

Similarly as the world we wish to model becomes more complicated, such environments inevitably become harder and harder to craft in ways that behave with appropriate fidelity.

Verification: Building a Model by Examining the Agents

The alternative to trying to create a JAVA model to accurately describe the behaviour of the real world is to analyse instead the inputs in terms of perceptions and messages received by the agent program. This is our approach. We construct a model in which, every time the agent program queries the environment for perceptions, the environment returns a random subset of these. In the case of car1 there are only two perceptions at_speed_limit and started and so we need an environment that generates inputs from these two.

The MCAPL framework provides support for creating these kinds of environments for GWENDOLEN programs through an abstract

`gwendolen.mas.VerificationofAutonomousSystemsEnvironment`

class that can, in turn, be sub-classed. The sub-classes simply have to sub-class the methods for generating random perceptions, generate_percepts, and random messages, generate_messages.

Example 8

```
public Set<Predicate> generate_percepts()                    1
{                                                            2
   Set<Predicate> beliefs = new HashSet<Predicate>();        3
                                                             4
   boolean at_speed_limit =                                  5
      random_bool_generator.nextBoolean();                   6
   boolean started =                                         7
      random_bool_generator.nextBoolean();                   8
   if (at_speed_limit)                                       9
   {                                                         10
      beliefs.add(new Predicate("at_speed_limit"));          11
      AJPFLogger.info(logname, "At Speed Limit");            12
   }                                                         13
                                                             14
   if (started)                                              15
```

```
    {                                                                    16
        beliefs.add(new Predicate("started"));                           17
        AJPFLogger.info(logname, "Started");                             18
    }                                                                    19
    return beliefs;                                                      20
}                                                                        21
```

Here, we show the generate_percepts *method. Two booleans are generated at random and are used to decide whether or not a percept is added to the set returned to the agent. For our car example, the two percepts in question are* at_speed_limit *and* started*. A similar mechanism can be used to generate messages at random. A logging mechanism* `AJPFlogger` *prints output about the perceptions generated for a user to see.*

Using this environment many simple properties are false, for instance *"if car1 accelerates, eventually it will be at the speed limit"* is false. This is because the environment does not link the acceleration action in any way to the car's speed. In fact the actions taken by the agent in such an environment have no causal link to the perceptions that are returned. Essentially, we cannot make any assumptions about whether the software and machinery involved in making acceleration happen are working, nor whether the sensors for detecting the car's speed are working.

To prove useful properties with this kind of environment we typically prove properties of the general form

"If whenever the agent does X eventually the agent believes Y then...".

So for instance we can prove, using the above environment, that "provided that, if car1 invokes acceleration then eventually car1 believes it is at the speed limit, then eventually car1 will invoke finished", i.e:

$$\Box(\mathcal{A}_{car1} accelerate \rightarrow \diamond \mathcal{B}_{car1}\, at\_speed\_limilt) \rightarrow \diamond \mathcal{A}_{car1} finished.$$

There are many properties of this form discussed in the examples in the MCAPL distribution.

Just as we could make our model based on examination of the environment more complex and so increase the state space, we can reduce the state space for models based on examination of the agents by linking the generation of percepts to actions. By default `VerificationofAutonomousSystemsEnvironment` only randomly generates new sets of perception after an action has been invoked – any other time the agent polls the environment for perceptions it receives the same set it was sent last time it asked. While this does introduce assumptions about the behaviour of the real world – that changes in perceptions only occur after an agent has taken some action, it is normally comparatively safe if you can assume that agent deliberation is very fast compared to the time it takes to execute an action and for changes in the world to occur. This reduces the possibilities and the complexity of the model-checking problem.

It is also possible to make application specific assumptions to constrain the generation of sets by the environment: for instance that the at_speed_limit perception can not be included in a set until after the accelerate action has been performed at least once. This does increase the risk that the environment used for verification may exclude behaviours that would be observed in the real environment.

Part II

Agent Programming Languages

Chapter 6

The GWENDOLEN Programming Language

This chapter contains a set of tutorials on the GWENDOLEN programming language. The semantics of the language can be found in [Dennis, 2017]. These tutorials give an introduction to the use of the language.

6.1 Tutorial 1 — Introduction to Running Gwendolen Programs

This is the first in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers the basics of running GWENDOLEN programs, the configuration files, perform goals and print actions. It assumes the reader is familiar with the basics of Prolog notation. Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial1`.

The tutorials assume some familiarity with the Prolog programming language as well as the basics of running Java programs either at the command line or in Eclipse.

6.1.1 Hello World

You will find a GWENDOLEN program in the tutorial directory called `hello_world.gwen`. It's contents should look like Example 9.

This can be understood as follows. Line 1 states the language in which the program is written (this is because the AIL allows us to create multi-agent systems from programs written in several different languages). Line 3 gives the name of the agent (hello). Line 5 starts the section for initial beliefs (there are none). Line 7 starts the section for initial goals. There is one a *perform* goal

Example 9

```
GWENDOLEN                                                    1
                                                             2
:name: hello                                                 3
                                                             4
:Initial Beliefs:                                            5
                                                             6
:Initial Goals:                                              7
                                                             8
say_hello [perform]                                          9
                                                             10
:Plans:                                                      11
                                                             12
+!say_hello [perform] : {True} ← print(hello);               13
```

to say_hello (we will cover the different sorts of goal in a later tutorial). Line 11 starts the section for plans. There is one plan which can be understood as saying if the goal is to say hello +!say_hello then do the action print(hello). There is a third component to the plan ({True}) which is a *guard* that must be true before the plan is applied. In this case the guard is always true so the plan applies whenever the agent has a goal to perform +!say_hello.

Running the Program

To run the program you need to run the JAVA program `ail.mas.AIL` and supply it with a suitable configuration file as an argument. You will find an appropriate configuration file, `hello_world.ail` in the same directory as `hello_world.gwen`. You can do this either from the command line or using the IntelliJ or Eclipse `run-AIL` configuration (with `hello_world.ail` selected in the Project Files/Package Explorer window) as detailed in chapter 3.

Run the program now.

6.1.2 The Configuration File

Now open the configuration file, `hello_world.ail` shown in figure 6.1.

This is a very simple configuration consisting of four items only.

mas.file gives the path to the GWENDOLEN program to be run.

mas.builder gives a java class for building the file. In this case `gwendolen.GwendolenMASBuilder` parses a file containing one or more GWENDOLEN agents and compiles them into a multi-agent system.

env provides an environment for the agent to run in. In this case we use the default environment provided by the AIL.

```
mas.file = /src/examples/gwendolen/tutorials/tutorial1/hello_world.gwen
mas.builder = gwendolen.GwendolenMASBuilder

env = ail.mas.DefaultEnvironment

log.warning = ail.mas.DefaultEnvironment
```

Figure 6.1: A Simple Configuration File

log.warning sets the level of output for the class `ail.mas.DefaultEnvironment`. This is a pretty minimal level of output (warnings only). We will see in later tutorials that it is often useful to get more output than this.

6.1.3 Some Simple Exercise to Try

1. Change the filename of `hello_world.gwen` to something else (e.g., `hello.gwen`). Update `hello_world.ail` to reflect this change. Check you can still run the program.
2. Edit the hello world program so it prints out `hi` instead of `hello`.
3. Edit the hello world program so the goal is called `hello` instead of `say_hello`. If you don't change the plan notice how the behaviour of the program changes. Edit the plan to return to the original behaviour of the program.
4. Change the plan to

   ```
   +!say_hello [perform] : {True} <- print(hello), print(louise);
   ```

   and see how this changes the behaviour of the program.
5. Experiment getting the program to print out various different strings. Note that in order to print a string containing whitespace, the string must be contained in double quotes (i.e. `print("hello world");`)

6.2 Tutorial 2 — Simple Beliefs, Goals and Actions

This is the second in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers the basics of beliefs, goals and actions as they appear in GWENDOLEN.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial2`.

6.2.1 Pick Up Rubble

You will find a GWENDOLEN program in the tutorial directory called `pickuprubble.gwen`. It's contents should look like Example 10.

Example 10

```
GWENDOLEN                                                          1
                                                                   2
:name: robot                                                       3
                                                                   4
:Initial Beliefs:                                                  5
                                                                   6
:Initial Goals:                                                    7
                                                                   8
goto55 [perform]                                                   9
                                                                   10
:Plans:                                                            11
                                                                   12
+!goto55 [perform] : {True} ← move_to(5, 5);                       13
                                                                   14
+rubble(5, 5): {True} ← lift_rubble;                               15
                                                                   16
+holding(rubble): {True} ← print(done);                            17
```

This is a program for moving around a simple grid based environment and picking up rubble. The robot can perform three actions in this environment,

move_to(X, Y) moves to grid square (X, Y) and adds the belief at(X, Y).

lift_rubble attempts picks up a piece of rubble and adds the belief holding(rubble) if there is rubble at the robot's location.

drop drops whatever the robot is holding and removes any beliefs about what the robot is holding.

The default actions (e.g., print) are also available to the robot. As the environment is set up there is a block of rubble at square (5, 5) which the robot will see if is in square (5, 5). When the robot picks something up it can see that it is holding it. This environment is programmed in JAVA and is the class gwendolen.tutorials.SearchAndRescueEnv.

The program can be understood as follows.

Line 1 states the language in which the program is written (this is because the AIL allows us to create multi-agent systems from programs written in several different languages).

Line 3 gives the name of the agent (robot).

Line 5 starts the section for initial beliefs (there are none).

Line 7 starts the section for initial goals. There is one a *perform* goal to goto55.

Line 11 starts the section for plans. There are three plans. The first (line 13) states that in order to perform goto55 the agent must move to square (5, 5). The second (line 15) states that if the agent sees rubble at 5, 5 it should lift the rubble and the third (line 17) states that if the agent sees it is holding rubble then it should print done.

There are three different sorts of syntax being used here to distinguish between beliefs, goals and actions.

Beliefs Beliefs are predicates (e.g., rubble(5, 5) that are preceded either by a + symbol (to indicate a adding a belief) or a - symbol (to indicate removing a belief).

Goals Goals are predicates preceded by an exclamation mark (e.g., !goto55) again these are then preceded either by a + symbol (to indicate a adding a goal) or a - symbol (to indicate removing a goal). After the goal predicate there is also a label stating what kind of goal it is. Goals can either be *perform* goals or *achieve* goals. We will discuss the difference between these in a moment.

Actions Are just predicates. Actions are performed externally to the agent and can not be added or removed (they are just done).

Running the Program and Getting more Log output

To run the program you need to call `ail.mas.AIL` and supply it with a suitable configuration file. You will find an appropriate configuration file, `pickuprubble.ail` in the same directory as `pickuprubble.gwen`. You can do this either from the command line or using the IntelliJ or Eclipse `run-AIL` configuration as detailed in the MCAPL manual.

Run the program now.

As in Tutorial 1, all you see is the robot printing the message done once it has finished. However what has happened is that the robot moved to square (5, 5) (because of the perform goal). Once in square (5, 5) it saw the rubble and so lifted it (thanks to the second plan). Once it had lifted the rubble it saw that it was holding it and printed done (thanks to the third plan).

You can get more information about the execution of the program by changing the logging information in the configuration file. Open the configuration file and edit

```
log.warning = ail.mas.DefaultEnvironment
```

to

```
log.info = ail.mas.DefaultEnvironment
```

You will now see logging information printed out about each action the robots takes.

If you add the line

```
log.format = brief
```

to the configuration file you will get the log messages in a briefer form.

6.2.2 Perform and Achieve Goals

pickuprubble_achieve.gwen is a slightly more complex version of the rubble lifting program which introduces some new concepts. It is shown in Example 11 The first changes are in lines 7-9. Here we have a list of initial beliefs. The

Example 11

```
GWENDOLEN                                                                    1
                                                                             2
:name: robot                                                                 3
                                                                             4
:Initial Beliefs:                                                            5
                                                                             6
possible_rubble(1, 1)                                                        7
possible_rubble(3, 3)                                                        8
possible_rubble(5, 5)                                                        9
                                                                             10
:Initial Goals:                                                              11
                                                                             12
holding(rubble) [achieve]                                                    13
                                                                             14
:Plans:                                                                      15
                                                                             16
+!holding(rubble) [achieve] :                                                17
  {B possible_rubble(X, Y), ~B no_rubble(X, Y)} <- move_to(X, Y);            18
                                                                             19
+at(X, Y) : {~B rubble(X, Y)} <- +no_rubble(X, Y);                           20
                                                                             21
+rubble(X, Y): {B at(X, Y)} <- lift_rubble;                                  22
                                                                             23
+holding(rubble): {True} <- print(done);                                     24
```

agent believes there may be rubble in one of three squares (1, 1), (3, 3) and (5, 5). As we know, in the environment, there is only rubble in (5, 5).

The next change is in line 13. Here instead of a *perform* goal, goto55 [perform] there is an *achieve* goal, holding(rubble) [achieve]. The difference between perform goals and achieve goals is as follows:

- When an agent adds a perform goal it searches for a plan for that goal, executes the plan and then drops the goal, it does not check that the plan has succeeded.
- When an agent adds an achieve goal it searches for a plan for that goal, executes the plan and then checks to see if it now has a belief corresponding to the goal. If it has no such beliefs it searches for a plan again, if it does have such a belief then it drops the goal.

In the case of this program the robot will continue executing the plan for holding(rubble) until it actually believes that it is holding some rubble.

On lines 17 and 18, you can see the plan for the goal. This plan no longer has a trivial guard. Instead the plan only applies if the agent believes that there is possible rubble in some square (X, Y) and it does not (the $\sim$ symbol) believe there is no rubble in that square. If it can find such a square then the robot moves to it. The idea is that the robot will check each of the possible rubble squares in turn until it successfully finds and lifts some rubble. Note that we are using capital letters for variables that can be unified against beliefs (like in Prolog).

The plan at line 20 gets the robot to add the belief no_rubble(X, Y) if it is at some square, (X, Y), and it can't see any rubble there. By this means the plan in lines 17 and 18 will be forced to pick a different square next time it executes. Up until now all the plans you have used have simply executed actions in the plan body. This one adds a belief.

The plan at line 22 is similar to the plan in line 15 of Example 10 only in this case we are using variables for the rubble coordinates rather than giving it the coordinates (5, 5).

You can run this program using the pickuprubble_achieve.ail configuration file.

6.2.3 Some Simple Exercise to Try

1. Instead of having the robot print done once it has the rubble, get it to move to square (2, 2) and drop the rubble.

   Hint: you may find you need to add a "housekeeping" belief that the rubble has been moved to prevent the robot immediately picking up the rubble once it has been dropped.

2. Rewrite the program so that instead of starting with an achievement goal holding(rubble), it starts with an achievement goal rubble(2, 2) – i.e., it wants to believe there is rubble in square (2, 2).

   Hint: you may want to reuse the plan for achieve holding(rubble) by setting it up as a subgoal. You can use this by adding the command +!holding(rubble) [achieve] in the body of a plan.

Sample answers for these two exercises can be found in `gwendolen/examples/tutorials/tutorial2/answers`.

6.3 Tutorial 3 — Plan Guards and Reasoning Rules

This is the third in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers the use of Prolog style rules as they appear in GWENDOLEN and also looks at plan guards in a little more detail.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial3`.

6.3.1 Pick Up Rubble (Again)

You will find a GWENDOLEN program in the tutorial directory called `pickuprubble_achieve.gwen`. Its contents should look like Example 12.

Example 12

```
GWENDOLEN

:name: robot

:Initial Beliefs:

possible_rubble(1, 1) possible_rubble(3, 3) possible_rubble(5, 5)

:Reasoning Rules:

square_to_check(X, Y) :- possible_rubble(X, Y), ~no_rubble(X, Y);

:Initial Goals:

holding(rubble) [achieve]

:Plans:

+!holding(rubble) [achieve] : {B square_to_check(X, Y)} <-
    move_to(X, Y);
+at(X, Y) : {~B rubble(X, Y)} <- +no_rubble(X, Y);
+rubble(X, Y): {B at(X, Y)} <- lift_rubble;
+holding(rubble): {True} <- print(done);
```

This is very similar to the second program in section 6.2. However instead of having

```
{B possible_rubble(X, Y), ~B no_rubble(X, Y)}
```

as the guard to the first plan we have B square_to_check(X, Y) as the guard.

We can then reason about whether there is a square to check using the Prolog style rule on line 13. The syntax is very similar to Prolog syntax but there are a few differences. We use the symbol, ∼, to indicate "not". It is possible to use Prolog "cuts" to control backtracking in belief reasoning in GWENDOLEN programs using the standard `!` syntax to indicate the cut. Standard Prolog built-in predicates such as `member`, `var` etc., are not currently available.

6.3.2 Using Prolog Lists

You can use Prolog style list structures in GWENDOLEN programs. Example 13 shows the previous example using lists.

Example 13

```
:name: robot                                                                1
                                                                            2
:Initial Beliefs:                                                           3
                                                                            4
possible_rubble([sq(1, 1), sq(3, 3), sq(5, 5)])                             5
                                                                            6
:Reasoning Rules:                                                           7
                                                                            8
square_to_check(X, Y) :- possible_rubble(L), check_rubble(L, X, Y);         9
                                                                            10
check_rubble([sq(X, Y) | T], X, Y) :- ~no_rubble(X, Y);                     11
check_rubble([sq(X, Y) | T], X1, Y1) :- no_rubble(X, Y),                    12
                      check_rubble(T, X1, Y1);                              13
                                                                            14
:Initial Goals:                                                             15
                                                                            16
holding(rubble) [achieve]                                                   17
                                                                            18
:Plans:                                                                     19
                                                                            20
+!holding(rubble) [achieve] : {B square_to_check(X, Y)} ←                   21
     move_to(X, Y);                                                         22
+at(X, Y) : {~B rubble(X, Y)} ← +no_rubble(X, Y);                           23
+rubble(X, Y): {B at(X, Y)} ← lift_rubble;                                  24
+holding(rubble): {True} ← print(done);                                     25
```

Prolog list structures can also be used in GWENDOLEN plans and a recursive style plan may sometimes provide a more efficient program than the kind of program that relies on the failure of a plan to achieve a goal to re-trigger the plan. Example 14 shows an example of this style of programming where the achieve goal calls a perform goal that recurses through the list of squares one at a time.

Example 14

```
:name: robot                                                                    1
                                                                                2
:Initial Beliefs:                                                               3
                                                                                4
possible_rubble([sq(1, 1), sq(3, 3), sq(5, 5)])                                 5
                                                                                6
:Reasoning Rules:                                                               7
                                                                                8
rubble_in_current :- at(X, Y), rubble(X, Y);                                    9
                                                                                10
:Initial Goals:                                                                 11
                                                                                12
holding(rubble) [achieve]                                                       13
                                                                                14
:Plans:                                                                         15
                                                                                16
+!holding(rubble) [achieve] : {B possible_rubble(L)} ←                          17
    +! check_all_squares(L) [perform];                                          18
                                                                                19
+!check_all_squares([]) [perform] : {True} ← print(done);                       20
+!check_all_squares([sq(X, Y) | T]) : {~B rubble_in_current} ←                 21
    move_to(X, Y),                                                              22
    +!check_all_squares(T) [perform];                                           23
+!check_all_squares([sq(X, Y) | T]) : {B rubble_in_current} ←                  24
    print(done);                                                                25
                                                                                26
+at(X, Y) : {~B rubble(X, Y)} ← +no_rubble(X, Y);                               27
                                                                                28
+rubble(X, Y): {B at(X, Y)} ← lift_rubble;                                      29
```

6.3.3 More Complex Prolog Reasoning – Grouping predicates under a negation

`pickuprubble_grouping.gwen` shows more complex use of Reasoning Rules including some syntax not available in Prolog. This is shown in example 15

In this program the agent's goal is to achieve done. It achieves this either if it is holding rubble (deduced using the code on line 14), or if there is no square it thinks may possibly contain rubble that has no rubble in it (deduced using the code on line 15).

The rule on line 15

```
done :- ~ (possible_rubble(X, Y), ~no_rubble(X, Y));
```

isn't standard Prolog syntax. Here we group the two predicates possible_rubble (X, Y), ~no_rubble(X, Y) (from square_to_check) together using brackets and then negate the whole concept (i.e., there are no squares left to check).

Example 15

```
GWENDOLEN                                                                    1
                                                                             2
:name: robot                                                                 3
                                                                             4
:Initial Beliefs:                                                            5
                                                                             6
possible_rubble(1, 1)                                                        7
possible_rubble(3, 3)                                                        8
possible_rubble(5, 5)                                                        9
                                                                             10
:Reasoning Rules:                                                            11
                                                                             12
square_to_check(X, Y) :- possible_rubble(X, Y), ~no_rubble(X, Y);            13
done :- holding(rubble);                                                     14
done :- ~ (possible_rubble(X, Y), ~no_rubble(X, Y));                         15
                                                                             16
:Initial Goals:                                                              17
                                                                             18
done [achieve]                                                               19
                                                                             20
:Plans:                                                                      21
                                                                             22
+!done [achieve] : {B square_to_check(X, Y)} ← move_to(X, Y);                23
                                                                             24
+at(X, Y) : {~B rubble(X, Y)} ← +no_rubble(X, Y);                            25
                                                                             26
+rubble(X, Y): {B at(X, Y)} ← lift_rubble;                                   27
                                                                             28
+holding(rubble): {True} ← print(done);                                      29
```

Some Simple Exercises to Try

1. Try removing the initial belief that there is possible rubble in square (5, 5). You should find the the program still completes and prints out done.

2. Try replacing the rule on line 15 with one that refers to square_to_check

6.3.4 Using Goals in Plan Guards

`pickuprubble_goal.gwen` shows how reasoning rules can be used to reason about both goals and beliefs. This is shown in example 16. Recall that in the exercises in section 6.2 we had to use a belief to prevent the robot picking up the rubble after it had put it down. Here instead we have added G holding(rubble) [achieve] as a guard to the plan that is activated when the robot sees some rubble. In this case it only picks up the rubble if it has goal to be holding rubble.

Example 16

```
GWENDOLEN                                                                    1
                                                                             2
:name: robot                                                                 3
                                                                             4
:Initial Beliefs:                                                            5
                                                                             6
possible_rubble(1, 1)                                                        7
possible_rubble(3, 3)                                                        8
possible_rubble(5, 5)                                                        9
                                                                             10
:Initial Goals:                                                              11
                                                                             12
rubble(2, 2) [achieve]                                                       13
                                                                             14
:Plans:                                                                      15
                                                                             16
+!rubble(2, 2) [achieve]: {True} ← +! holding(rubble)[achieve],              17
    move_to(2, 2),                                                           18
    drop;                                                                    19
                                                                             20
+!holding(rubble) [achieve] :                                                21
    {B possible_rubble(X, Y), ~B no_rubble(X, Y)} ←                          22
move_to(X, Y);
                                                                             23
+at(X, Y) : {~B rubble(X, Y)} ← +no_rubble(X, Y);                            24
                                                                             25
+rubble(X, Y): {B at(X, Y), G holding(rubble) [achieve]} ←                   26
    lift_rubble;                                                             27
```

6.3.5 Reasoning about Beliefs and Goals

`pickuprubble_goalat.gwen` shows how goals can be used in plan guards. This is shown in example 17. Here the reasoning rule on line 13 is used both in the plan on line 29 in order to reason about whether the robot is holding the rubble at square (2, 2) *and* in the plan on line 31 to deduce that the robot has a goal to get the rubble to square (2, 2), from the the fact that it has a goal to be holding rubble (added on line 21) *and* a goal to be at (2, 2) – the initial goal.

Note that we can't use reasoning rules to break down a goal into subgoals. So if you gave the robot the initial goal rubble_at_22 you need to provide a plan specifically for rubble_at_22. It is no good providing a plan for holding rubble and a plan for being at 2, 2 and then expecting the robot to compose these sensibly in order to achieve rubble_at_22.

Try changing the agent's initial goal to rubble_at_22 [achieve] without changing anything else in the program. You should see a warning generated that the agent can not find a plan for the goal. At this point the program

Example 17

```
GWENDOLEN

:name: robot

:Initial Beliefs:

possible_rubble(1, 1)
possible_rubble(3, 3)
possible_rubble(5, 5)

:Reasoning Rules:

rubble_at_22 :- holding(rubble), at(2, 2);

:Initial Goals:

at(2, 2) [achieve]

:Plans:

+!at(X, Y) [achieve]: {True} ← +! holding(rubble)[achieve],
     move_to(X, Y);

+!holding(rubble) [achieve] :
    {B possible_rubble(X, Y), ~B no_rubble(X, Y)} ←
move_to(X, Y);

+at(X, Y) : {~B rubble(X, Y), ~B rubble_at_22} ←
    +no_rubble(X, Y);
+at(X, Y) : {B rubble_at_22} ← drop;

+rubble(X, Y): {B at(X, Y), G rubble_at_22 [achieve]} ←
    lift_rubble;
```

will fail to terminate (when GWENDOLEN can't find a plan for a goal it cycles infinitely looking a plan to handle a failed goal (most programs don't include one of these)). **You will need to terminate the program** (control-C at the command line or clicking the red stop square in Eclipse).

6.3.6 Some Simple Programs to Write

NB. The environment `gwendolen.tutorials.SearchAndRescueEnv` contains rubble both at (5, 5) and at (3, 4).

1. Write a program to make the robot check every square in a 5x5 grid (i.e., (1, 1), (1, 2), (1, 3) etc.,) until it finds some rubble at which point it stops. Try implementing this program both with and without using lists in plans.

(NB. For the list version you may need to insert a plan that asserts a belief when the rubble is seen, in order to make sure the robot doesn't progress through the squares too rapidly. See comment about do_nothing in the next exercise and further discussion of this in later tutorials).

2. Write a program to make the robot search every square in a 5x5 grid (i.e., (1, 1), (1, 2) etc.,) taking all the rubble it finds to the square (2, 2) until it believes there is only rubble in square (2, 2).

   Hints:

   (a) You may see the warning similar to:

   ```
   ail.semantics.operationalrules.GenerateApplicablePlansEmptyProblemGoal[WARNING|main|2:
   Warning no applicable plan for goal _aall_squares_checked()
   ```

   As noted above, this warning appears if the agent can not find a plan to achieve a goal. Sometimes this arises because of bugs in the code, but it can also happen if the agent has not had a chance to process all new perceptions/beliefs before it once again looks for a plan to achieve a goal (we will talk about this some more in later tutorials).

   It may be worth adding an action, do_nothing, into your plan, this will act to delay the next time the agent attempts to achieve the goal giving it time to process all new beliefs.

   (b) You may need to include at(2, 2) in your goal in some way to make sure the agent actually takes the final piece of rubble to the square (2, 2).

Sample answers for these two exercises can be found in `gwendolen/examples/tutorials/tutorial3/answers`.

6.4 Tutorial 4 — Troubleshooting

This is the fourth in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial looks at some of the things that typically cause errors in GWENDOLEN programs and how to identify and fix the errors.

For this tutorial we will be working with files from previous tutorials but editing them to introduce errors. You may wish to create a separate folder, `tutorial4` for this work and copy files into it. Remember to update the paths in your configuration files if you do so.

GWENDOLEN does not have its own debugger, however you can get a long way using error outputs and logging information.

6.4.1 Path Errors

If you supply the wrong path or filename in a configuration file GWENDOLEN will not be able to find the program you want to run. You will see an error similar to the following:

```
ail.mas.AIL[SEVERE|main|3:24:57]: Could not find file.  Checked:
 /src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen,
 /Users/lad/src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen,
 /Users/lad/Eclipse/mcapl/src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen
```

GWENDOLEN looks for program files on

1. The absolute path given in the configuration file –

   `/src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen`
   above.

2. The path from the `HOME` environment variable (normally the user's home directory on Unix systems) –

   `/Users/lad/src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen`
   above.

3. The path from the directory from which the JAVA program `ail.mas.AIL` is called –

   `/Users/lad/Eclipse/mcapl/src/examples/gwendolen/tutorials/tutorial3/pickuprubble_achiev.gwen`
   above.

4. and the path from `AJPF_HOME` if that environment variable has been set – not shown above.

These should provide sufficient information to appropriately correct the path name.

6.4.2 Parsing Errors

Parsing errors typically arise because of failures to punctuate your program correctly. E.g., failing to close brackets, missing out commas or semi-colons etc.

This is the output that arises if you remove the comma between possible_rubble (X, Y) and ~no_rubble(X, Y) in `pickuprubble_achieve.gwen` from Tutorial 3.

```
line 36:47 mismatched input '~' expecting SEMI
```

followed by the program nevertheless attempting to execute and failing to terminate, so you will need to kill this.

The first line of this output is from the parser. This identifies the line number (36) and character in the line (47) where the error was first noticed. It highlights the character that has caused the problem, ~ and then makes a guess at what it should have been. In this case the guess is incorrect. It suggests a semi-colon, `SEMI`, when a comma is needed. This first line is frequently the most useful piece of output for parsing errors so it is worthwhile watching for these kinds of errors at the start when you try to execute a program.

Parsing errors can also cause plans to fail to apply. For instance, if we delete the comma from between `X` and `Y` in the guard of the first plan of this program we get the following output:

```
line 44:51 extraneous input 'Y' expecting CLOSE
ail.semantics.operationalrules.GenerateApplicablePlansEmptyProblemGoal[WARNING|main|3:
Warning no applicable plan for goal _aholding(rubble)()
```

In this case we once again get some useful output from the parser giving us the line and position in the line where the error occurred. It is followed by a warning that no plan can be found to match the goal `holding(rubble)`. That is because this is the plan that didn't parse.

Some Exercises

Experiment with adding and deleting syntax from your existing programming files and get used to the kinds of parsing errors that they generate. Remember that where a `no applicable plan` warning is generated you will often need to manually stop the program execution.

6.4.3 Why isn't my plan applicable?

As mentioned in section 6.3 sometimes a plan can fail because the agent has not had time to process incoming beliefs and perceptions. We can get more information about the agent's operation by using log messages.

Add the lines

```
log.fine = ail.semantics.AILAgent
```

to the configuration file for the sample solution to the second exercise in section 6.3.6. This is in

```
/src/examples/gwendolen/tutorials/tutorial3/answers/pickuprubble_ex5.2.ail
```

This generates a lot of output. If you are using Eclipse you may need to set Console output to unlimited in Eclipse → Preferences → Run/Debug → Console.

The output should start

```
ail.semantics.AILAgent[FINE|main|4:03:27]: Applying Perceive
ail.semantics.AILAgent[FINE|main|4:03:27]: robot
=============
After Stage StageE :
[square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5),
square(2,1), square(2,2), square(2,3), square(2,4), square(2,5),
square(3,1), square(3,2), square(3,3), square(3,4), square(3,5),
square(4,1), square(4,2), square(4,3), square(4,4), square(4,5),
square(5,1), square(5,2), square(5,3), square(5,4), square(5,5), ]
[]
[]
source(self)::
   *  start||True||+!_aall_squares_checked()()||[]
[]
```

This tells us that the agent is applying the rule from the agent's *reasoning cycle* called `Percieve` (we will discuss the reasoning cycle in a later tutorial).

Then we get the current state of the agent. It is called, `robot`, and we have a list of its beliefs (lots of beliefs about squares), then a list of goals (none at the start because it hasn't yet added the initial goal) and then a list of sent messages (also empty) and then the intentions. In this case the initial intention is the `start` intention and the intention is to acquire the goal `all_squares_checked` which is an achievement goal (the `_a` at the start of the goal name) – again we will cover intentions in a later tutorial.

A little further on in the output the agent adds this as a goal:

```
ail.semantics.AILAgent[FINE|main|4:10:45]: Applying Handle Add Achieve Test Goal with Event
ail.semantics.AILAgent[FINE|main|4:10:45]: robot
=============
After Stage StageD :
[square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1), square(2,
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::
   *  +!_aall_squares_checked()||True||npy()||[]
   *  start||True||+!_aall_squares_checked()()||[]
[]
```

So you can see that `all_squares_checked` now appears as a goal in the goal list.

If you remove the action `do_nothing` from the first plan in `pickuprubble_ex5.2.gwen` then you end up with repeating output of the form:

```
ail.semantics.AILAgent[FINE|main|4:20:16]: Applying Generate Applicable Plans Empty with Problem
ail.semantics.AILAgent[FINE|main|4:20:16]: robot
=============
After Stage StageB :
[at/2-at(5,5), ,
checked/2-checked(1,1), checked(1,2), checked(1,3), checked(1,4), checked(1,5), checked(2,1), che
holding/1-holding(rubble), ,
rubble/2-rubble(2,2), ,
square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1), square(2,2
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::
   *  x!_aall_squares_checked()||True||npy()||[]
   *  +!_aall_squares_checked()||True||npy()||[]
   *  start||True||+!_aall_squares_checked()()||[]
```

Where `x!_aall_squares_checked()||True||npy()||[]` indicates that there is some problem with the goal and the agent is seeking to handle this.

Finding where this problem first occurred in all the output is something of a chore though it can sometimes be possible to search forwards through the output for the first occurrence of `x!_` or for the Warning message. Here we see the agent is in the following state:

```
ail.semantics.operationalrules.GenerateApplicablePlansEmptyProblemGoal[WARNING|main|4:
ail.semantics.AILAgent[FINE|main|4:20:15]: Applying Generate Applicable Plans Empty wi
ail.semantics.AILAgent[FINE|main|4:20:15]: robot
=============
After Stage StageB :
[at/2-at(5,5), ,
checked/2-checked(1,1), checked(1,2), checked(1,3), checked(1,4), checked(1,5), checke
rubble/2-rubble(2,2), rubble(5,5), ,
square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1),
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::
   *  +!_aall_squares_checked()||True||npy()||[]
   *  start||True||+!_aall_squares_checked()()||[]

[source(self)::
   *  +checked(5,5)||True||npy()||[]
, source(percept)::
   *  start||True||-rubble(5,5)()||[]
, source(percept)::
   *  start||True||+holding(rubble)()||[]
]
```

Here we see the agent believes it is at square (5, 5). It believes it has checked all the squares. But it does not yet believe it is holding any rubble. It *does* have an intention to hold rubble:

```
source(percept)::
    *  start||True||+holding(rubble)()||[]
```

but it hasn't processed this yet and so hasn't added `holding(rubble)` to its belief base.

As you learn more about GWENDOLEN, the reasoning cycle and intentions you will be able to get more information from this output. However at the moment it is important to note it can be useful for seeing exactly what is in the agent's belief base and goal base at any time.

Some Exercises

Run some of your other programs with `ail.semantics.AILAgent` set at log level `fine` and see if you can get a feel for how an agents beliefs and goals change as the program executes.

Note: If you add

```
pretty = gwendolen
```

you will get a slightly different presentation of the output in a more natural language format. You may want to experiment with which style of output. you prefer.

6.4.4 Tracing the execution of reasoning rules

Another logger that can be useful is the one that traces the application of Prolog reasoning rules. This can be also be useful for working out why a plan that *should* apply does not. Try adding the line

```
log.fine = ail.syntax.EvaluationAndRuleBaseIterator
```

to `pickuprubble_achieve.ail` in section 6.3. If you now run the program you will get a lot of information about the unification of the reasoning rule starting with:

```
ail...[FINE|...]: Checking unification of holding(rubble)() with unifier []
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Looking for a rule match for square_to_check(X0,Y0) :- (possible_rubble(X0,Y0) & not (no_r
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) with unifier [X-_VC1, X0-_VC1, Y-_VC2, Y0-_
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) and <possible_rubble(1,1), >
ail...[FINE|...]: Unifier for possible_rubble(X0,Y0) and <possible_rubble(1,1), > is [X-1, X0-1, Y-1, Y0-1]
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) with unifier [X-1, X0-1, Y-1, Y0-1]
ail...[FINE|...]: square_to_check(X,Y)() matches the head of a rule.
ail...[FINE|...]: Rule instantiated with [X-1, X0-1, Y-1, Y0-1]
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Looking for a rule match for square_to_check(X0,Y0) :- (possible_rubble(X0,Y0) & not (no_r
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) with unifier [X-_VC3, X0-_VC3, Y-_VC4, Y0-_
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) and <possible_rubble(1,1), >
ail...[FINE|...]: Unifier for possible_rubble(X0,Y0) and <possible_rubble(1,1), > is [X-1, X0-1, Y-1, Y0-1]
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) with unifier [X-1, X0-1, Y-1, Y0-1]
ail...[FINE|...]: square_to_check(X,Y)() matches the head of a rule.
ail...[FINE|...]: Rule instantiated with [X-1, X0-1, Y-1, Y0-1]
ail.mas.DefaultEnvironment[INFO|main|4:44:40]: robot done move_to(1,1)
```

This is the selection process for the first plan in the program. We discuss it line by line.

- First it unifies with the achieve goal `holding(rubble)`. This does not instantiate any variables so there is an empty unifier, `[]`.
- Then it checks the plan guard which is B square_to_check(X, Y).
- Since there is nothing in the belief base about this but there is a reasoning rule it now looks for a unifier between these. Notice how it has renamed the variables in the rule to `X0` and `Y0` – this is to avoid errors arising from false unifications.

- It then attempts to unify B square_to_check(X, Y) with the head of this rule.
- As a result of this unification `X` and `X0` are unified and `Y` and `Y0` are unified. For technical reasons these are unified via *variable clusters*, `VC1` and `VC2` respectively.
- It then checks the body of the rule starting with looking for something to unify `possible_rubble(X0, Y0)`. This unifies with the belief `possible_rubble(1, 1)`.
- The unifier is reported showing all the variables are now unified with the number 1.
- The system then checks to see if `no_rubble(X0, Y0)` unifies with anything using this unifier. The rule will fail if it does match because this predicate was negated.
- It doesn't match so the rule has matched.
- With everything unified to 1.
- The process then repeats because of the way the reasoning rule processes transitions.

If we look later in the trace we can see the same process being run after `no_rubble(1, 1)` has been added to the belief base.

```
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Looking for a rule match for square_to_check(X0,Y0) :- (possible_rubl
ail...[FINE|...]: Checking unification of square_to_check(X,Y)() with unifier []
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) with unifier [X-_VC5,
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) and <possible_rubble(
ail...[FINE|...]: Unifier for possible_rubble(X0,Y0) and <possible_rubble(1,1), > is [
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) with unifier [X-1, X0-1, Y-
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) and <no_rubble(1,1), >
ail...[FINE|...]: Unifier for no_rubble(X0,Y0) and <no_rubble(1,1), > is [X-1, X0-1, Y-
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) with unifier [X-_VC5,
ail...[FINE|...]: Checking unification of possible_rubble(X0,Y0) and <possible_rubble(
ail...[FINE|...]: Unifier for possible_rubble(X0,Y0) and <possible_rubble(3,3), > is [
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) with unifier [X-3, X0-3, Y-
ail...[FINE|...]: Checking unification of no_rubble(X0,Y0) and <no_rubble(1,1), >
ail...[FINE|...]: square_to_check(X,Y)() matches the head of a rule.
ail...[FINE|...]: Rule instantiated with [X-3, X0-3, Y-3, Y0-3]
```

Here after a unifier is found for `no_rubble(1, 1)` that unifier for the rule has failed and the process backtracks to look for a different unifier for `possible_rubble(X0, Y0)` in this instance finding (3, 3) and this time the rule succeeds.

6.4.5 Conclusion

Hopefully this tutorial has given you some basic tools for tracking errors in your GWENDOLEN programs. Although the logging facilities generate a lot of output that can be tiresome to read through, they are occasionally very useful for working out what is going wrong in a program. We will look at more debugging possibilities after we have covered the GWENDOLEN reasoning cycle in a tutorial.

6.5 Tutorial 5 — Events and Intentions

This is the fifth in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial looks in more depth at the GWENDOLEN concepts of Event and Intention. It is primarily theoretical but these will be important concepts for future tutorials.

For this tutorial we will be working with files from previous tutorials. You may wish to create a separate folder, `tutorial5` for this work and copy files into it. Remember to update the paths in your configuration files if you do so.

6.5.1 AIL – The Agent Infrastructure Layer

GWENDOLEN is implemented using the AIL Toolkit. This is mostly irrelevant to the programming of GWENDOLEN agents, however it can be useful in understanding some of the logging output that you may wish to use for debugging, since the logging is based around the underlying JAVA data structures rather than their specific implementation in GWENDOLEN.

The following discussion of intentions in the AIL is taken from [Dennis et al., 2011].

6.5.2 Intentions

AIL's most complex data structure is that which represents an *intention*. BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some from of *deed stack* (deeds include actions, belief updates, and the commitment to goals). Intention structures in BDI languages may also maintain information about the (sub-)goal they are intended to achieve or the event that triggered them. In AIL, we aggregate this information: an intention becomes a stack of tuples of an event, a guard, a deed, and a unifier. This AIL intention data structure is most simply viewed as a matrix structure consisting of four columns in which we record events (new perceptions, goals committed to and so forth), deeds (a plan of future actions, belief updates, goal commitments, etc.), guards (which must be true before a deed can be performed) and unifiers. These columns form an event stack, a deed stack, a guard stack, and a unifier stack. Rows associate a particular deed with the event that has caused the deed to be placed on the intention, a guard which must be believed before the deed can be executed, and a unifier. New events are associated with an empty deed, ϵ.

Example The following shows the full structure for a single intention to clean a room. We use a standard BDI syntax: $!g$ to indicate the goal g, and $+!g$ to indicate the commitment to achieve that goal (i.e., a new goal that g becomes true is adopted). Constants are shown starting with lower case letters, and variables with upper case letters.

event	guard	deed	unifier
`+!clean()`	`dirty(Room)`	`+!goto(Room)`	`Room = room1`
`+!clean()`	$\top$	`+!vacuum(Room)`	`Room = room1`

This intention has been triggered by a desire to clean — the commitment to the goal *clean*() is the trigger event for both rows in the intention. An intention is processed from top to bottom so we see here that the agent first intends to commit to the goal *goto*(*Room*), where *Room* is to be unified with *room*1. It will only commit to this goal if it believes the (guard) statement, *dirty*(*Room*). Once it has committed to that goal it then commits to the goal *vacuum*(*Room*). In many languages the process of committing to a goal causes an expansion of the intention stack, pushing more deeds on it to be processed. So *goto*(*Room*) may be expanded *before* the agent commits to vacuuming the room. In which case the above intention might become

event	guard	deed	unifier
`+!goto(Room)`	$\top$	`+!planRoute(Room, Route)`	`Room = room1`
`+!goto(Room)`	$\top$	`+!follow(Route)`	`Room = room1`
`+!goto(Room)`	$\top$	`+!enter(Room)`	`Room = room1`
`+!clean()`	$\top$	`+!vacuum(Room)`	`Room = room1`

At any moment, we assume there is a *current intention* which is the one being processed at that time. The function $\mathcal{S}_{\text{int}}$ (implemented as a method in AIL) may be used to select an intention. By default, this chooses the first intention from a queue, but this choice may be overridden for specific languages and applications. Intentions can be *suspended* which allows further heuristic control. A suspended intention is, by default, *not* selected by $\mathcal{S}_{\text{int}}$. Typically an intention will remain suspended until some trigger condition occurs, such as a message being received. Many operational semantic rules (such as those involved with perception) resume *all* intentions — this allows suspension conditions to be re-checked.

6.5.3 Events

Events are things that occur within the system to which an agent may wish to react. Typically we think of these as changes in beliefs or the new commitment to goals. In many (though not all) programming languages, events trigger plans (i.e., a plan might selected for execution only when the corresponding event has taken place).

In AIL there is a special event, '`start`', that is used to start off an intention which is not triggered by anything specific. This is mainly used for the initial goals of an agent — the intention begins as a `start` intention with the deed to commit to a goal. In some languages the belief changes caused by perception

are also treated in this way. Rather than being added directly to the belief base, in AIL such beliefs are assigned to intentions with the event `start` and then added to the belief base when the intention is actually executed.

6.5.4 Intentions in GWENDOLEN

Let us look back at some of the logging output generated in section 6.4

```
ail.semantics.AILAgent[FINE|main|4:10:45]: Applying Handle Add Achieve Test Goal with Event
ail.semantics.AILAgent[FINE|main|4:10:45]: robot
=============
After Stage StageD :
[square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1), square(2,
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::
   *  +!_aall_squares_checked()||True||npy()||[]
   *  start||True||+!_aall_squares_checked()()||[]
[]
```

In the above agent state you can see the belief base (beliefs about squares), goal base (`all_squares_checked`), the empty sent messages box and then the current intention. The main addition is the record of a *source* for the intention (in this case `self` – which means the agent generated the intention itself rather than getting it via perception).

As you can see GWENDOLEN uses the `start` event mentioned above for new intentions. In this case it was the intention to commit to the achieve goal ! all_squares_checked . When the goal was committed to it became an event and was placed as a new row on the top of the intention. The row associated with the `start` event remains on the intention because this will force the agent to check if the goal is achieved when it reaches that row again. A special deed has been used `npy()` which stands for *no plan yet*. This means that although the goal has been committed to the agent has not yet looked for an applicable plan for achieving the goal.

If you run `pickuprubble_ex5.2.gwen` with logging for `ail.semantics.AILAgent` set to fine then you will later see the intention become:

```
source(self)::
   *  +!_aall_squares_checked()||True||move_to(X,Y)()||[X-1, X0-1, Y-1, Y0-1]
   *  start||True||+!_aall_squares_checked()()||[]
```

when an applicable plan is found the intention becomes

```
source(self)::
   *  +!_aall_squares_checked()||True||move_to(X,Y)()||[X-1, X0-1, Y-1, Y0-1]
      +!_aall_squares_checked()||True||do_nothing()||[X-1, X0-1, Y-1, Y0-1]
   *  start||True||+!_aall_squares_checked()()||[]
```

So now the intention is to first take a `move_to` action in order to get to (1, 1) and then make a `do_nothing` action and then check if the goal has been achieved.

After the agent performs the move action new information comes in from perception.

```
source(percept)::
   *  start||True||+at(1,1)()||[]

[source(self)::
   *  +!_aall_squares_checked()||True||do_nothing()||[]
   *  start||True||+!_aall_squares_checked()()||[]
]
```

The first intention in this list is the *current intention* which is the one the agent will handle next. In this case it is a new intention (indicated by the `start` event) and the intention is to add the belief, `at(1, 1)`. The source of this intention is noted as `percept` (i.e., perception) rather than the agent itself.

Since the agent hadn't finished processing the existing intention this is now contained in a list of other intentions.

When the agent adds the new belief the current intention becomes empty, but GWENDOLEN actually adds yet another new intention indicating that a new belief has been adopted which allows the agent to react to this with a new plan. So the agent's intentions become

```
source(percept)::

[source(self)::
   *  +!_aall_squares_checked()||True||do_nothing()||[]
   *  start||True||+!_aall_squares_checked()()||[]
, source(self)::
   *  +at(1,1)||True||npy()||[]
]
```

The current intention is empty, and there are now two intentions waiting for attention. The empty intention will be removed as the agent continues processing.

GWENDOLEN works on each intention in turn handling the top row on the intention. So the very first intention becomes the current intention again in due course:

```
source(self)::
   *  +!_aall_squares_checked()||True||do_nothing()||[]
   *  start||True||+!_aall_squares_checked()()||[]

[source(self)::
   *  +at(1,1)||True||npy()||[]
]
```

After the agent has done nothing, therefore, the new intention triggered by the new belief `at(1, 1)` becomes the current intention:

```
source(self)::
   *  +at(1,1)||True||npy()||[]

[source(self)::
   *  start||True||+!_aall_squares_checked()()||[]
]
```

If there was no plan for reacting to the new belief the agent would just delete the intention but since there is a plan the intention becomes

```
source(self)::
   *  +at(X0,Y0)||True||+checked(X0,Y0)()||[X-1, X0-1, Y-1, Y0-1]

[source(self)::
   *  start||True||+!_aall_squares_checked()()||[]
]
```

And so on.

Exercises

Run some of your existing programs with logging of `ail.semantics.AILAgent` set to fine and see if you can follow how the agent is handling events, intentions and plans.

6.6 Tutorial 6 — Manipulating Intentions and Dropping Goals

This is the sixth in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers finer control of intentions by suspending and locking them. It also looks at how goals can be dropped.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial6`.

6.6.1 Wait For: Suspending Intentions

Recall the sample answer to the second exercise in section 6.3 in which we had to introduce a "do nothing" action in order to delay the replanning of an achievement goal. In the code in Example 18 we use, instead some new syntax `*checked(X, Y)` which means *wait until checked*(X, Y) *is true before continuing.*

We have adapted the program so that after moving to the square (X, Y) the agent waits until it believes it has checked that square. Then we delay the addition of that belief until after the agent as lifted rubble.

Example 18

```
GWENDOLEN

:name: robot

:Initial Beliefs:

square(1, 1) square(1, 2) square(1, 3) square(1, 4) square(1, 5)
square(2, 1) square(2, 2) square(2, 3) square(2, 4) square(2, 5)
square(3, 1) square(3, 2) square(3, 3) square(3, 4) square(3, 5)
square(4, 1) square(4, 2) square(4, 3) square(4, 4) square(4, 5)
square(5, 1) square(5, 2) square(5, 3) square(5, 4) square(5, 5)

:Reasoning Rules:

square_to_check(X, Y) :- square(X, Y), ~checked(X, Y);
no_rubble_in(X, Y) :- checked(X, Y), no_rubble(X, Y);
all_squares_checked :-
      ~square_to_check(X, Y), ~holding(rubble), at(2, 2);

:Initial Goals:

all_squares_checked [achieve]

:Plans:

+!all_squares_checked [achieve] :
     {B square_to_check(X, Y), ~B holding(rubble)} ←
          move_to(X, Y), *checked(X, Y);
+!all_squares_checked [achieve] : {B holding(rubble)} ←
     move_to(2, 2), drop;

+rubble(X, Y) : {~B at(2, 2)} ← lift_rubble, +checked(X, Y);

+at(X, Y) : {~B rubble(X, Y)} ← +checked(X, Y);
+at(2, 2) : {True} ← +checked(2, 2);
```

If you run this program with logging for `ail.semantics.AILAgent`, you will see that the intention is marked as `SUSPENDED` when the wait for deed is encountered.

```
SUSPENDED
source(self)::
   *  +!_aall_squares_checked()||True||+*...checked(1,1)()||[X-1, Y-1]
   *  start||True||+!_aall_squares_checked()()||[]
```

Once an intention is suspended it can not become the current intention until it is unsuspended. In the case of the wait for command this happens when the

predicate that is waiting for is believed. Below you can see how this happens when `checked(1, 1)` is added to the belief base.

```
ail.semantics.AILAgent[FINE|main|4:01:48]: robot
=============
After Stage StageC :
[at/2-at(1,1), ,
square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1), square(2,2
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::
   *  +at(X0,Y0)||True||+checked(X0,Y0)()||[X-1, X0-1, Y-1, Y0-1]

[SUSPENDED
source(self)::
   *  +!_aall_squares_checked()||True||+*...checked(1,1)()||[X-1, Y-1]
   *  start||True||+!_aall_squares_checked()()||[]
]
ail.semantics.AILAgent[FINE|main|4:01:48]: Applying Handle Add Belief with Event
ail.semantics.AILAgent[FINE|main|4:01:48]: robot
=============
After Stage StageD :
[at/2-at(1,1), ,
checked/2-checked(1,1), ,
square/2-square(1,1), square(1,2), square(1,3), square(1,4), square(1,5), square(2,1), square(2,2
[all_squares_checked/0-[_aall_squares_checked()]]
[]
source(self)::

[source(self)::
   *  +!_aall_squares_checked()||True||+*...checked(1,1)()||[X-1, Y-1]
   *  start||True||+!_aall_squares_checked()()||[]
, source(self)::
   *  +checked(1,1)||True||npy()||[]
]
```

The wait for command is particularly useful in simulated or physical environments where actions may take some time to complete. It allows the agent to continue operating (e.g., performing error monitoring) while waiting until it recognises that an action has finished before continuing with what it was doing.

6.6.2 Lock and Unlock: Preventing interleaving of Intentions

In the code in Example 19 we have complicated our agent's situation a little. This agent has to explore squares (0, 0) to (0, 5) as well as the squares it was

exploring previously. It also has to switch warning lights on and off before and after it lifts rubble. Lastly if a warning sounds it must stop searching and move to square (0, 0) until it is able to continue searching again.

We use some new syntax here.

- At line 34 we have an empty plan. This can be useful in situations where we don't want to raise a "no plan" warning but we don't want the agent to actually do anything.
- At line 36 we have a plan triggered by −warning. This is a plan that is triggered when something is no longer believed (in this case that the warning sound can no longer be heard).
- At line 37 we include the deed, −search_mode in a plan. This is an instruction to remove a belief.

The agent in Example 19 uses a belief, search_mode to control whether it is actively searching squares or whether it is returning to to the "safe" square (0,0) in order to wait for the warning to switch off.

Run this program and see if you can spot a problem with its execution.

Hopefully you observed an output something like:

```
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done move_to(0,1)
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done move_to(0,2)
gwendolen.tutorials.SearchAndRescueEnv[INFO|main|10:31:44]: Warning is Sounding
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done warning_lights_on
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done move_to(0,0)
gwendolen.tutorials.SearchAndRescueEnv[INFO|main|10:31:44]: Warning Ceases
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done lift_rubble
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done warning_lights_off
ail.mas.DefaultEnvironment[INFO|main|10:31:44]: robot done move_to(2,2)
```

So *before* the robot lifts the rubble at square (0, 2) it has moved to square (0, 0) because the warning has sounded. This is happening because GWENDOLEN executes the top deed from each intention in turn. So it executes warning_lights_on from the intention triggered by finding rubble, then it moves to (0, 0) from the intention triggered by hearing the warning and then it lifts the rubble (next in the intention to do with seeing the rubble).

This situation often arises where there are a sequence of deeds that need to be performed *without interference* from other intentions such as moving to the wrong place. To overcome this GWENDOLEN has a special deed, .lock which "locks" an intention in place and forces GWENDOLEN to execute deeds from that intention *only* until the intention is unlocked. The syntax +.lock locks an intention and the syntax −.lock unlocks an intention.

Exercise Add a lock and an unlock to `pickuprubble_lock` in order to force it to pick up the rubble before obeying the warning.

NB. As usual you can find a sample solution in `/src/examples/gwendolen/tutorials/tutorial6/answers`

Example 19

```
GWENDOLEN

:name: robot

:Initial Beliefs:

square(0, 0)
square(0, 1) square(0, 2) square(0, 3) square(0, 4) square(0, 5)
square(1, 1) square(1, 2) square(1, 3) square(1, 4) square(1, 5)
square(2, 1) square(2, 2) square(2, 3) square(2, 4) square(2, 5)
square(3, 1) square(3, 2) square(3, 3) square(3, 4) square(3, 5)
square(4, 1) square(4, 2) square(4, 3) square(4, 4) square(4, 5)
square(5, 1) square(5, 2) square(5, 3) square(5, 4) square(5, 5)

search_mode

:Reasoning Rules:

square_to_check(X, Y) :- square(X, Y), ~checked(X, Y);
no_rubble_in(X, Y) :- checked(X, Y), no_rubble(X, Y);
all_squares_checked :-
     ~square_to_check(X, Y), ~holding(rubble), at(2, 2);

:Initial Goals:

all_squares_checked [achieve]

:Plans:

+!all_squares_checked [achieve] : {~ B search_mode} <-
     *search_mode;
+!all_squares_checked [achieve] :
     {B search_mode, B square_to_check(X, Y), ~B holding(rubble)} <-
          move_to(X, Y), *checked(X, Y);
+!all_squares_checked [achieve] : {B holding(rubble)};

-warning: {True} <- +search_mode;
+warning: {True} <- -search_mode, move_to(0, 0);

+rubble(X, Y) : {~B at(2, 2)} <-
     warning_lights_on,
     lift_rubble,
     warning_lights_off,
     move_to(2, 2),
     drop,
     +checked(X, Y);

+at(X, Y) : {~B rubble(X, Y)} <- +checked(X, Y);
+at(2, 2) : {True} <- +checked(2, 2);
```

6.6.3 Dropping Goals

As a final note as well as dropping beliefs as a deed in plans (as we are doing with warning and search_mode in the programs here, it is possible to drop goals with the syntax −!goalname [goaltype] - e.g., −! all_squares_checked [achieve].

Goal drops can appear in the deeds of plans but can not[1] be used to trigger plans.

Exercise Write a program for picking up and moving rubble which, on hearing the warning sound, drops all its goals and leaves the area (use the action `leave`).

NB. As usual you can find a sample solution in `/src/examples/gwendolen/tutorials/tutorial6/answers`

6.7 Tutorial 7 — The Gwendolen Reasoning Cycle

This is the seventh in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers the GWENDOLEN Reasoning Cycle, looks at some simple ways to use a Java Debugger to debug GWENDOLEN programs and sets a more significant programming challenge than previous tutorials.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial7`.

6.7.1 The GWENDOLEN Reasoning Cycle

The execution of a GWENDOLEN agent is governed by a *reasoning cycle*. This is a set of stages the agent passes through, each stage is governed by a set of rules and the agent may choose one to execute in that stage. The reasoning cycle is shown in figure 6.2

Stage A A GWENDOLEN agent starts execution in stage A. In this stage the agent selects an intention to be the current intention. GWENDOLEN will cycle through the set of intentions ignoring any that are suspended until the current intention is locked in which case it will be reselected. If there are no unsuspended intentions GWENDOLEN will sleep the agent. In multi-agent contexts this means the agent will not do anything until GWENDOLEN detects that something has changed which may mean the agent now has something to do. In single agent contexts the program stops when the agent sleeps. At this stage GWENDOLEN also cleans up any empty intentions.

Stage B The system generates all possible plans for the current intention - if the intention has already been planned then these are simply to continue

[1] at least not at present.

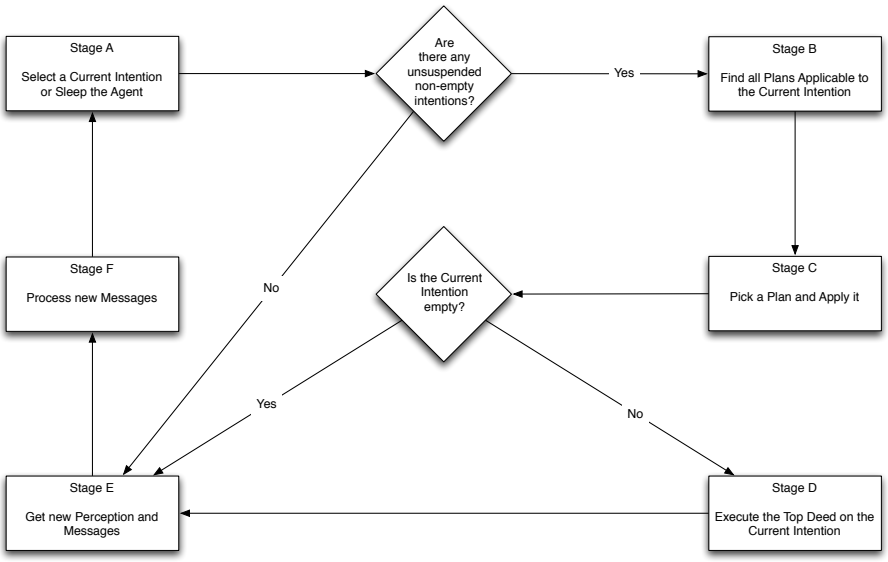

Figure 6.2: The GWENDOLEN Reasoning Cycle

processing the intention. If the agent can't find a plan then it deletes the intention unless it has been triggered by a goal in which case it registers that there is a problem with the goal and generates a warning.

Stage C GWENDOLEN has a list of plans. It selects the first one in the list and applies it to the current intention.

Stage D GWENDOLEN executes the top deed on the intention. This might be taking an action, adding or removing a goal, adding or removing a belief, locking or unlocking the intention or suspending the intention using "wait for".

Stage E GWENDOLEN requests that the agent's environment send it a list of *percepts* (things the agent can detect) and messages. The messages are stored for processing in the agent's inbox. The percepts are compared with the agent's beliefs. If a percept is new then an intention is created to add a belief corresponding to the percept. If a previous percept can no longer be perceived then an intention is created to delete the belief.

Stage F The agent sorts though its inbox and converts the messages into new intentions.

The actual code for the reasoning cycle can be found in `gwendolen.semantics.GwendolenRC`. Each of the various rules that can be used in in a stage is a java class and they can all be found in the package `ail.semantics.operationalrules`.

6.7.2 Using Java Debuggers to Debug GWENDOLEN programs

Since the GWENDOLEN reasoning cycle is implemented in JAVA it is possible to use a JAVA debugger to debug GWENDOLEN programs. In particular it can be useful to use a JAVA debugger to step through a GWENDOLEN program one stage of the reasoning cycle at a time watching to see how the state of the agent changes at each stage.

It is outside the scope of these tutorials to explain the use of JAVA debuggers. There are many out there and one is built into most IDE's including Eclipse.

In our experience it is particularly useful when debugging in this way to place breakpoints in the JAVA `ail.semantics.AILAgent` class which is the generic class supporting agents in the Agent Infrastructure Layer (upon which GWENDOLEN is built). In particular `ail.semantics.AILAgent` has a method called `reason` which controls looping through an agent's reasoning cycle. We recommend placing such a break point either after `while(! RC.stopandcheck())` which is the top level loop through the reasoning cycle or at `rule.apply(this)` which is the moment that the outcome of a rule is calculated.

Exercise Find a JAVA debugger (e.g., the one shipped with Eclipse) and discover how to set breakpoints using the debugger. Set a breakpoint at `rule.apply(this)` in the `reason()` method in `ail.semantics.AILAgent` (you can find this in the `src/classes/ail/semantics` directory). Run one of your programs with this breakpoint in place and see what happens and experiment in seeing what information you can discover about the agent state.

6.7.3 Programming Exercise

This is a fairly major programming exercise using the GWENDOLEN constructs you have already been introduced to. As usual a sample solution can be found in the `answers` subdirectory for `tutorial7`.

In `examples/gwendolen/tutorials/tutorial7` you will find a new environment, `SearchAndRescueDynamicEnv.java`. This extends the previous search and rescue example so that the environment may change with the agent directly taking any action. The following describes the environment.

The environment consists of a 5x5 grid of squares. The squares in the grid are numbered from (0, 0) (bottom left) to (4, 4) (top right). On this grid are

- Exactly four humans which may start in any square on the grid. Some of these humans may be **injured**.
- Exactly one robot which starts in the bottom left corner of the grid.
- Up to four buildings which may appear in any square on the grid.
- Up to four bits of rubble which may appear in any square on the grid. Any human on the same square as some rubble at the start is **injured** and is hidden under the rubble.

At any point the following may happen:

- A human who is not **injured**, **in a building**, or has been **directed to leave** the area may move one square in any direction.
- A building may collapse into rubble.

The robot has the following actions available to it:

back_left If possible the robot will move one square diagonally down the grid and to the left. If not possible the robot will do nothing.

back If possible the robot will move one square down the grid. If not possible the robot will do nothing.

back_right If possible the robot will move one square diagonally down the grid and to the right. If not possible the robot will do nothing.

left If possible the robot will move one square to the left. If not possible the robot will do nothing.

right If possible the robot will move one square to the right. If not possible the robot will do nothing.

forward_left If possible the robot will move one square diagonally forward in the grid and to the left. If not possible the robot will do nothing.

forward If possible the robot will move one square forward in the grid. If not possible the robot will do nothing.

forward_right If possible the robot will move one square diagonally forward in the grid and to the right. If not possible the robot will do nothing.

lift_rubble If the robot is not currently holding rubble then it will pick up one piece of rubble in the square revealing anything underneath it.

drop_rubble If the robot is holding rubble then it will drop it in the current square, injuring and concealing any humans if they are in the square.

assist_human If there is an injured human in the square then the robot treats them with first aid. After this the human is not injured.

direct_humans If there are any humans in the square then the robot tells them to leave the area immediately.

check_building If there is a building in the square then the robot looks inside it to see if there is a human there.

Other Actions Other standard actions such as `print` and `do_nothing` are also available.

The robot may perceive the following things:

holding_rubble perceived if the robot has rubble in its hands.

rubble(X, Y) perceived if the robot sees some rubble in square (X, Y). The robot can see the square it is in and one square in each direction.

building(X, Y) perceived if the robot sees a building in square (X, Y). The robot can see the square it is in and one square in each direction.

injured_human(X, Y) perceived if the robot sees an injured human in square (X, Y). The robot can see the square it is in and one square in each direction.

uninjured_human(X, Y) perceived if the robot sees an uninjured human in square (X, Y). The robot can see the square it is in and one square in each direction.

The following also hold true:

- If a human is **in a building** when it collapses then they will be **injured** and concealed by the rubble.
- If a human is **in a building** then the robot can not see them unless it checks the building.

Humans exhibit the following behaviour:

- **injured** humans do not move.
- **directed** humans move diagonally down and left until they leave the grid. They do not enter buildings.
- If a human is not **directed** and finds itself in a square with a building then it will enter the building and stay there.
- Humans which are not **injured**, **directed** or **in a building** will move at random around the grid.

Recording and Replaying AIL Programs

We will cover recording and replaying AIL programs in more detail in a later tutorial. However some of the challenges from this tutorial will arise because of the difficulty in reproducing a specific sequence of events in the environment. In the AIL configuration file you can put the line

```
ajpf.record = true
```

This will record the sequence of events that occur in the environment (and store them in a file called `record.txt` in the folder `records`). If you want to *replay* the last recorded run of the problem then replace

```
ajpf.record = true
```

with

```
ajpf.replay = true
```

Exercise

Write a GWENDOLEN program that will get the robot to search the grid until all humans have been found, assisted if injured, and directed to leave.

6.8 Tutorial 8 — Multi-Agent Systems and Communication

This is the eighth in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers the use of communication in GWENDOLEN and also looks at setting up a multi-agent system.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial8`.

6.8.1 Pick Up Rubble (Again)

You will find a GWENDOLEN program in the tutorial directory called `simple_mas.gwen`. Its contents should look like Example 20.

This is very similar to the first program in section 6.2. However there are now two agents, `lifter` and `medic`. As in the program in section 6.2, the lifter robot moves to square (5, 5) and lifts the rubble there. However if he sees a human he performs a special kind of action which is a *send action*. This sends a message to the medic agent asking it to perform `assist_human(X, Y)`. When the medic receives a perform instruction it converts it into a perform goal and if it has a goal to assist a human it moves to their square and assists them.

You can run this program using `simple_mas.ail`. It uses a new environment `SearchAndRescueMASEnv.java` which is similar to `SearchAndRescueEnv.java`.

Syntax

A send action starts with the constant `.send`. It then has three arguments:

1. The first is the name of the agent to whom the message is to be sent,
2. the second is a performative, and
3. the last is a logical term.

The performative can be one of `:tell`, `:perform` or `:achieve`. GWENDOLEN attaches no particular meaning to these performatives but they are often used to tell an agent to believe something, ask an agent to adopt a perform goal or ask an agent to adopt an achieve goal.

When a message is received GWENDOLEN turns it into an event: `.received(P, F)` were `P` is the performative and `F` is the logical term. Since many GWENDOLEN programs interpret `:tell`, `:perform` and `:achieve` as described above, they often include the following three plans

Example 20

```
GWENDOLEN

:name: lifter

:Initial Beliefs:

:Initial Goals:

goto55 [perform]

:Plans:

+!goto55 [perform] : {True} ← move_to(5, 5);

+rubble(5, 5): {True} ← lift_rubble;

+human(X, Y): {True} ← .send(medic, :perform, assist_human(X, Y));

:name: medic

:Initial Beliefs:

:Initial Goals:

:Plans:

+.received(:perform, G): {True} ← +!G [perform];

+!assist_human(X, Y) [perform] : {True} ←
        move_to(X, Y),
        assist;
```

```
+.received(:tell, B): {True} <- +B;
+.received(:perform, G): {True} <- +!G [perform];
+.received(:achieve, G): {True} <- +!G [achieve];
```

which embody that interpretation. However many programs instead choose only to handle certain performatives (e.g., only `:tell` messages) or only certain message contents, (e.g., `.received(:perform, assist_human(X, Y))` only handles messages asking the agent to perform `assist_human(X, Y)` for some `X` and `Y`).

Exercise

Amend the `simple_mas` program so that, instead of sending a perform message, the lifter agent sends a tell message and the medic reacts to the new belief,

instead of the new goal.

NB. It is important, for using the `SearchAndRescueMASEnv.java` environment that the lifting agent be called `lifter` and the medic agent be called `medic`.

As usual sample solutions to all the exercises can be found in the `answers` directory for `tutorial8`.

6.8.2 Recording and Replaying AIL Programs

Now there is more than one agent in the system, you will observe that there are several paths through the program. These depend upon which agent acts when. Sometimes the `lifter` agent will go first (moving to (5, 5)) and sometimes the `medic` agent will go first (sleeping).

When debugging a multi-agent program you sometimes want to replay the exact sequence of events that occurred in the problem run. To do this you first need to record the sequence. You can get an AIL program to record its sequence of choices (in this case choices about which agent goes first) by adding the line

```
ajpf.record = true
```

To the program's AIL configuration file. By default this records the current path through the program in a file called `record.txt` in the directory, `records` of the MCAPL distribution. You can change the file using `ajpf.replay.file =`. There is an example of this in the configuration file `simple_mas_record.ail` in the tutorial directory.

When you want to play back a record then include

```
ajpf.replay = true
```

In the program's AIL configuration file. Again, by default, this will replay the sequence from `record.txt`, but will use a different file if `ajpf.replay.file =` is set. The configuration file `simple_mas_replay.ail` is set up to replay runs generated by `simple_mas_record.ail`

6.8.3 Two Ways to Create a Multi-Agent System

In the previous example we put all the agents in a multi-agent system in one file. However you often want to separate out your agents into different files, one for each agent. This is easy to do in the AIL. You write each agent as you normally would in a separate file. Then in the `.ail` file for running the system instead of using `mas.file` you use `mas.agent.1.file` (for the file containing agent one), `mas.agent.2.file` etc. Similarly instead of using a MAS builder you link to individual agent builders. GWENDOLEN's agent builder is `gwendolen.GwendolenAgentBuilder` – so you use

```
mas.agent.1.builder = gwendolen.GwendolenAgentBuilder
```

etc., for each agent rather than

```
mas.builder = gwendolen.GwendolenMASBuilder.
```

Exercise

Convert `simple_mas.gwen` into a system consisting of two agents in different files. NB. You will need to make sure both agent files start with the declaration `GWENDOLEN` for the language the agent is programmed in.

6.8.4 Duplicating an Agent

Sometimes you want to create a multi-agent system in which all agents behave identically. Ideally you would like to use the same agent code file for all these agents and just give them different names in the multi-agent system.

You can do this using files and builders, as above, with the addition of a `name` setting. So, for instance, `mas.agent.3.name = nurse` sets the name of agent 3 to `nurse` instead of whatever is given in the agent file.

Exercise

Adapt the system from exercise 2 by creating a new lifter agent that visits first square (5, 5) and summons the medic to assist the human there and, after that, visits square (3, 4) and summons a nurse to assist the human there. The medic and the nurse should both use the medic agent code file you developed for exercise 2. Give one of these agent's the name `nurse` in the `.ail` file.

6.9 Tutorial 9 — Default built-in actions: Strings and Arithmetic

This is the ninth in a series of tutorials on the use of the GWENDOLEN programming language. This tutorial covers a few final elements of GWENDOLEN and the actions that come with the Default Environment. It is important to note that if a GWENDOLEN agent *isn't* operating in some environment sub-classed from `DefaultEnvironment` then there is no guarantee that these actions will be available.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/gwendolen/tutorials/tutorial9`.

6.9.1 String Handling

In the tutorial directory you will find a program called `strings.gwen`. It's contents should look like Example 21

Example 21

```
:name: strings

:Initial Beliefs:

string1("hello")
string2(" ")
string3("world")

:Initial Goals:

print_string [perform]

:Plans:

+! print_string [perform] : {True} ←
        print("hello world");
```

If you run this program you will see that it prints out `hello world`. Here "print" is an action which is implemented in `DefaultEnvironment`

Built-in String Actions

If you look at `strings.ail` you will see that you are using AIL's `DefaultEnvironment` class. Most GWENDOLEN environments are based on the default environment and this means they all support a set of standard actions that come with the Default Environment. The built-in actions for strings are:

toString(T, S) This will take any term, T, that you are passing around your program and unify the variable, S, to that term.

append(S1, S2, S3) This takes two strings, S1 and S2 and unifies, S3, to the concatenation of those two strings. So, for instance, append(''gwen'',''dolen'',S) will unify S to gwendolen.

Exercise

You will notice that `strings.gwen` contains three beliefs about strings. Adapt the program so that instead of printing out `hello world` directly, it instead uses append to join the three strings together to print out the message.

Hint. You will need to use append twice.

As usual you can find sample solutions in the `answers` directory.

6.9.2 Arithmetic

GWENDOLEN can use numbers as terms but it is both fiddly and inefficient to program up arithmetic operations using Reasoning Rules. As a result the Default environment has four simple actions for manipulating numbers.

sum(X, Y, Z) This unifies Z to the sum of X and Y.

minus(X, Y, Z) This takes Y away from X and unifies Z to the result.

div(X, Y, Z) This divides X by Y and unifies Z to the result.

times(X, Y, Z) This multiplies X by Y and unifies Z to the result.

Exercise

In the tutorial directory you will find a partial program, `arithmetic_shell.gwen`. This is shown in Example 22

Example 22

```
GWENDOLEN

:name: arithmetic

:Initial Beliefs:

:Initial Goals:

do_maths [perform]

:Plans:

+! do_maths[perform] : {True} ←
        +! do_sum [perform],
        +! do_minus [perform],
        +! do_div [perform],
        +! do_mult [perform];
```

Implement the four missing plans so that

- do_sum adds two numbers and prints out the result as, for instance, `The Sum of 1 and 5 is 6`. You will need to use `toString` and `append` to generate the string you want.
- do_minus subtracts two numbers and prints out the result as, for instance, `5.5. take 3.2. is 2.3`.
- do_div divides one number by another and prints out the result as, for instance, `7 divided by 2 is 3.5`
- do_mult multiplies two numbers and prints out the result as, for instance, `100 times 2.5 is 250`.

6.9.3 Using Equations in Plan Guards

Once you are using numbers in your program you quickly get to situations where you want to use equations in plan guards. GWENDOLEN has some limited support for this. It can't perform arithmetic in the guards of plans, but it can compare numbers using < (less than) and == (equals).

Exercise

In the tutorial directory you will find a partial program, `equation_shell.gwen`. This is shown in Example 23

Example 23

```
GWENDOLEN

:name: equation

:Initial Beliefs:

number1(3)
number2(5)
number3(4.8)
number4(3)

:Initial Goals:

compare_numbers [perform]

:Plans:

+! compare_numbers [perform] : {B number1(N1), B number2(N2),
                                 B number3(N3), B number4(N4)} ←
        +!compare(N1, N2) [perform],
        +!compare(N1, N3) [perform],
        +!compare(N1, N4) [perform],
        +!compare(N2, N3) [perform],
        +!compare(N2, N4) [perform],
        +!compare(N3, N4) [perform];
```

Complete this program by implementing plans for the goal, compare(N1, N2), so that the program prints out the following output.

```
3 is less than 5
3 is less than 4.8
3 is equal to 3
4.8 is less than 5
3 is less than 5
3 is less than 4.8
```

6.9.4 Print Actions

GWENDOLEN's default environment has three print actions.

print(X) you have already encountered and prints out the term, X.

printagentstate prints the current state of the agent to standard error.

printstate prints the current state of the agent to standard out.

Clearly printagentstate and printstate are virtually identical. They are mostly of use when debugging and generally either can be used, but in certain situations you may have a preference about which output channel you want to use.

Exercise

Experiment inserting printagentstate and printstate into one of your existing programs.

6.10 The Property Specification Language and its Relation to GWENDOLEN Programs

6.10.1 Implementation of BDI Modalities in GWENDOLEN

In GWENDOLEN the BDI modalities of the AJPF property specification language are implemented as follows.

- $\mathcal{B}_{ag}\, f$. An agent, ag, believes the formula, f, if f appears in its belief base or is deducible from its goal base using its reasoning rules.
- $\mathcal{G}_{ag} f$. An agent, ag, has a goal f, if f is a goal that appears in the agent's goal base.
- $\mathcal{I}_{ag} f$. An agent, ag, has an intention f, if f is a goal in the goal base a plan has been selected to achieve or perform the goal.
- $\mathcal{ID}_{ag} f$. An agent, ag, intends to do f, if f is an action that appears in the deed stack of some intention.

Intending to Send a Message

GWENDOLEN uses a special syntax for send actions (`.send(ag, :tell, c)`) which is not recognised by the property specification language. If you want to check that a GWENDOLEN agent intends to send a messsage then you need to use the syntax `send(agname, number, c)` where `agname` is the name of the recipient, `number` is

1 For `:tell`,

2 For `:perform`,

3 For `:achieve`

and `c` is the content of the message.

Chapter 7

Gwendolen Semantics

This chapter duplicates the operational semantics for GWENDOLEN presented in [Dennis, 2017].

7.1 Intentions

Intentions are crucial to understanding GWENDOLEN. BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some from of *deed stack* (deeds include actions, belief updates, and the commitment to goals). Intention structures also maintain information about the (sub-)goal they are intended to achieve or the event that triggered them. GWENDOLEN aggregates this information: an intention becomes a stack of tuples of an event, a deed, and a unifier. This tuple is most simply viewed as a matrix structure consisting of three columns in which we record events (new perceptions, goals committed to and so forth), deeds (a plan of future actions, belief updates, goal commitments, etc.), and unifiers. These columns form an event stack, a deed stack, and a unifier stack. Rows associate a particular deed with the event that has caused the deed to be placed on the intention, and a unifier. New events are associated with an empty deed, ϵ.

Example The following shows the full structure for a single intention to clean a room. We use a standard BDI syntax: $!g$ to indicate the goal g, and $+!g$ to indicate the commitment to achieve that goal (i.e., a new goal that g becomes true is adopted). Constants are shown starting with lower case letters, and variables with upper case letters.

event	deed	unifier
`+!clean()`	`+!goto(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	`Room = room1`

This intention has been triggered by a goal to clean — the commitment to the goal *clean*() is the trigger event for both rows in the intention. An intention

is processed from top to bottom so we see here that the agent first intends to commit to the goal $goto(Room)$, where $Room$ is to be unified with $room1$. Once it has committed to that goal it then commits to the goal $vacuum(Room)$. In GWENDOLEN the process of committing to a goal causes an expansion of the intention stack, pushing more deeds on it to be processed. So $goto(Room)$ is expanded *before* the agent commits to vacuuming the room and the above intention becomes

event	deed	unifier
`+!goto(Room)`	`+!planRoute(Room, Route)`	`Room = room1`
`+!goto(Room)`	`+!follow(Route)`	`Room = room1`
`+!goto(Room)`	`+!enter(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	`Room = room1`

At any moment, we assume there is a *current intention* which is the one being processed at that time. The stacks that form the intention are further paired with two booleans, *suspended*, and *locked* which indicate the intention's status. A suspended intention is, by default, *not* selected at the intention selection phase of the agent's reasoning. Typically an intention will remain suspended until some belief condition occurs, normally that a belief is acquired via perception or from the receipt of a message. If an intention is locked, conversely, then it must be selected at the intention selection phase.

7.2 Plans, Applicable Plans and Intentions

A GWENDOLEN agent also has a *plan library* which is an ordered list of plans. Plans are matched against intentions and manipulate them. There are three main components to a plan,

1. A *trigger event* which may match the top event of an intention.
2. A *guard*: the guard is checked against the agent's state for plan applicability.
3. A *body* which is the new deed stack that the plan proposes for execution.

We use the syntax $trigger : \{guard\} \leftarrow body$ to represent plans.

Plans only match intentions which contain unplanned goals (i.e., those associated with the "no plan yet" deed, ϵ). For instance after a commitment to `goto(Room)` the above intention might appear as:

event	deed	unifier
`+!goto(Room)`	ϵ	`Room = room1`
`+!clean()`	`+!goto(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	`Room = room1`

which would match the plan

$+!goto(Room) : \{upstairs(Room)\} \leftarrow +!goto(stairs); +!goto(Room)$

This plan says that in order to achieve the goal $goto(Room)$ in the case where the room is upstairs, $(upstairs(Room))$, first the goal $goto(stairs)$ must be achieved and then the goal $goto(Room)$ achieved.

This would transform the intention to:

event	deed	unifier
`+!goto(Room)`	`+!goto(stairs)`	`Room = room1`
`+!goto(Room)`	`+!goto(Room)`	`Room = room1`
`+!clean()`	`+!goto(Room)`	`Room = room1`
`+!clean()`	`+!vacuum(Room)`	`Room = room1`

7.2.1 Applicable Plans

Applicable plans are an interim data structure that describe how a plan from an agent's plan library changes the current intention. An applicable plan describes the new rows that will replace the top row of the intention. The new rows are generated from an event, a unifier and a stack of deeds. The new intention rows are generated by creating a row for each deed and associating the event and unifier with each of those rows (so the event and unifier are duplicated several times).

Therefore, an applicable plan is a tuple, (p_e, p_{ds}, p_θ), of an event p_e, a deed stack p_{ds}, and a unifier p_θ. The applicable plan in the example above would be

$$(+!goto(Room), [+!goto(stairs); +!goto(Room)], \{Room = room1\}) \quad (7.1)$$

Applicable plans are used because GWENDOLEN first determines a list of applicable plans and then picks one plan to be applied. The function $\mathcal{S}_{\text{plan}}$ is used to select *one* applicable plan from a set. By default, this treats the set as a list and picks the first plan, but it may be overridden by specific applications.

Applicable Plan Generation Method The function **appPlans**, generates a set of applicable plans from the current intention, i, and an agent's internal state.

There are two cases. In the first case the top deed on the intention is not ϵ (i.e., no planning is needed). In this case the set of applicable plans is for continuing to process intention i without any changes (i.e., it represents the top row of the intention). So the set of applicable plans is the singleton:

$$\{(\mathtt{hd}_e(i), \mathtt{hd}_d(i), \theta^{\mathtt{hd}(i)}) \quad | \quad \mathtt{hd}_d(i) \neq \epsilon\} \quad (7.2)$$

where $\mathtt{hd}_e(i)$ is the top event in i, $\mathtt{hd}_d(i)$ is the top deed, and $\theta^{\mathtt{hd}(i)}$ is the top unifier.

In the case where the top deed on the intention is ϵ, **appPlans** generates the set

$$\begin{array}{l}\{(p_e, p_d, \theta^{\mathtt{hd}(i)} \cup \theta) \quad | \\ \quad p_e : \{p_{gu}\} \leftarrow p_d \in P \ \wedge \ \mathtt{hd}_e(i)\theta^{\mathtt{hd}(i)} \models p_e, \theta' \ \wedge \ ag \models p_{gu}\theta', \theta\}\end{array} \quad (7.3)$$

where P is the agent's library of plans. $\mathtt{hd}_e(i) \models p_e, \theta'$ means that the plan's trigger event follows from the top event on the current intention returning a unifier, θ'. This allows for Prolog-style reasoning on plan triggers.

Notation	Description
$\xi.\mathbf{do}(a)$	Executes an action. Returns a unifier.
$\xi.getmessages(ag)$	Returns a set of new messages for agent ag.
$\xi.\mathbf{Percepts}(ag)$	Returns a set of new perceptions (logical formulae) for agent, ag.
$\xi.done$	True if the environment is incapable of further independent action.

Table 7.1: Methods implemented by GWENDOLEN Environments

The notation $ag \models g, \theta$ means that the guard, g, is satisfied by agent ag given unifier θ. Again this allows Prolog-style reasoning. Plan guards may refer to the agent's belief base, goal base or outbox. For instance $\mathcal{B}b$ means some belief, b should follow by logical inference from the agent's belief base and $\mathcal{G}g$ means that some goal g should follows by logical inference from the goal base.

The notation $t\theta$ indicates the application of unifier θ to term t. So, for instance, $\mathtt{hd}_e(i)\theta^{\mathtt{hd}(i)}$ is the result of applying the unifier $\theta^{\mathtt{hd}(i)}$ to the top event on the intention.

7.3 The Environment

A feature of BDI agent programming languages is that BDI programs do not, in general, stand alone but exist within a computational environment. GWENDOLEN programs expect to interact with environments programmed in JAVAwhich implement a specific interface. This means the semantics of some rules will depend upon the environment used. Environments offer various functions – executing agent actions, supplying sets of perceptions etc. The execution of these functions may also induce a change in the environment itself according to its own semantics.

We represent the environment as ξ. Table 7.1 summarises the functions that all environments are required to offer by the GWENDOLEN semantics. Some environments only change when one of these functions is called but others may be independently dynamic (e.g., because other agents, not programmed in GWENDOLEN are acting in them). We therefore also allow a transition relation on environments ξ: $\xi \rightarrow_\xi \xi'$ and represent the transitions caused by the functions in table 7.1 as $\xi \xrightarrow{\mathbf{do}(a)}_\xi \xi'$, $\xi \xrightarrow{getmessages}_\xi \xi'$ and $\xi \xrightarrow{\mathbf{Percepts}(ag)}_\xi \xi'$. *done* does not change the environment.

7.4 Multi-Agent System Semantics, Scheduling, Reasoning Cycle

A GWENDOLENagent is executed as part of a multi-agent system which includes an environment and a *scheduler*. The scheduler is specific to the application and

so its policy for the order in which agents (and where relevant the environment) are executed varies.

We represent the operational semantics of the multi-agent system a set of transition rules. The first rules, $\rightarrow_s$ operate on tuples of the environment, a set of agents and the scheduler and represents how the scheduler chooses the next agent for execution. The agent then transitions through stages in a *reasoning cycle* (represented with $\rightarrow_a$). At each stage in the reasoning cycle specific rules are selected which cause transitions on the agent (and sometimes also on the environment).

We assume the existence of the following functions: $next\_job(s)$ returns a tuple of an agent (or the environment) and an updated version of the scheduler depending on the scheduler policy; *sleeping(a, s)* returns true if the scheduler lists a as asleep; $sleep(a)$ returns true if the agent's status is that it has no further reasoning at the moment and $sleep(a, s)$ returns an updated scheduler that lists a as sleeping. $\rightarrow_a^*$ represents the transitive closure of the semantics on an agent's reasoning cycle so $\langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle$ represents the effect of a run of the agent's reasoning cycle (from stage **A** to **F** – see below) on both the agent and the environment. $\xi \rightarrow_\xi \xi'$ represents an update of the environment according to its own semantics (not considered here).

The following rules represent the operation of the scheduler.

$$\frac{\neg \xi.done \quad next\_job(a) = \langle \xi, s' \rangle \quad \xi \rightarrow_\xi \xi'}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A, s' \rangle} \tag{7.4}$$

$$\frac{\begin{array}{c}\exists a \in A. \neg sleeping(a, s) \quad next\_job(s) = \langle a, s' \rangle \\ \langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle \quad \neg sleep(a')\end{array}}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A[a \backslash a'], s' \rangle} \tag{7.5}$$

$$\frac{\begin{array}{c}\exists a \in A. \neg sleeping(a, A) \quad next\_job(s) = \langle a, s' \rangle \\ \langle \xi, a, \mathbf{A} \rangle \rightarrow_a^* \langle \xi', a', \mathbf{F} \rangle \quad sleep(a')\end{array}}{\langle \xi, A, s \rangle \rightarrow_s \langle \xi', A[a \backslash a'], sleep(a', s') \rangle} \tag{7.6}$$

It should be noted that, among other things, $next\_job(s)$ can change the internal state of the scheduler, for instance altering the set of agents marked as sleeping if, for instance, new perceptions are available in the environment that might mean the agent now has something to do.

The GWENDOLEN reasoning cycle is a set consisting of size stages (**A, B, C, D, E,** and **F**). Each stage is a list of rules which are discussed in section 7.5. The agent reasoning cycle transitions, $\rightarrow_a$, by picking the first applicable rule, r, from the list in the current reasoning stage, RS, transitioning the agent (and in some cases environment) according to the rule $\rightarrow_r$ and then moving the reasoning cycle on according to the function $next$ (see (7.9)).

$$\frac{\exists r \in rules(RS).\, \langle \xi, a \rangle \rightarrow_r \langle \xi', a' \rangle}{\langle \xi, a, RS \rangle \rightarrow_a \langle \xi', a', next(a', RS) \rangle} \tag{7.7}$$

$$\frac{\neg \exists r \in rules(RS).\, \langle \xi, a \rangle \rightarrow_r \langle \xi', a' \rangle}{\langle \xi, a, RS \rangle \rightarrow_a \langle \xi, a, next(a, RS) \rangle} \tag{7.8}$$

A GWENDOLEN agent is a tuple $\langle ag, i, I, P, Pl, B, R, In, Out, S\rangle$ of an identifier, current intention, intention set, plan library, applicable plan set, belief base, rule base, inbox, outbox and sleep flag (more in this in section 7.5). The definition of *next* in (7.9) sometimes uses the current intention, i, and intention set, I, to compute the next reasoning stage. In these cases we represent the agent a as $\langle \ldots i \ldots\rangle$ or $\langle \ldots i, I \ldots\rangle$ as appropriate.

$$\begin{aligned}
next(\langle \ldots i, I \ldots\rangle, \mathbf{A}) &= \begin{cases} \mathbf{E} & i = [] \wedge \forall i' \in I.\ \textbf{is_suspended}(\mathbf{i}') \\ \mathbf{B} & i \neq [] \vee \exists i' \in I.\ \neg\textbf{is_suspended}(\mathbf{i}') \end{cases} \\
next(a, \mathbf{B}) &= \mathbf{C} \\
next(\langle \ldots i \ldots\rangle, \mathbf{C}) &= \begin{cases} \mathbf{E} & i = [] \\ \mathbf{D} & i \neq [] \end{cases} \\
next(a, \mathbf{D}) &= \mathbf{E} \\
next(a, \mathbf{E}) &= \mathbf{F} \\
next(a, \mathbf{F}) &= \mathbf{A}
\end{aligned} \tag{7.9}$$

where **is_suspended**(i) is true if the intention, i, is suspended.

7.5 Stage Rules: The Agent Reasoning Cycle

We represent an agent as a tuple $\langle ag, i, I, P, Pl, B, R, In, Out, S\rangle$ where:

- ag is a unique identifier for the agent (it's name);
- i is the current intention (see section 7.1); Note that there can be no current intention which we will indicate with the expression $i = null$.
- I is a stack of intentions $\{i, i', ..\}$;
- P is an ordered list of the agent's plans (see section 7.2);
- Pl is a set of currently applicable plans (see section 7.2);
- B is a set of the agent's beliefs which are pairs of ground first-order formulae and a string indicating the *source* of the belief. In GWENDOLEN all beliefs are automatically assigned the source `self` unless they are acquired by perception in which case they are assigned the source `percept`;
- R is a set of Prolog-style rules used in reasoning;
- In is the agent inbox. Elements of inbox have the form $\downarrow^{id,ilf} m$ where id is the identifier of the sender, ilf is the illocutionary force of the message and can be *tell*, *perform*, or *achieve*, and m is the message content, a ground first-order formula.
- Out is the agent outbox. Messages in this set have the format $\uparrow^{id,ilf} m$ where id is the identifier of the recipient, ilf is the illocutionary force and m is the message content, a ground first-order formula.

a	An action.
b	A belief.
$+b$	A belief addition.
$-b$	A belief removal.
$b\{source\}$	A belief, from source *source*.
$!_\tau g$	A goal of type τ.
$+!_\tau g$	A goal addition.
$-!_\tau g$	A goal drop.
$\times!_\tau g$	A goal which can't be planned.
`lock`	An lock.
`unlock`	An unlock.
$\uparrow^{ag,ilf}\ m$	A message m sent to ag.
$\downarrow^{ag,ilf}\ m$	A message m received from ag.
$\top$	An structure who's logical content is trivially true.
ϵ	A special marker indicating that some event has no plan yet.

Table 7.2: Notations for deed type checks

- S is a boolean indicating whether the agent should be *slept* by the scheduler or not.

In its initial state the current intention is *null*, the intention set consists of one intention for each of the initial goals provided by the programmer. These intentions are of the form $(\texttt{start}, +!_{\tau_g} g, \emptyset)$ where `start` is a special event used for intentions with no specific trigger. Its plan library is a set of plans provided by a programmer. The applicable plans are empty. The belief base and rule base are as defined by the programmer. The inbox and outbox are empty and the sleep flag is false.

Many of the transition rules make a check on a deed to see what type it is (e.g. the addition of a belief, the deletion of a goal). We represent these checks implicitly using the notation shown in table 7.2. Many of the rules also check intentions for various properties and manipulate them. Table 7.3 summarises various operations on intentions that are used in the rules.

It is generally unwieldy to present the full agent tuple in the description of a transition rule. As a result we restrict ourselves to presenting only those parts of the intention that are changed by the rule as we did in (7.9).

We now discuss each stage of the reasoning cycle in turn.

7.5.1 Stage A

Stage A of the GWENDOLEN reasoning cycle consists of a list of three rules which are focused around managing intention selection:

[`select_intention`, `sleep`, `drop_intention`]

Notation	Description
$\mathtt{U}_\theta$	Compose a unifier with the top unifier on the intention.
$\mathtt{empty}(i)$	The deed stack of the intention is empty.
$\mathtt{events}(i)$	The stack of events associated with intention i.
$\mathtt{hd}_e(i)$	The top event on the intention.
$\mathtt{hd}_d(i)$	The top deed on the intention.
$\theta^{\mathtt{hd}(i)}$	The top unifier on the intention.
@	Add a new event, deed stack, and unifier to the top of the intention.
$;_p$	Add a new event, deed, and unifier as the top row of the intention.
$\mathtt{tl}_i(i)$	Drop the top row of the intention.
$\mathtt{drop}_E(e, i)$	Drop all rows in the intention above and including the first appearance of e as a trigger.
$\mathtt{lock}(i)$	Mark the intention as locked.
$\mathtt{locked}(i)$	The intention is locked.
$\mathtt{suspend}(i)$	Mark the intention as suspended.
$\mathbf{is\_suspended}(i)$	The intention is suspended.
$\mathtt{unlock}(i)$	Mark the intention as unlocked.

Table 7.3: Operations on Intentions

Select Intention (`select_intention`)

$$\frac{\neg\mathtt{empty}(i) \quad \neg\mathtt{locked}(i) \quad \exists i'' \in I \cup \{i\}.\, \neg\mathbf{is\_suspended}(\mathbf{i}'') \quad \mathcal{S}_{\text{int}}(I \cup \{i\}) = (i', I') \quad \mathtt{hd}_e(i') \neq -!_{\tau_g} g \vee \mathtt{hd}_d(i') \neq \epsilon}{\langle \xi, \langle \ldots i, I \ldots \rangle\rangle \rightarrow_{\mathtt{select\_intention}} \langle \xi, \langle \ldots i', I' \ldots \rangle\rangle} \tag{7.10}$$

$$\frac{\neg\mathtt{empty}(i) \quad \mathtt{locked}(i) \quad \mathtt{hd}_e(i) \neq -!_{\tau_g} g \vee \mathtt{hd}_d(i) \neq \epsilon \quad \exists i'' \in I \cup \{i\}.\, \neg\mathbf{is\_suspended}(\mathbf{i}'')}{\langle \xi, \langle \ldots i, I \ldots \rangle\rangle \rightarrow_{\mathtt{select\_intention}} \langle \xi, \langle \ldots i, I \ldots \rangle\rangle} \tag{7.11}$$

where $\mathtt{empty}(i)$ is true if intention i has an empty deed stack, $\mathtt{locked}(i)$ is true if intention i is locked, and $\mathbf{is\_suspended}(i)$ is true if intention i is suspended. Table 7.3 summarises all the operations on intentions.

This rule has two cases, one for when the current intention isn't locked and one for when it is. When the intention isn't locked the system uses the application specific selection function $\mathcal{S}_{\text{int}}$ to pick a new current intention (by default this treats the intention set I as a LIFO queue and selects the first unsuspended intention from the queue). The rule is inapplicable if the current intention is empty or the selected intention's trigger is a drop goal event.

Sleep (`sleep`)

$$\frac{(i = null \vee \mathtt{empty}(i) \vee \mathbf{is\_suspended}(i)) \quad \forall \mathbf{i}' \in I.\, \mathbf{is\_suspended}(\mathbf{i}')}{\langle \xi, \langle \ldots i, I, \ldots S \rangle\rangle \rightarrow_{\mathtt{sleep}} \langle \xi, \langle \ldots, i, I, \ldots \top \rangle\rangle} \tag{7.12}$$

Table 7.3 summarises all the operations on intentions such as `empty` etc,.

This rule sets an agent's sleep flag if all its intentions are empty or suspended. The agent will then be marked as sleeping by the scheduler once the reasoning cycle is concluded.

Drop Intention (`drop_intention`)

$$\frac{i \neq null \quad \texttt{empty}(i) \quad I \neq \emptyset \quad \mathcal{S}_{\text{int}} I = (i', I')}{\langle \xi, \langle \ldots i, I \ldots \rangle \rangle \rightarrow_{\texttt{drop\_intention}} \langle \xi, \langle \ldots i', I' \ldots \rangle \rangle} \tag{7.13}$$

Table 7.3 summarises all the operations on intentions such as `empty` etc,.

This rule drops the intention i if it is empty and selects a new current intention from the intention set. The additional $i \neq null$ is necessary since a few rules can leave the agent state with no current intention.

7.5.2 Stage B

Stage B of the GWENDOLEN reasoning cycle consists of a list of two rules based on generating a set of applicable plans: [`generate_plan`, `no_plan`]

Generate Plan (`generate_plan`)

$$\frac{\mathbf{appPlans}(i) \neq \emptyset}{\langle \xi, \langle \ldots i, Pl \ldots \rangle \rangle \rightarrow_{\texttt{generate\_plan}} \langle \xi, \langle \ldots i, \mathbf{appPlans}(i) \ldots \rangle \rangle} \tag{7.14}$$

appPlans is as described in section 7.2.

No Applicable Plans (`no_plan`)

$$\frac{\mathbf{appPlans}(i) = \emptyset \quad \texttt{hd}_e(i) = +!_\tau g}{\langle \xi, \langle \ldots i, Pl \ldots \rangle \rangle \rightarrow_{\texttt{no\_plan}} \langle \xi, \langle \ldots i, [(\texttt{x}!_\tau g, [\epsilon], \theta^{\texttt{hd}(i)})] \ldots \rangle \rangle} \tag{7.15}$$

$$\frac{\mathbf{appPlans}(i) = \emptyset \quad \neg\texttt{hd}_e(i) = +!_\tau g}{\langle \xi, \langle \ldots i, Pl \ldots \rangle \rangle \rightarrow_{\texttt{no\_plan}} \langle \xi, \langle \ldots i, [(\texttt{hd}_e(i), [], \emptyset)] \ldots \rangle \rangle} \tag{7.16}$$

appPlans(i) is empty if there is no plan applicable to the current intention. This rule differentiates between whether the intention trigger is a goal commitment (in which case the rule creates an applicable plans consisting of an unplanned "problem goal" event ($\texttt{x}!_\tau g$) (which might, for instance, be responded to by suspending the intention until the agent's beliefs have changed and some plan does become applicable). Otherwise it generates an applicable plan with an empty deed stack. This will have the effect of removing the top row of the intention and replacing it by nothing – i.e., it ignores the event that had no applicable plan for handling it. The reasoning behind this is that such events (notifications of beliefs acquired or dropped generally only require a planning response in special cases and can normally be ignored).

7.5.3 Stage C

Stage C of the GWENDOLEN reasoning cycle consists of a list of a single rule for modifying the current intention according to the applicable plan: [apply_plan]

Apply Plan (apply_plan)

$$\frac{(e, Ds, \theta) = \mathcal{S}_{\text{plan}}(Pl)}{\langle \xi, \langle \ldots i \ldots Pl \ldots \rangle \rangle \rightarrow \langle \xi, \langle \ldots (e, Ds, \theta) \mathbin{@} \mathtt{tl}_i(i) \ldots \emptyset \ldots \rangle \rangle} \tag{7.17}$$

where $\mathtt{tl}_i(i)$ represents intention, i, with its top row removed and $(e, Ds, \theta) \mathbin{@} \mathtt{tl}_i(i)$ represents the applicable plan (e, Ds, θ) expanded and added to the top of the intention, i in place of its top row as described in section 7.2. Table 7.3 summarises all the operations on intentions such as empty etc,.

This rule selects a plan from the agent's applicable plans as determined by the application specific $\mathcal{S}_{\text{plan}}$ (by default this is the first applicable plan found in the plan library and, where a unifier is required, this is the first returned by checking against the agents internal state (this lists beliefs and goals, etc., in alphabetical order)). The plan is represented as a tuple of the trigger event, the plan's deed stack and unifier. The top row of the current intention is dropped and the applicable plan is "glued" in its place.

7.5.4 Stage D

Stage D of the GWENDOLEN reasoning cycle consists of a list of rules for processing the top deed on the current intention:

[empty, add_achieve_goal, add_perform_goal, drop_goal, add_belief, drop_belief, lock_unlock, wait_for, problem_goal, action, send, null]

Handle Empty Deed Stack (empty)

$$\frac{\mathtt{empty}(i)}{\langle \xi, \langle \ldots i \ldots \rangle \rangle \rightarrow_{\mathtt{empty}} \langle \xi, \langle \ldots i \ldots \rangle \rangle} \tag{7.18}$$

Table 7.3 summarises all the operations on intentions such as empty etc,.

This rule does nothing if the current intention's deed stack is empty (which can occur if there is no plan for handling the intention's trigger event). This leaves the intention unchanged and it will be removed during the select intention phase (Stage A).

Handle Add Achieve Goal (add_achieve_goal)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +!_a g \quad B, R \models g, \theta_g}{\begin{array}{c}\langle \xi, \langle \ldots i \ldots B, R \ldots \rangle \rangle \\ \rightarrow_{\mathtt{add\_achieve\_goal}} \\ \langle \xi, \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta_g \ldots B, R \ldots \rangle \rangle\end{array}} \tag{7.19}$$

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +!_a g \quad B, R \not\models g}{\begin{array}{c}\langle \xi, \langle \ldots i \ldots B, R \ldots \rangle\rangle \\ \rightarrow_{\mathtt{add\_achieve\_goal}} \\ \langle \xi, \langle \ldots (+!_a g, \epsilon, \theta^{\mathtt{hd}(i)});_p i \ldots B, R \ldots \rangle\rangle\end{array}} \tag{7.20}$$

where $B, R \models g, \theta_g$ means that the formula g (which is the goal with the top unifier from the intention applied to it) follows using Prolog-style reasoning from the agent's belief base when the additional unifier θ_g is applied. $\mathtt{tl}_i(i)\mathtt{U}_\theta\theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\mathtt{tl}_i(i)$. $(e, d, \theta);_p i$ represents the addition of a row (e, d, θ) to the top of an intention i. Table 7.3 summarises all the operations on intentions such as `empty` etc,.

GWENDOLEN recognises two types of goal, *achieve* goals and *perform* goals (goal types a and p respectively). This rule handles the commitment to an achieve goal. An achieve goal is one that triggers a plan if it not already believed but does no more than set a unifier if it is. If it is to trigger a plan, then we register the commitment to planning the goal as an event on the top of the intention stack. In this case the top row of the intention is not dropped so the deed intending a commitment to the goal remains. This means that if, after execution of the plan, the goal is not achieved then it will be replanned.

Handle Add Perform Goal (`add_perform_goal`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +!_p g}{\begin{array}{c}\langle \xi, \langle \ldots i \ldots \rangle\rangle \\ \rightarrow_{\mathtt{add\_perform\_goal}} \\ \langle \xi, \langle \ldots (+!_p g, \epsilon, \theta^{\mathtt{hd}(i)});_p (\mathtt{hd}_e(i), null, \theta^{\mathtt{hd}(i)});_p \mathtt{tl}_i(i) \ldots \rangle\rangle\end{array}} \tag{7.21}$$

where $(e, d, \theta);_p i$ represents the addition of a row (e, d, θ) to the top of an intention i. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

Perform goals always trigger planning but are not replanned if they fail to achieve some state of the world. This being the case we replace the top deed (the request to commit to the goal) on the intention with *null* so that this is automatically processed once the system reaches that row of the intention. We then add a new top row with the trigger event of the new goal and a no plan yet deed.

Handle Drop Goal (`drop_goal`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = -!_{\tau_g} g \quad \exists e \in \mathtt{events}(i).\mathbf{unify}(e, +!_{\tau_g} g)}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{drop\_goal}} \langle \xi, \langle \ldots \mathtt{tl}_i(\mathtt{drop}_E(e, i)) \ldots \rangle\rangle} \tag{7.22}$$

where $\mathbf{unify}(e_1, e_2)$ indicates that two events can be unified. $\mathtt{drop}_E(e, i)$ is a function that recurses through an intention dropping every row after the first occurrence of e – i.e. it prunes the intention back to the point where the event

first occurred. Table 7.3 summarises all the operations on intentions such as `events` etc,.

This rule searches the current intention for the most earliest add goal event that unifies with the goal to be dropped and then deletes all rows on the intention above that. It then deletes the new top row which will be he one that contains the instruction to commit to the goal (if an achieve goal) or *null* (if a perform goal).

Handle Add Belief (`add_belief`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = +b}{\begin{array}{c}\langle \xi, \langle \ldots i, I, B \ldots \rangle\rangle \rightarrow_{\mathtt{add\_belief}} \\ \langle \xi, \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)}, \mathbf{unsuspend}(I, b) \cup \mathtt{new}(+b, \epsilon, \emptyset), \\ B \cup \{b\}, \ldots \rangle\rangle\end{array}} \tag{7.23}$$

where $\mathbf{unsuspend}(I, b)$ unsuspends all suspended intentions in I that are waiting for b to become true. $\mathtt{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

This rule adds new belief to the belief base and a new intention noting the appearance of the new belief. At the same time it unsuspends all intentions which are waiting for b to be achieved as part of their suspend condition.

Handle Drop Belief (`drop_belief`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = -b \quad B^1 = \{b' | b' \in B \wedge \mathbf{unify}(b', b)\}}{\begin{array}{c}\langle \xi, \langle \ldots i, I, B \ldots \rangle\rangle \rightarrow_{\mathtt{drop\_belief}} \\ \langle \xi, \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)}, I \cup \mathtt{new}(-b, \epsilon, \emptyset), B \backslash B^1, \ldots \rangle\rangle\end{array}} \tag{7.24}$$

where $\mathbf{unify}(b', b)$ means that b' and b unify with each other. $\mathtt{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

This rule drops a belief from the belief base. At the same time it generates a new intention containing the event that the belief has been dropped. Appropriate handling of this event can allow the agent to form plans in reaction to it.

Handle Lock and Unlock (`lock_unlock`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = \mathtt{lock}}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{lock\_unlock}} \langle \xi, \langle \ldots, \mathtt{lock}(\mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)}) \ldots \rangle\rangle} \tag{7.25}$$

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = \mathtt{unlock}}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{lock\_unlock}} \langle \xi, \langle \ldots \mathtt{unlock}(\mathtt{tl}_i(i)\mathtt{U}_\theta\theta^{\mathtt{hd}(i)}) \ldots \rangle\rangle} \tag{7.26}$$

Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

This allows an intention to be "locked" as the current intention, for instance to allow a complete sequence of belief changes be processed before any other reasoning takes place. Once finished the intention has to be unlocked.

Handle Wait For

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *\ldots b \quad B, R \models b, \theta_b}{\begin{array}{c}\langle \xi, \langle \ldots i \ldots B, R \ldots \rangle\rangle \rightarrow_{\mathtt{wait\_for}} \\ \langle \xi, \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, (\theta^{\mathtt{hd}(i)} \cup \theta_b) \ldots B, R \ldots \rangle\rangle\end{array}} \tag{7.27}$$

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *\ldots b \quad B, R \not\models b, \theta_b \quad \exists i' \in I.\, \neg\mathbf{is\_suspended}(\mathbf{i}')}{\begin{array}{c}\langle \xi, \langle \ldots i, I, B, R \ldots \rangle\rangle \rightarrow_{\mathtt{wait\_for}} \\ \langle \xi', \langle \ldots \mathtt{suspend}(i\theta^{\mathtt{hd}(i)}), I, B, \mathtt{R} \ldots \rangle\rangle\end{array}} \tag{7.28}$$

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = *\ldots b \quad B, R \not\models b, \theta_b \quad \forall i' \in I.\mathbf{is\_suspended}(\mathbf{i}')}{\begin{array}{c}\langle \xi, \langle \ldots i, I, B \ldots In \ldots S \rangle\rangle \rightarrow_{\mathtt{wait\_for}} \\ \langle \xi', \langle \ldots null, I \cup \{\mathtt{suspend}(i\theta^{\mathtt{hd}(i)})\}, B, \mathtt{R} \ldots \top \rangle\rangle\end{array}} \tag{7.29}$$

where $B, R \models b, \theta_b$ means that the formula b follows using Prolog-style reasoning from the agent's belief base and Prolog rule-base when the additional unifier θ_b is applied. $\mathtt{tl}_i(i)\, \mathtt{U}_\theta\, (\theta^{\mathtt{hd}(i)} \cup \theta_b)$ indicates the union of unifier $(\theta^{\mathtt{hd}(i)} \cup \theta_b)$ with the unifier on the top of the intention $\mathtt{tl}_i(i)$. $\mathtt{suspend}(i)$ suspends an intention. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

If an intention is waiting for some belief, b, to become true then if that belief is now true the intention continues processing. Otherwise the intention is suspended. If all intentions are suspended then the agent is told to sleep at the next opportunity.

Ignore Unplanned Problem Goal (`problem_goal`)

$$\frac{\mathtt{hd}_e(i) = \mathtt{x!}_{\tau_g} g \quad \mathtt{hd}_d(i) = \epsilon}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{problem\_goal}} \langle \xi, \langle \ldots i \ldots \rangle\rangle} \tag{7.30}$$

Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

This rule ignores an unplanned problem goal. It simply does nothing but allows the reasoning cycle of the agent to continue processing on the assumption that planning of the goal may become possible later.

Handle General Action (`action`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi \xrightarrow{\mathbf{do}(a)} \xi' \quad \xi.\mathbf{do}(a) = \theta_a \quad a \neq \uparrow^{ag,ilf} m}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{action}} \langle \xi', \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, (\theta^{\mathtt{hd}(i)} \cup \theta_a) \ldots \rangle\rangle} \tag{7.31}$$

$$\frac{\begin{array}{c}\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi \xrightarrow{\mathbf{do}(a)} \xi' \quad \neg\xi.\mathbf{do}(a) \quad \mathtt{hd}_e(i) = +!_{\tau_g} g \\ a \neq \uparrow^{ag,ilf} m\end{array}}{\langle \xi, \langle \ldots i \ldots \rangle\rangle \rightarrow_{\mathtt{action}} \langle \xi', \langle \ldots (\mathtt{x!}_{\tau_g} g, \epsilon, \theta^{\mathtt{hd}(i)} \cup \theta_a);_p i \ldots \rangle\rangle} \tag{7.32}$$

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = a \quad \xi \xrightarrow{\mathbf{do}(a)} \xi' \quad \neg\xi.\mathbf{do}(a) \quad \mathtt{hd}_e(i) \neq +!_{\tau_g} g \quad a \neq \uparrow^{ag,ilf} m}{\langle \xi, \langle \ldots i \ldots \rangle \rangle \rightarrow_{\mathtt{action}} \langle \xi', \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)} \ldots \rangle \rangle} \tag{7.33}$$

where $\xi.\mathbf{do}(a\theta^{\mathtt{hd}(i)}) = \theta_a$ means that the environment successfully executes a returning unifier θ_a. $\mathtt{tl}_i(i)\,\mathtt{U}_\theta\,\theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\mathtt{tl}_i(i)$. $(e, d, \theta);_p i$ represents the addition of a row (e, d, θ) to the top of an intention i. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

In this rule, the agent attempts the action (unless it is a send action – $\uparrow^{ag,ilf} m$). If the action succeeds it returns a unifier and the environment updates. Otherwise, if the trigger event at the top of the intention is a goal then this is generates a problem goal event for handling by some plan.

Handle Send Action (`send`)

$$\frac{\xi \xrightarrow{\mathbf{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \quad \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = \uparrow^{ag',ilf} m \quad \xi.\mathbf{do}(\uparrow^{ag',ilf} m_{ag}) = \theta_a}{\begin{array}{c}\langle \xi, \langle ag, i, I \ldots Out \ldots \rangle \rangle \rightarrow_{\mathtt{send}} \\ \langle \xi', \langle ag, \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, (\theta^{\mathtt{hd}(i)} \cup \theta_a), I \cup \{\mathtt{new}(+\uparrow^{ag',ilf} m, \epsilon, \emptyset)\}, \\ \ldots Out \cup \{\uparrow^{ag',ilf} m\} \ldots \rangle \rangle\end{array}} \tag{7.34}$$

$$\frac{\begin{array}{c}\xi \xrightarrow{\mathbf{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \\ \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = \uparrow^{ag',ilf} m \quad \neg\xi.\mathbf{do}(\uparrow^{ag',ilf} m_{ag}) \quad \mathtt{hd}_e(i) = +!_{\tau_g} g\end{array}}{\begin{array}{c}\langle \xi, \langle ag, i, I \ldots Out \ldots \rangle \rangle \rightarrow_{\mathtt{send}} \\ \langle \xi', \langle ag, (\mathtt{x}!_{\tau_g} g, \epsilon, \theta^{\mathtt{hd}(i)} \cup \theta^{\mathtt{hd}(i)});_p i, I \ldots Out \ldots \rangle \rangle\end{array}} \tag{7.35}$$

$$\frac{\begin{array}{c}\xi \xrightarrow{\mathbf{do}(\uparrow^{ag',ilf} m_{ag})} \xi' \\ \mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = \uparrow^{ag',ilf} m \quad \neg\xi.\mathbf{do}(\uparrow^{ag',ilf} m_{ag}) \quad \neg\mathtt{hd}_e(i) = +!_{\tau_g} g\end{array}}{\begin{array}{c}\langle \xi, \langle ag, i, I \ldots Out \ldots \rangle \rangle \rightarrow_{\mathtt{send}} \\ \langle \xi', \langle ag, \mathtt{tl}_i i\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)}, I \ldots Out \ldots \rangle \rangle\end{array}} \tag{7.36}$$

where $\xi.\mathbf{do}(\uparrow^{ag',ilf} m_{ag})$ is the environment executing the sending of a message, m, from ag to ag' with illocutionary force ilf. $\mathtt{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier (in this case the event is the sending of a message to ag'). $\mathtt{tl}_i(i)\,\mathtt{U}_\theta\,\theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\mathtt{tl}_i(i)$. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

This rule behaves much as the rule for handling general actions with the exception that when a send action succeeds a new intention is generated registering the event that a message was sent and the message itself is added to the agent's outbox.

Handle Null (`null`)

$$\frac{\mathtt{hd}_d(i)\theta^{\mathtt{hd}(i)} = null}{\langle \xi, \langle \ldots i \ldots \rangle \rangle \rightarrow_{\mathtt{null}} \langle \xi, \langle \ldots \mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta^{\mathtt{hd}(i)} \ldots \rangle \rangle} \tag{7.37}$$

where $\mathtt{tl}_i(i)\, \mathtt{U}_\theta\, \theta_g$ indicates the union of unifier θ_g with the unifier on the top of the intention $\mathtt{tl}_i(i)$. Table 7.3 summarises all the operations on intentions such as $\theta^{\mathtt{hd}(i)}$ etc,.

The null action is used as a place holder to note, when a perform goal has been committed to, a record of the relevant trigger event in an intention stack. This rule simply ignores the null action when it is encountered and deletes that row from the intention.

7.5.5 Stage E

Stage E of the GWENDOLEN reasoning cycle consists of a list of a single rule for handling perception: [`perceive`]

Perceive

$$\frac{\begin{array}{c}\xi \xrightarrow{\mathbf{Percepts}(ag)}_{\xi} \xi_1 \quad \xi_1 \xrightarrow{getmessages(ag)}_{\xi} \xi' \\ P = \xi.\mathbf{Percepts}(ag) \\ OP = \{b \mid b \in B \backslash P \wedge source\_of(b) = \mathtt{percept}\} \\ P \backslash B \cup OP \cup \xi.getmessages(ag) \neq \emptyset \end{array}}{\begin{array}{c}\langle \xi, \langle \ldots I, B \ldots In \ldots S \rangle \rangle \rightarrow_{\mathtt{perceive}} \\ \langle \xi', \langle \ldots \\ I \cup \{\mathtt{new}(\mathtt{start}, +b, \emptyset) \mid b \in P \backslash B\} \cup \{\mathtt{new}(\mathtt{start}, -b, \emptyset) \mid b \in OP\}, \\ B \ldots In \cup \xi.getmessages(ag) \ldots \top \rangle \rangle \end{array}} \tag{7.38}$$

$$\frac{\begin{array}{c}\xi \xrightarrow{\mathbf{Percepts}(ag)}_{\xi} \xi_1 \quad \xi_1 \xrightarrow{getmessages(ag)}_{\xi} \xi' \\ P = \xi.\mathbf{Percepts}(ag) \\ OP = \{b \mid b \in B \backslash P \wedge source\_of(b) = \mathtt{percept}\} \\ P \backslash B \cup OP \cup \xi.getmessages(ag) = \emptyset \end{array}}{\langle \xi, \langle \ldots I, B \ldots In \ldots \rangle \rangle \rightarrow_{\mathtt{perceive}} \langle \xi', \langle \ldots I, B \ldots In \ldots \rangle \rangle} \tag{7.39}$$

where $\xi.\mathbf{Percepts}(ag)$ returns a set of new beliefs to the agent which are all annotated as coming from the source `percept`. $source\_of(b)$ returns the source of a belief b. $\xi.getmessages(ag)$ returns a set of new messages to the agent. $\mathtt{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. In this case the event is a special distinguished event `start` which is used to indicate an intention with no trigger.

This rule adds all messages to the inbox. It also creates new intentions, each triggered by the event of acquiring or losing a percept. A key part of the working of the rule depends on AIL's annotation of all beliefs in the belief base with a source and its use of a special annotation for beliefs whose source is perception. If some change is bought about either to the agent's inbox or to its intentions

then the agent's sleep flag is set to true (i.e., the agent will not sleep at the end of this reasoning cycle).

Note that in the EASS variant of GWENDOLEN the perception rule also updates the belief base directly, unlike this rule which creates intentions to update the belief base and leaves these to later reasoning cycles for execution.

7.5.6 Stage F

Stage F of the GWENDOLEN reasoning cycle consists of a list of a single rule for processing messages in the inbox: [`messages`]

Handle Messages (`messages`)

$$\frac{}{\langle \xi, \langle \ldots I \ldots In \ldots \rangle \rangle \rightarrow_{\texttt{messages}} \langle \xi, \langle \ldots I \cup \{\texttt{new}(+received(ag, ilf, m), \epsilon, \emptyset) \mid \downarrow^{ag,ilf} \ m \in In\} \ldots [] \ldots \rangle \rangle} \quad (7.40)$$

where $\texttt{new}(e, d, \theta)$ creates a new intention from an event, deed and unifier. In this case the event is a belief that the agent has received message m from agent, ag with illocutionary force, ilf. It is up to the programmer to decide how messages should be handled, there is no default mechanism for handling messages of any particular illocutionary force (unlike many BDI languages which give a specific semantics to such constructs).

This rule does not poll the environment for messages. It takes all messages currently in an agent's inbox and converts them to intentions (triggered by a perception that the message has been received), emptying the inbox in the process. It should be noted that it does not store the message anywhere once the inbox is emptied. It assumes that some plan will act appropriately to the message received event. If this does not happen then the message content may be lost.

Chapter 8

The EASS Variant of the Gwendolen Programming Language

This chapter contains a set of tutorials on the EASS variant of the Gwendolen programming language which is intended for programming agent-based hybrid autonomous systems.

8.1 Tutorial 1 – The EASS variant of Gwendolen

This is the first in a series of tutorials on the use of the EASS variant of the Gwendolen language that was first developed as part of the Engineering Autonomous Space Software project. The EASS variant is adapted for use with physical systems and simulations, such as mobile robots, satellites and unmanned aircraft. This tutorial covers the basic concepts behind the EASS variant and its differences to the Gwendolen language.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/eass/tutorials/tutorial1`.

The tutorial assumes familiarity with the Gwendolen programming language.

8.1.1 Abstraction and Reasoning Engines

Figure 8.1 shows the typical architecture of an EASS Agent. The agent is actually a pair of agents, the *reasoning engine* which is responsible for complicated reasoning tasks, and the *abstraction engine* which is responsible for processing and filtering incoming perceptions so only those actually needed for reasoning are passed on to the reasoning engine itself.

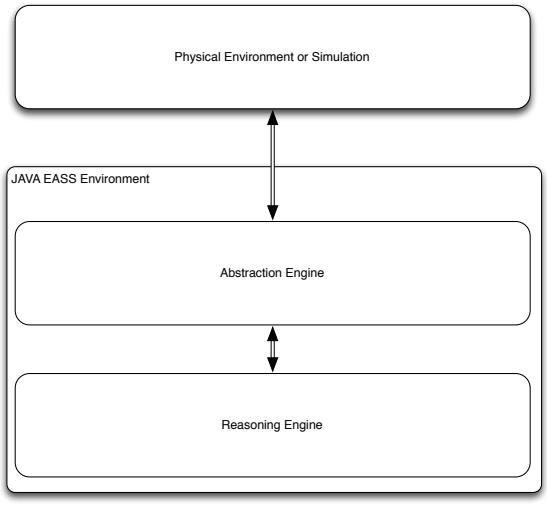

Figure 8.1: The Architecture of an EASS Agent

The reasons for this separation are primarily driven by the observation that BDI agents are unable to process incoming perceptions from real or simulated sources fast enough and so they get "clogged" by an ever increasing number of intentions to do with perception processing.

The theoretical underpinnings of this architecture are described in [Dennis et al., 2010, Dennis et al., 2016]. The key points are that the *reasoning engine* does not interact with the physical world (or a simulation) at all. It gains perceptions via *shared beliefs* which are communicated with the abstraction engine via the Java EASS environment. Similarly most of its actions (ideally all) are communicated to the abstraction engine which then reifies them (e.g., adding more low level detail that may be required by the physical system or simulation to actually enact the action). As a rule of thumb while perceptions and actions, as used by the physical system or simulation generally involve numeric values, reasoning generally uses logical ("yes/no") information and outputs simple non-numeric commands. Therefore the abstraction engine should be responsible for converting numeric data ("distance = 5.4m") into logical statements ("too close") and converting simple commands ("slow down") into numerical instructions ("apply a deceleration of $-1m/s^2$"). This is only a rule of thumb and the reality is that a certain amount of experimentation is often required to balance a system appropriately so that reasoning happens fast enough to adequately control the physical system.

In order for this to work GWENDOLEN's reasoning cycle was adapted slightly, a set of dedicated actions were introduced for handling shared beliefs and delegated actions, and some new constructs were added to the language.

8.1.2 Key Differences

Perception Processing

In the GWENDOLEN language incoming perceptions are converted into intentions which contain a deed to add the perception to the belief base. In theory this gives the agent more control over the contents of its beliefs (although in practice no use has ever been made of this). However a side effect is that it takes the agent two turns of the reasoning cycle to convert a perception into a belief and this slowed down the processing of perceptions.

In the EASS variant, therefore, new perceptions are placed directly into the agent's belief base during the perception stage of the reasoning cycle.

Identifying Abstraction and Reasoning Engines

Since each agent is, in fact, a pair of agents, it is necessary to identify and link the abstraction and reasoning engines. This is done by starting the abstraction engine with the line

```
:abstraction: agentname
```

instead of

```
:name: agentname
```

which is reserved as the start of the reasoning engine code. So long as the two agents have the same name then the environment will link them.

Shared Beliefs

The reasoning engine does not receive percepts from the outside world but only via a *shared belief* set. An abstraction engine may get perceptions both form the outside world and from the shared beliefs.

EASS environments support this communication via support for two dedicated actions, assert_shared(B) and remove_shared(B) which can be used to assert and remove the shared belief, B.

Both the abstraction and reasoning engine may use these commands.

Perf

The reasoning engine may also request the abstraction engine to reify an action to be sent to some external system. It does this via the dedicated actions, `perf`. This sends a message to the abstraction engine asking it to adopt a perform goal.

This means that abstraction engines need to implement plans for handling perform messages.

8.1.3 Example

Example 24 shows a simple EASS program to control a car by making it accelerate up to the speed limit and then maintain that speed. Lines 3-28 are the abstraction engine, and lines 30-46 are the reasoning engine.

As an initial belief the abstraction engine has that the speed limit on the road is 5. Every time the perception, yspeed(X) comes in (lines 20-23) the abstraction compares this to the speed limit and, if appropriate asserts a shared belief (NB., we are using $+_\Sigma(B)$ as shorthand for assert_shared (B) and $-_\Sigma(B)$ as shorthand for remove_shared(B)).

Lines 13-15 are the standard plans for handling messages. It is important that the abstraction engine has these to it correctly handles `perf` requests from the reasoning engine.

If the reasoning engine has a goal to reach the speed limit (lines 41-43) then it requests the abstraction engine to perform accelerate . This is passed on directly to the environment (line 25). Once the reasoning engine believes the speed limit is reached (lines 45 and 46) then it requests the abstraction engine to perform maintain_speed.

Lastly, in order to allow time for the simulation to start, a perception, started is used. When the abstraction engine perceives this (lines 17 and 18) it asserts the shared belief start which causes the reasoning engine to adopt the goal of reaching the speed limit (lines 38 and 39).

The environment passes on requests for acceleration etc., to the simulator and reports on the simulated speed and position using perceptions.

Running the Example

The example uses `MotorwayMain` which you can find in the directory `src/examples/eass/tutorials/motorwaysim`. This must be run as a separate java program and must be started before the EASS program is run. When it starts you should see message:

```
Motorway Sim waiting Socket Connection
```

Now run the EASS program as normal for AIL programs. You will find the AIL configuration file in the tutorial directory. You should see the window shown in figure 8.2 (NB. You may need to move other windows out of the way to find it!). Click on start and you should see the car accelerate up to a speed of 5.

8.1.4 Exercise

`eass.tutorials.tutorial1.CarOnMotorwayEnvironment` provides four actions to the abstraction engine.

accelerate Accelerates the car.

decelerate Decelerates the car (if the car reaches a speed of 0 it stops).

Example 24

```
EASS

:abstraction: car

:Initial Beliefs:

speed_limit(5)

:Initial Goals:

:Plans:
/* Default plans for handling messages */
+.received(:tell, B): {True} ← +B;
+.received(:perform, G): {True} ← +!G [perform];
+.received(:achieve, G): {True} ← +!G [achieve];

+started : {True} ←
        +Σ(start);

+yspeed(X) : {B speed_limit(SL), SL < X} ←
        +Σ(at_speed_limit);
+yspeed(X) : {B speed_limit(SL), X < SL} ←
        −Σ(at_speed_limit);

+! accelerate [perform] : {B yspeed(X)} ← accelerate;
+! accelerate [perform] : {~B yspeed(X)} ←
     print("Waiting for Simulator to Start");
+! maintain_speed [perform] : {True} ← maintain_speed;

:name: car

:Initial Beliefs:

:Initial Goals:

:Plans:

+start: {True} ←
        +!at_speed_limit[achieve];

+! at_speed_limit [achieve] : {True} ←
        perf(accelerate),
  *at_speed_limit;

+at_speed_limit: {True} ←
        perf(maintain_speed);
```

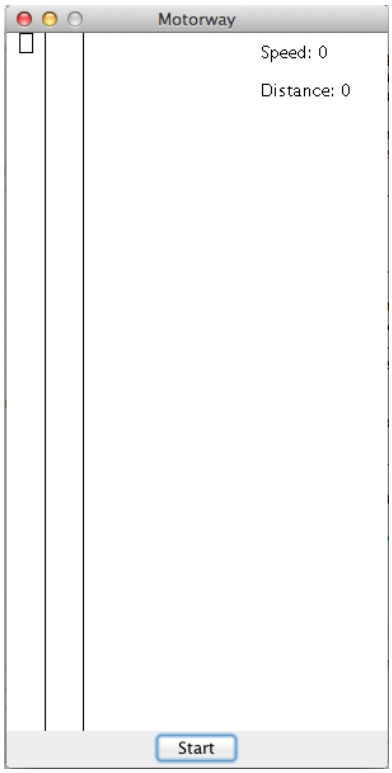

Figure 8.2: The Motorway Simulator

maintain_speed Maintains the speed of the car.

calculate_totaldistance(D) unifies `D` with the total distance the car has travelled.

The perceptions it sends to the abstraction engine are:

xpos(X) `X` is the x position of the car.

ypos(X) `X` is the y position of the car.

xspeed(X) `X` is the speed of the car in the x direction.

yspeed(X) `X` is the speed of the car in the x direction.

started The simulation has started.

The x and y positions of the car reset to 0 each time the car loops around the simulator window.

Exercise 1

Adapt `car.eass` so that the reasoning engine prints out the total distance travelled when it reaches the speed limit.

Exercise 2

Extend `car.eass` so that it accelerates to the speed limit, then continues until it has reached a total distance of 600 metres and then decelerates.

Hint. The solution does this by having the reasoning engine request an alert at 600 units. The abstraction engine then calculates the total distance each time `ypos` updates until this is reached at which point it alerts the reasoning engine. There are lots of other ways to do this but this solution maintains the idea that the reasoning engine makes decisions while the abstraction engine processes data.

Sample answers for the exercises can be found in `eass/examples/tutorials/tutorial1/answers`.

8.2 Tutorial 2 – Environments for the EASS variant of Gwendolen

This is the second in a series of tutorials on the use of the EASS variant of the Gwendolen language. This tutorial covers creating environments for agent programs by extending the `eass.mas.DefaultEASSEnvironment` class.

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/eass/tutorials/tutorial2`.

The tutorial assumes a good working knowledge of Java programming, and some basic understanding of sockets. It also assumes the reader is familiar with the creation of AIL environments (see section 4.2).

8.2.1 The Default EASS Environment and the EASSEnv Interface

All environments for use with EASS must implement a java interface `eass.mas.EASSEnv`. This extends `ail.mas.AILEnv` (discussed in section 4.2) and `ajpf.MCALJobber` which specifies the functionality required for AJPF to include the environment in the scheduler. One of the key features of EASS environments is that they are *dynamic* – that is things may occur in the environment which are not caused by the agents. The AJPF framework uses a scheduler to switch between agents and any other *jobbers* known to the scheduler. When a dynamic environment is used the scheduler switches between agents and the environment. In fact there are a number of schedulers that can be used. These are discussed in section 4.3. As well as the functionality required by the two interfaces it extends, `eass.mas.EASSEnv` requires some extra functionality to support the EASS GWENDOLEN variant, particularly managing the links between abstraction and reasoning engines and shared beliefs.

`eass.mas.DefaultEASSEnvironment` provides a basic level implementation of all these methods, so any environment that extends it only has to worry about those aspects particular to that environment. Typically this is just the way that actions performed by the agents are to be handled and the way perceptions may change in between agent actions. `ail.mas.DefaultEASSEnvironment` also provides a set of useful methods for handling changing the perceptions available to the agent that can then be used to program these action results. `eass.mas.DefaultEASSEnvironment` extends `ail.mas.DefaultEnvironment` (see section 4.2) so all the methods available in that class are also available to classes that subclass `eass.mas.DefaultEASSEnvironment`.

8.2.2 A Survey of some of Default EASS Environment's Methods

We note here some of the more useful methods made available by the Default Environment before we talk about implementing the outcomes of agent actions and getting new perceptions.

public static void scheduler_setup(EASSEnv env, MCAPLScheduler s)

This takes an environment (typically one sub-classing `eass.mas.DefaultEASSEnvironment`) and a scheduler and sets the environment and scheduler up appropriately. In general an EASS environment will want to use `ail.mas.NActionScheduler` – this is a scheduler which can switch between agents and the environment every time an agent takes an action but will also switch every N reasoning cycles in to force checking of changes in the environment. A good value to start N at is 100 though this will vary by application. A typical constructor for an environment may look something like Example 25.

Example 25

```
public MyEnvironment() {                                                    1
        super();                                                            2
        super.scheduler_setup(this, new NActionScheduler(100));             3
}                                                                           4
```

public void addUniquePercept(String s, Predicate per) It is fairly typical in the kinds of applications the EASS is used for that incoming perceptions indicate the current value of some measure – e.g. the current distance to the car in front. This gets converted into a predicate such as *distance*(5.5) however the application only wants to have one such percept available. `addUniquePercept` avoids the need to use `removeUnifiesPercept` followed by `addPercept` each time the value changes. Instead `addUniquePercept` takes a unique reference string, `s` (normally the functor of the predicate – e.g., `distance`) and then removes the old percept and adds the new one.

public void addUniquePercept(String agName, String s, Literal pred) As above but the percept is perceivable only by the agent called `agName`.

8.2.3 Default Actions

Just as `ail.mas.DefaultEnvironment` provides a set of built-in actions, so does `eass.mas.DefaultEASSEnvironment`. These critically support some aspects of the EASS language:

assert_shared(B) This puts B in the shared belief set.

remove_shared(B) This removes B from the shared belief set.

remove_shared_unifies(B) This removes all beliefs that unify with B from the shared belief set. This is useful when you do not necessarily know the current value of one of a shared belief's parameters.

perf(G) This send a message to the abstraction engine requesting it adopt G as a perform goal.

append_string_pred(S, T, V) This is occasionally useful for converting between values treated as parameters by the agent, but which need to be translated to unique actions for the application (e.g., converting from *thruster*(2) to "thruster_2". It takes a string as its first argument, a term, T, as its second argument. It converts T to a string and then unifies the concatenation of the first string and the new string with V.

8.2.4 Adding Additional Actions

Adding additional actions can be done in the same way as for environments that subclass `ail.mas.DefaultEnvironment` (see section 4.2).

8.2.5 Adding Dynamic Behaviour

Dynamic behaviour can be added by overriding the method `do_job()`. This method is called each time the scheduler executes the environment. Overrides of this method can be used simply to change the set of percepts (possibly at random) or to read data from sockets or other communications mechanisms.

Once an environment is dynamic and is included in the scheduler it sometimes becomes important to know when the multi-agent system has finished running. This is not always the case, sometimes you want it to keep running indefinitely and just kill it manually when you are done, but if you want the system to shut down neatly then the scheduler needs to be able to detect when the environment has finished. To do this it calls the methods `public boolean done()` which should return `true` if the environment has finished running and `false` otherwise.

8.2.6 Example

You can find an example EASS environment, `CarOnMotorwayEnvironment`, for connecting to the Motorway Simulator over a socket in the tutorial directory. We will examine this section by section [1].

Example 26

```
public class CarOnMotorwayEnvironment extends DefaultEASSEnvironment {
                                                                        2
  String logname =                                                      3
          "eass.tutorials.tutorial2.CarOnMotorwayEnvironment";          4
                                                                        5
  /**                                                                   6
   * Socket that connects to the Simulator.                           7
   */                                                                  8
  protected AILSocketClient socket;                                     9
                                                                        10
  /**                                                                   11
   * Has the environment concluded?                                   12
   */                                                                  13
  private boolean finished = false;                                     14
```

[1] EASS comes with a dedicated class `eass.mas.socket.EASSSocketClientEnvironment` for environments that communicate with simulators via sockets. We discuss an environment implementation that does not use this class for tutorial purposes, but many applications may wish to use it.

Example 26 shows initialisation of the class. It sub-classes `DefaultEASSEnvironment`, sets up a name for logging and and a socket. The AIL comes with some support for socket programming. `ail.util.AILSocketClient` is a class for sockets which are clients of some server (as required by the Motorway simulator). Lastly the environment sets up a boolean to track whether it has finished executing.

Example 27

```
1  public CarOnMotorwayEnvironment() {
2    super();
3    super.scheduler_setup(this, new NActionScheduler(100));
4    AJPFLogger.info(logname, "Waiting Connection");
5    socket = new AILSocketClient();
6    AJPFLogger.info(logname, "Connected to Socket");
7  }
```

Example 27 shows the class constructor. We've set the environment up with an *NActionScheduler* – this scheduler switches between jobbers every time an agent takes and action, *but also*, every n turns of a reasoning cycle. In this case n is set to 100. This means that the environment keeps up to date processing input from the simulator even while agent deliberation is going on. We then create the socket. We don't supply a port number for the socket. The AIL socket classes have a default port number they use and the Motorway simulator uses this port so we don't need to specify it. We are using the `AJPFLogger` class to provide output. We will cover this in future tutorials. In this instance printing messages to System Error or System out would work as well.

Example 28 shows the code that gets executed each time the environment is scheduled to run. In this case we want to get up-to-date values from the simulator by reading them off the socket. The simulator posts output in sets of four doubles and then an integer representing the x position, y position, x speed, y speed of the car and finally the integer represents whether the simulation has started or not. The code in lines 17-21 reads off these values. This particular application isn't interested in the x and y position, so these are ignored but the speeds and starting information are saved as variables. Note that different methods are used to read doubles and integers, it is important to use the right methods otherwise simulator and agent environment can get out of sync since different datatypes use up different numbers of bytes on the socket. Lines 23-33 then repeat this process on a loop. `socket.pendingInput()` returns true if there is any data left to be read off the socket. Since the environment and simulator probably won't be entirely running in sync this loop is used to read all available data off the socket. The final assignment of values to variables will represent the most recent state of the simulation and so is probably the best data to pass on to the agent. Lines 35-44 show the environment turning the

Example 28

```
public void do_job() {
  if (socket.allok()) {
    readPredicatesfromSocket();
  }     else {
    System.err.println("something wrong with socket");
  }
}

/**
 * Reading the values from the sockets and turning them into perceptions.
 */
public void readPredicatesfromSocket() {
  socket.readDouble();
  socket.readDouble();
  double xdot = socket.readDouble();
  double ydot = socket.readDouble();
  int started = socket.readInt();

  try {
    while (socket.pendingInput()) {
      socket.readDouble();
      socket.readDouble();
      xdot = socket.readDouble();
      ydot = socket.readDouble();
      started = socket.readInt();
    }
  } catch (Exception e) {
    AJPFLogger.warning(logname, e.getMessage());
  }

  Literal xspeed = new Literal("xspeed");
  xspeed.addTerm(new NumberTermImpl(xdot));

  Literal yspeed = new Literal("yspeed");
  yspeed.addTerm(new NumberTermImpl(ydot));

  if (started > 0) {
    addPercept(new Literal("started"));
  }

  addUniquePercept("xspeed", xspeed);
  addUniquePercept("yspeed", yspeed);
}
```

numbers read from the socket into literals for use by the agent (see discussion in section 4.2). Finally `addUniquePercept` is used to add the percepts for `xspeed` and `yspeed` to the environment. We only want one value for each of these to be available to the agent so we use the special method to remove the old value and add the new one.

Example 29

```
public Unifier executeAction(String agName, Action act) throws AILexception {
                                                                        2
  if (act.getFunctor().equals("accelerate")) {                          3
    socket.writeDouble(0.0);                                            4
    socket.writeDouble(0.01);                                           5
  } else if (act.getFunctor().equals("decelerate")) {                   6
    socket.writeDouble(0.0);                                            7
    socket.writeDouble(-0.1);                                           8
  } else if (act.getFunctor().equals("maintain_speed")) {               9
    socket.writeDouble(0.0);                                            10
    socket.writeDouble(0.0);                                            11
  } else if (act.getFunctor().equals("finished")) {                     12
    finished = true;                                                    13
  }                                                                     14
                                                                        15
  return super.executeAction(agName, act);                             16
}                                                                       17
```

Example 29 shows the `executeAction` method which was discussed in section 4.2. Here, as well as the actions, such as assert_shared , remove_shared and perf provided by `DefaultEASSEnvironment` the environment offers accelerate , decelerate , maintain_speed and finished . The Motorway simulator regularly checks the socket and expects to find pairs of doubles on it giving the acceleration in the x and y directions respectively. The environment treats requests for acceleration and deceleration from the agent as requests to speed up or slow down in the y direction, but since the simulator expects a pair of values it has to write the x acceleration to the socket as well. finished is treated as a request to stop the environment and the boolean `finished` is set to true.

Example 30 shows the code used when to notify the system that the environment is finished (by overriding `done`) and an over-ride of the `cleanup()` method which is called before the system shuts down. This is used to close the socket.

Executing the Example

The example is a variation on the one used in section 8.1 and can be executed in the same way by first starting up `MotorwayMain` and then running AIL on `car.ail`. The main difference is that the agent in this program executes the

Example 30

```
public void cleanup() {                                                    1
  socket.close();                                                          2
}                                                                          3
                                                                           4
/*                                                                         5
 * (non-Javadoc)                                                           6
 * @see ail.others.DefaultEnvironment#done()                              7
 */                                                                        8
public boolean done() {                                                    9
  if (finished) {                                                          10
    return true;                                                           11
  }                                                                        12
  return false;                                                            13
}                                                                          14
```

finished action once the car has reached the speed limit. This results in the multi-agent system shutting down and the socket being closed. These actions don't terminate the simulation which will continue executing, but you will be able to see error messages of the form `WARNING: Broken pipe` being generated by its attempts to read data from the socket.

8.2.7 Sending Messages

The `executeAction` method in the default environment simply places messages directly into the intended recipient's inbox. Obviously there will be situations, particularly if the multi-agent system needs to send messages over a socket or similar, where this will not suffice.

In fact `executeAction` calls a method, `executeSendAction`: `public void executeSendAction(String agName, SendAction act)` so the simplest way to alter an environment's message sending behaviour is to override this method.

The `SendAction` class has several useful methods such as:

Message getMessage(String agName) which returns the `Message` object assocated with the action and takes the name of the sender as an argument.

Term getReciever() returns the name of the intended receiver of the message as a `Term`.

`Message` objects are described in section 4.2.

8.2.8 Exercises

Changing Lane

In the tutorial directory you will find an EASS program, `car_exercises.eass`. This contains a car control program that attempts to change lane (action change_lane) once the car has reached the speed limit. It then checks a perception, xpos(X) for the x position of the car until it believes it is in the next lane at which point it instructs the environment to stay in that lane (action stay_in_lane).

Extend and adapt `CarOnMotorwayEnvironment.java` to act as a suitable environment for this program. As normal answers (including an AIL configuration file) can be found in the `answers` directory for the tutorial.

Changing the Simulator Behaviour

It is possible to change the behaviour of the Motorway Simulator by providing it with a config file. A sample one is provided in the tutorial directory. This new configuration gets the simulator to write 7 values to the socket rather than five. These are, in order, the total distance travelled in the x direction, the total distance travelled in the y direction, the x coordinate of the car in the interface, the y coordinate of the car in the interface, the speed in the x direction, the speed in the y direction and whether the simulator has started now.

The simulator can be started in this configuration by supplying `"/src/examples/eass/tutorials/tutorial2/config.txt"` as an argument to `MotorwayMain` (If you are using Eclipse you can add arguments to Run Configurations in a tab). Adapt the environment to run `car_exercises.eass` in this environment. As normal answers (including an AIL configuration file) can be found in the `answers` directory for the tutorial.

8.3 Tutorial 3 – Verifying Reasoning Engines

This is the third in a series of tutorials on the use of the EASS variant of the Gwendolen language. This tutorial covers verifying EASS reasoning engines as described in [Dennis et al., 2014, Fisher et al., 2013].

Files for this tutorial can be found in the `mcapl` distribution in the directory

`src/examples/eass/tutorials/tutorial3`.

The tutorial assumes a good working knowledge of Java programming. It also assumes the reader is familiar with the basics of using AJPF to verify programs (see section 5.1 and section 5.2).

8.3.1 Overview

The process for verifying an EASS reasoning engine is to first analyse the agent program in order to identify all the shared beliefs that are sent from the abstraction engine to the reasoning engine. In multi-agent systems it is also necessary

to identify all messages that the reasoning engine may receive from other agents in the environment. This is discussed in some detail in [Dennis et al., 2014]. Once a list of shared beliefs and messages has been identified, an environment is constructed for the reasoning engine alone in such a way that every time the agent takes an action the set of perceptions and messages available to it are created *at random*. When model checking the random selection causes the search tree to branch and the model checker to explore all possibilities.

8.3.2 Example

As an example we will consider the accelerating car controller we looked at in section 8.1. The full code for this is show in Example 31 and from this we can see there are two shared beliefs used by the program, start and at_speed_limit .

For verification purposes, we are only interested in the reasoning engine so we create a file containing just the reasoning engine. This is `car_re.eass` in the tutorial directory. You will also find an AIL configuration file `car.ail`, a JPF configuration file, `car.jpf` and a property specification file, `car.psl` in the tutorial directory.

The environment for verifying the car reasoning engine is shown in example 32. This subclasses `eass.mas.verification.EASSVerificationEnvironment` which sets up a basic environment for handling verification of single reasoning engines. In order to use this environment you have to implement two methods, `generate_sharedbeliefs(String AgName, Action act)` and `generate_messages(String AgName, Action act)`. It is assumed that these methods will randomly generate the shared beliefs and messages of interest to your application. `EASSVerificationEnvironment` handles the calling of these methods each time the reasoning engine takes an action. It should be noted that `EASSVerificationEnvironment` ignores assert_shared and remove_shared actions, assuming these take negligible time to execute – this is largely in order to keep search spaces as small as possible. `generate_sharedbeliefs` and `generate_messages` both take the agent's name and the last performed action as arguments. These are used if creating *structured environments* which are not discussed here.

In the example verification environment, `generate_messages` returns an empty set of messages because we didn't identify any messages in the program. `generate_sharedbeliefs` is responsible for asserting at_speed_limit and start. `EASSVerificationEnvironment` provides `random_bool_generator` which is a member of the `ajpf.util.choice.UniformBoolChoice` class and `random_int_generator` which is a member of the `ajpf.util.choice.UniformIntChoice` class. These can be used to generate random boolean and integer values. In this case `random_bool_generator` is being used to generate two booleans, `assert_at_speed_limit` and `assert_start`. If these booleans are true then the relevant predicate is added to the set returned by the method while, if it is false, nothing is added to the set. An `AJPFLogger` is used to print out whether the shared belief was generated or not

Example 31

```
EASS

:abstraction: car

:Initial Beliefs:

speed_limit(5)

:Initial Goals:


:Plans:
/* Default plans for handling messages */
+.received(:tell, B): {True} ← +B;
+.received(:perform, G): {True} ← +!G [perform];
+.received(:achieve, G): {True} ← +!G [achieve];

+started : {True} ←
        +Σ(start);

+yspeed(X) : {B speed_limit(SL), SL < X} ←
        +Σ(at_speed_limit);
+yspeed(X) : {B speed_limit(SL), X < SL} ←
        −Σ(at_speed_limit);

+! accelerate [perform] : {B yspeed(X)} ← accelerate;
+! accelerate [perform] : {~B yspeed(X)} ←
print("Waiting for Simulator to Start");
+! maintain_speed [perform] : {True} ← maintain_speed;

:name: car

:Initial Beliefs:

:Initial Goals:

:Plans:

+start: {True} ←
        +!at_speed_limit[achieve];

+! at_speed_limit [achieve] : {True} ←
        perf(accelerate),
        *at_speed_limit;

+at_speed_limit: {True} ←
        perf(maintain_speed);
```

Example 32

```
/**                                                                    1
 * An environment for verifying a simple car reasoning engine.        2
 * @author louiseadennis                                              3
 *                                                                    4
 */                                                                   5
public class VerificationEnvironment extends                          6
                      EASSVerificationEnvironment {                   7
                                                                      8
  public String logname = "eass.tutorials.tutorial3.VerificationEnvironment";  9
                                                                      10
  public Set<Predicate> generate_sharedbeliefs(String AgName, Action act) {  11
    TreeSet<Predicate> percepts = new TreeSet<Predicate>();           12
    boolean assert_at_speed_limit = random_bool_generator.nextBoolean();  13
    if (assert_at_speed_limit) {                                      14
      percepts.add(new Predicate("at_speed_limit"));                  15
      AJPFLogger.info(logname, "At the Speed Limit");                 16
    } else {                                                          17
      AJPFLogger.info(logname, "Not At Speed Limit");                 18
    }                                                                 19
                                                                      20
    boolean assert_start = random_bool_generator.nextBoolean();       21
    if (assert_start) {                                               22
      percepts.add(new Predicate("start"));                           23
      AJPFLogger.info(logname, "Asserting start");                    24
    } else {                                                          25
      AJPFLogger.info(logname, "Not asserting start");                26
    }                                                                 27
    return percepts;                                                  28
  }                                                                   29
                                                                      30
  public Set<Message> generate_messages() {                           31
    TreeSet<Message> messages = new TreeSet<Message>();               32
      return messages;                                                33
  };                                                                  34
}                                                                     35
```

– this can be useful when debugging failed model checking runs.

There are four properties in the property specification file:

1 $\Box\neg\mathcal{B}_{\texttt{car}}\, crash$ – The car never believes it has crashed. We know this to be impossible – no such belief is ever asserted – but it can be useful to have a simple property like this in a file in order to check the basics of the model checking is working.

2 $\Box(\mathcal{A}_{\texttt{car}} perf(accelerate) \Rightarrow (\Diamond\mathcal{A}_{\texttt{car}} perf(maintain\_speed) \vee \Box\neg\mathcal{B}_{\texttt{car}}\, at\_speed\_limit))$ – If the car ever accelerates then either eventually it maintains its speed, or it never believes it has reached the speed limit.

3 $\Box(\mathcal{B}_{\texttt{car}}\, at\_speed\_limit) \Rightarrow \Diamond\mathcal{A}_{\texttt{car}} perf(maintain\_speed)$ – If the car believes it is at the speed limit then eventually it maintains its speed. Properties of this form are often not true because $\mathcal{A}_{ag}a$ only applies to the last action performed and beliefs are often more persistent than that so the agent acquires the belief, b, does action a, and then does something else. At this point it still believes b but unless it does a again the property will be false in LTL. In this case, however, the property is true because perf(maintain_speed) is the last action performed by the agent.

4 $\Box\neg\mathcal{B}_{\texttt{car}}\, start \Rightarrow \Box\neg\mathcal{A}_{\texttt{car}} perf(accelerate)$ – If the car never believes the simulation has started then it never accelerates.

The JPF configuration file in the tutorial directory is set to check property 3. It is mostly a standard configuration, as discussed in section 5.2. However it is worth looking at the list of classes passed to `log.info`. These are:

ail.mas.DefaultEnvironment As discussed in section 5.2, this prints out the actions that an agent has performed and is useful for debugging.

eass.mas.verification.EASSVerificationEnvironment This prints out when an agent is just about to perform an action, before all the random shared beliefs and messages are generated. If both this class and `ail.mas.DefaultEnvironment` are passed to `log.info` then you will see a message before the agent does an action, then the search space branching as the random shared beliefs and messages are generated and then a message when the action completes. You may prefer to have only one of these print out.

eass.tutorials.tutorial3.VerificationEnvironment As can be seen in example 32 this will cause information about the random branching to get printed.

ajpf.product.Product As discussed in section 5.2, this prints out the current path through the AJPF search space.

8.3.3 Messages

Normally there is no need to construct messages in environments since this is handled by the way `ail.mas.DefaultEnvironment` handles send actions. However for EASS verification environments, where messages must be constructed at random, it is necessary to do this in the environment.

The important class is `ail.syntax.Message` and the main constructor of interest is `public Message(int ilf, String s, String r, Term c)`. The four parameters are

ilf This is the *illocutionary force* or the *performative*. For EASS agents this should be 1, for a tell message, 2 for a perform message and 3 for an achieve message. If in doubt you can use the static fields `EASSAgent.TELL`, `EASSAgent.PERFORM` and `EASSAgent.ACHIEVE`..

s This is a string which is the name of the sender of the message.

r This is a string which is the name of the receiver of the message.

c This is a term for the content of the message and should be created using the AIL classes for `Predicate`s etc.

Where messages are to be randomly generated a list of them should be created in `generate_messages`.

8.3.4 Exercises

Exercise 1

Take the sample answer for exercise 2 in section 8.1, and verify that if the car never gets an alert then it never stops. As usual you can find a sample answer in the `answers` sub-directory.

Exercise 2

In the tutorial directory you will find a reasoning engine, `car_re_messages.eass`. This is identical to `car_re.eass` apart from the fact that it can process tell messages. Provide a verification environment where instead of `start` being asserted as a shared belief, the agent receives it as a tell message from the simulator. Check you can verify the same properties of the agent. As usual you can find a sample answer in the `answers` sub-directory.

Chapter 9

Executing and Verifying GOAL agents in the AIL and AJPF

The implementation of GOAL in the AIL for use with AJPF is based upon the 2014 version of GOAL. Documentation for this version can be found in the version of *Programming Cognitive Agents in GOAL* included with the AJPF distribution in the `doc/goal` directory. These notes are intended to be used alongside the Programming Guide which explains how to program in GOAL.

While every effort has been made to keep the syntax and format of the GOAL program files in line with this document. The usage and format of configuration files differs. These differences are outlined in this document.

Many examples from *Programming Cognitive Agents in GOAL* can be found in the `mcapl` distribution in the directory

`src/examples/goal/programming_guide`.

9.1 AIL Configuration Files

You will find an AIL configuration file in the chapter 1 directory called `hello_world.ail`. Its contents is shown below.

```
env = goal.mas.GoalEnvironment

mas.file = /src/examples/goal/programming_guide/chapter1/hello_world.gl
mas.builder = goal.GOALMASBuilder
```

This is a very simple configuration consisting of three items only.

mas.file gives the path to the GOAL program to be run.

mas.builder gives a java class for building the file. In this case `goal.GOALMASBuilder` parses a file containing a GOAL agent and compiles it into a multi-agent system.

env provides an environment for the agent to run in. In this case we use a default GOAL environment provided by the AIL.

The GOAL program file, `hello_world.gl` is also in the chapter directory.

9.1.1 Running the Program

To run the program you need to run the JAVA program `ail.mas.AIL` and supply it with a the configuration file as an argument. You can do this either from the command line or using the IntelliJ or Eclipse `run-AIL` configuration (with `hello_world.ail` selected in the Project Files/Package Explorer window) as detailed in chapter 3.

9.1.2 Configuration Files

Configuration files all contain a list of items of the form `key=value`. Particular agent programming languages, and even specific applications may have their own specialised keys that can be placed in this file. However the keys that are supported by all agent programs are as follows:

env This is the Java class that represents the environment of the multi-agent system. The value should be a java class name – e.g., `goal.mas.GoalEnvironment`.

mas.file This is the name of a file (including the path from the MCAPL home directory) which describes all the agents needed for a multi-agent system in some agent programming language.

mas.builder This is the Java class that builds a multi-agent system in some language. For GOAL this is `goal.GOALMASBuilder`. To find the builders for other languages consult the language documentation.

mas.agent.*N*.file This is the name of a file (including the path from the MCAPL home directory) which describes the Nth agent to be used by some multi-agent system. This allows individual agent code to be kept in separate files and allows agents to be re-used for different applications. It also allows a multi-agent system to be built using agents programmed in several different agent programming languages. You can see an example of the use of this in section 9.6.2.

mas.agent.*N*.builder This is the Java class that is to be used to build the Nth agent in the system. In the case of GOAL individual agents are built using `goal.GOALAgentBuilder`. You can see an example of the use of this in section 9.6.2.

mas.agent.*N*.name All agent files contain a default name for the agent but this can be changed by the configuration (e.g., if you want several agents which are identical except for the name – this way they can all refer to the same code file but the system will consider them to be different agents because they have different names). You can see an example of the use of this in section 9.6.2.

log.severe, log.warning, log.info, log.fine, log.finer, log.finest These all set the logging level for Java classes in the system. `log.finest` prints out the most information and `log.severe` prints out the least. By default most classes are set to `log.warning` but sometimes, especially when debugging, you may want to specify a particular logging level for a particular class.

log.format This lets you change the format of the log output from Java's default. At the moment the only value for this is `brief`.

ajpf.transition_every_reasoning_cycle This can be `true` or `false` (by default it is true). It is used during model checking with AJPF to determine whether a new model state should be generated for every state in the agent's reasoning cycle. This means that model checking is more thorough, but at the expense of generating a lot more states.

ajpf.record This can be `true` or `false` (by default it is false). If it is set to true then the program will record its sequence of choices (all choices made by the scheduler *and* any choices made by the special `ajpf.util.choice.Choice` class). By default (unless `ajpf.replay.file` is set) these choices are stored in a file called `record.txt` in the `records` directory of the MCAPL distribution.

ajpf.replay This can be set to `true` or `false` (by default it is false). If it is set to true then the system will execute the program using a set of scheduler and other choices from a file. By default (unless `ajpf.replay.file` is set) this file is `record.txt` in the `records` directory of the MCAPL distribution.

ajpf.replay.file This allows you to set the file used by either `ajpf.record` or `ajpf.replay`.

More information on the use of AIL configuration files can be found in section 4.1.

9.2 Notes on Chapter 1

AIL versions of all the examples that appear in chapter 1 of *Programming Cognitive Agents in GOAL* can be found in

`src/examples/goal/programming_guide/chapter1`.

In section 1.1. of *Programming Cognitive Agents in GOAL*, a MAS file is presented for a Hello World agent containing **agentfiles** and a **launchpolicy**. Instead of using a MAS file, users of the AIL version of GOAL should use an AIL configuration file as discussed in section 9.1.

Section 1.5 discusses using an environment `HelloWorldEnvironment.jar`. The MCAPL distribution instead supplies `HellowWorldEnvironment.java` in the chapter directory which can be included in an AIL configuration file as:

```
env = goal.programming_guide.chapter1.HelloWorldEnvironment
```

9.3 Model Checking GOAL Programs

GOAL programs can be model-checked in the same way that GWENDOLEN and other programs in the AIL framework can be checked. This involves creating a JPF configuration file and setting the program's AIL configuration file, a property specification file, and a property key as the `target.args` for a `target` of `ail.util.AJPF_w_AIL`. This process is outlined in more detail in section 5.1 and in section 5.2.

We outline the basics of this here.

9.3.1 Setting up Agent Java Pathfinder

Before you can run AJPF it is necessary to set up your computer to use Java Pathfinder. There are instructions for doing this in the MCAPL manual (which you can find in the `doc` directory of the distribution).

Just as you run multi-agent systems in the AIL by passing an AIL configuration file as an argument to `ail.mas.AIL`, you model-check a multi-agent system by passing a JPF configuration file as an argument to `ajpf.util.AJPF_w_AIL`.

9.3.2 Example

Figure 9.1 shows a JPF configuration file of the `hello_world.gl` example from chapter 1 of *Programming Cognitive Agents in GOAL* .

We explain each line of this below.

@using = mcapl Means that the proof is using the home directory for `mcapl`. This should be be set up in `.jpf/site.properties` (See the MCAPL manual).

target = ail.util.AJPF_w_AIL This is the Java file containing the main method for the program to be model checked. By default when model checking a program implemented using the AIL, you should use `ail.util.AJPF_w_AIL` as the target. For those who are familiar with running programs in the AIL, this class is very similar to `ail.mas.AIL` but with a few tweaks to set up and optimise model checking.

```
@using = mcapl

target = ail.util.AJPF_w_AIL
target.args = ${mcapl}/src/examples/goal/programming_guide/chapter1/hello_world.ail,${mcapl}/src/examples/g

log.info = ajpf.MCAPLAgent,ail.mas.DefaultEnvironment,ajpf.product.Product

listener+=,.listener.ExecTracker
et.print_insn=false
et.show_shared=false
```

Figure 9.1: Hello World Configuration File

target.args =... This sets up the arguments to be passed to `ail.util.AJPF_w_AIL`. `ail.util.AJPF_w_AIL` takes three arguments. In the configuration file these all have to appear on one line, separated by commas (but *no spaces*). This means you can not see them all in the file print out above. In order the arguments are:

1. The first is an AIL configuration file. In this example the file is `${mcapl}/src/examples/goal/programming_guide/chapter1/hello_world.ail` which is a configuration file for a simple Hello World program.
2. The second argument is a file containing a list of properties in AJPF's property specification language that can be checked. In this example this file is `simple.psl` in the directory chapter 1 of the programming guide.
3. The last argument is the name of the property to be checked, `1` in this case.

log.info =... JPF suppresses the logging configuration you have in your AIL configuration files so you need to add any logging configurations you want to the JPF configuration file. Useful classes when debugging a model checking run are

ail.mas.DefaultEnvironment At the `info` level this prints out any actions the agent performs. Since the scheduler normally only switches between agents when one sleeps or performs an action this can be useful for tracking progress on this model checking branch.

ajpf.MCAPLAgent At the `info` level this prints information when an agent sleeps or wakes. Again this can be useful for seeing what has triggered a scheduler switch. It can also be useful for tracking which agents are awake and so deducing which one is being picked.

ajpf.product.Product At the `info` level this prints out the current path through the search tree being explored by the agent. This can be

useful just to get a feel for the system's progress through the search space. It can also be useful, when an error is thrown and in conjunction with some combination of logging actions, sleeping and waking behaviour and (if necessary) internal agent states, to work out why a property has failed to hold.

It also prints the message `Always True from Now On` when exploration of a branch of the search tree is halted because the system deduces that the property will be true for the rest of that branch. This typically occurs when the property is something like $\diamond\phi$ (i.e., ϕ will eventually occur) and the search space is pruned once ϕ becomes true.

ajpf.psl.buchi.BuchiAutomaton At the `info` level this prints out the Büchi Automaton that has been generated from the the property that is to be proved. Again this is useful, when model checking fails, for working out what property was expected to hold in that state.

ail.semantics.AILAgent At the `fine` level this prints out the internal agent state once every reasoning cycle. Be warned that this produces a lot of output in the course of a model checking run.

listener+=,.listener.ExecTracker Adding `listener.ExecTracker` to JPF's listeners means that it collects more information about progress as it goes and then prints this information out. The next two lines suppress some of this information which is usually less useful.

9.3.3 Property Specification

The file `simple.psl` specifies two Linear Temporal Logic (LTL) properties for checking. Given the Hello World programs are so simple these properties are very basic.

```
1: [](~B(goal_agent,bad))

2: <>(B(goal_agent, nrOfPrintedLines(10)))
```

The first of these is equivalent to the LTL statement "it is always the case that `goal_agent` doesn't believe `bad`" $\Box\neg\mathcal{B}_{\texttt{goal\_agent}}\, bad$. The second is equivalent to the LTL statement "eventually `goal_agent` believes the number of printed lines is 10" $\diamond\mathcal{B}_{\texttt{goal\_agent}}\, nrOfPrintedLines(10)$.

You can find more detail on property specification in section 5.1.

9.3.4 Running AJPF

To run AJPF you need to run the program `gov.jpf.tool.RunJPF` which is contained in `lib/3rdparty/RunJPF.jar` in the MCAPL distribution. Alternatively you can use the `run-JPF (MCAPL)` Run Configuration in Eclipse.

You need to supply the JPF Configuration file as an argument.

9.4 Notes on Chapters 3 & 4

Chapters 3 and 4 build up to executable programs that are used in Chapters 5 and 6. As such we discuss the details in the notes on those chapters. In particular the notes on chapter 6 discuss the use of the Blocks World environment.

It should be noted that actual GOAL program code in AIL is identical to that presented in *Programming Cognitive Agents in GOAL*, it is only configuration files and sometimes environments that differ.

9.5 Notes on Chapter 5

The goal programs in chapter 5 do not need to use sensing to gain information from any external environment. As such they can be run successfully in the Default GOAL environment `goal.mas.GoalEnvironment`

Both stack builder programs used in the chapter can be found in

`src/examples/goal/programming_guide/chapter5`.

together with AIL configuration files allowing them to be run and AJPF configuration files allowing the system to check properties of them. It should be noted that the random version of the stack builder program takes considerably longer to model-check because of the increase in search space caused by the random evaluation of rules.

9.6 Notes on Chapter 6

All the programs discussed in the chapter 6 of *Programming Cognitive Agents in GOAL* can be found in

`src/examples/goal/programming_guide/chapter6`.

9.6.1 Section 6.1

Like GOAL, the AIL supports the Environment Interface Standard (EIS). However a little more effort is required to use this than in GOAL itself where it can all be managed via a configuration file.

In the AIL a mediating environment is needed between an environment that supports the EIS and the system itself. There are two of these in

`src/examples/goal/programming_guide/chapter6`.

one for the Blocks World and one for the Tower World. These mediating environments extend `GOALEISEnvironment` (which is an AIL style environment for GOAL that supports the EIS) and provide some simple configuration methods.

Table 6.1 in the programming guide shows a MAS file for use with the Blocks World environment. In the AIL version of goal this becomes the combination

of the `table6_1.ail` configuration file and the `BlocksWorldEnvironment.java` mediating environment. We will discuss these in turn.

```
env = goal.programming_guide.chapter6.BlocksWorldEnvironment
goal.env.init.start = bwconfigEx1.txt
goal.env.init.gui = true

mas.file = /src/examples/goal/programming_guide/chapter5/stackBuilder.gl
mas.builder = goal.GOALMASBuilder

goal.launchpolicy.entity.launch = goal_agent
```

Here instead of providing a jar file for the environment we supply the mediating environment to the `env` argument. However the other components of the environment section of the GOAL MAS file are present. `init = [start = ''bwconfigEx1.txt'']` becomes `goal.env.init.start = bwconfigEx1.txt`. We also add `goal.env.init.gui = true` since this displays a useful GUI interface for the environment.

The **agentfiles** section of the MAS file has become `mas.file` and `mas.builder` as described in the notes for Chapter 1.

Lastly the **launchpolicy** section says to launch the entity `goal_agent` which is the default agent name given to GOAL agents in AIL. We will discuss renaming agents when we examine the next example.

Example 33 shows the mediating environment. It is the constructor for this environment that contains the jar file for the EIS environment blocks world. In our case we are using `blocksworld-1.1.0.jar` which we've supplied with the MCAPL distribution. The `configure` method is used to configure the environment initialisation using the Java methods `addFileToInitMap`, `addToInitMap` which are supplied by the `GOALEISEnvironment` – these add tuples of a string and an EIS parameter to an *initialisation map* which is eventually passed to the EIS environment for initialisation. Since the initialisation parameters are specific to the environment and may be filenames or other types of EIS parameters it is necessary to implement the configuration of these specifically for the environment. The `configure` method then calls the method in the super class, `GOALEISEnvironment`, which handles configuration of launch policies, etc.

Lastly the `done()` method in `GOALEISEnvironment` is overridden. By default GOAL environments are assumed to change without an agent performing an action (e.g., because of non software agents acting in them). This is not the case with the Blocks World where things only move if the agent moves them. This being the case the method `done()` should return true indicating that the program can exit if the agent has finished executing (see the discussion of AIL environments in the AIL tutorials).

Example 33

```
package goal.programming_guide.chapter6;

import java.util.HashMap;
import java.util.Map;

import ail.util.AILConfig;
import eis.EnvironmentInterfaceStandard;
import eis.iilang.Identifier;
import eis.iilang.Parameter;
import goal.mas.GOALEISEnvironment;
import goal.mas.GoalEnvironment;
import goal.util.LaunchPolicy;

public class BlocksWorldEnvironment extends GOALEISEnvironment {

  public BlocksWorldEnvironment() {
    super("/lib/eis/blocksworld-1.1.0.jar");
  }

  @Override
  public void configure(AILConfig config) {
    if (config.containsKey("goal.env.init.start")) {
      String filename = config.getProperty("config_path") + "/" +
        config.getProperty("goal.env.init.start");
      addFileToInitMap("start", filename);
    }

    if (config.containsKey("goal.env.init.gui")) {
      Identifier value = new Identifier("true");
      if (config.getProperty("goal.env.init.gui").equals("false")){
        value = new Identifier("false");
      }
      addToInitMap("gui", value);
    }

    super.configure(config);
  }


  @Override
  public boolean done() {
    return true;
  }


}
```

9.6.2 Section 6.2

Section 6.2 introduces the use of two agents with the Blocks World environment. These are the `stackBuilder` agent from chapter 5, and a `tableAgent` introduced in chapter 6. The GOAL agent code in the AIL version is identical to that in the GOAL manual however the configuration file (figure 6.1 in the goal manual) is

```
env = goal.programming_guide.chapter6.BlocksWorldEnvironment
goal.env.init.start = bwconfigEx1.txt
goal.env.init.gui = true

mas.agent.1.file = /src/examples/goal/programming_guide/chapter6/stackBuilder.gl
mas.agent.1.builder = goal.GOALAgentBuilder
mas.agent.1.name = stackbuilder
mas.agent.2.file = /src/examples/goal/programming_guide/chapter6/tableAgent.gl
mas.agent.2.builder = goal.GOALAgentBuilder
mas.agent.2.name = tableagent

goal.launchpolicy.entity.launch = stackbuilder,tableagent
```

This uses the mechanisms for naming individual agents from AIL discussed in section 9.1.2

The configuration file can be found in the file `two_agents.ail` in the `chapter6` directly. Unfortunately when running this, events happen so fast that it is difficult to see what is going on although a series of actions taken by each agent will be printed to the console. To see the effect in the environment you need to run the system using a Java debugger. We would suggest with a breakpoint set in the `executeAction` method in the `ail.mas.eis.EISEnvironmentWrapper` class. This will allow you to observe the effect each time an agent executes an action in the environment.

9.6.3 Section 6.3 onwards

The remaining sections in chapter 6 refer to a "tower world" environment. Again this is included in the AIL distribution. The link to the tower world environment can be found in `TowerWorldEnvironment.java` in the `chapter6` package and the configuration file can be found in `tower_agent.ail`.

A further environment and configuration file `SimpleTowerWorldEnvironment.java` and `tower_agent_simple.ail` can be found in the directory. These were created to allow automated testing of the GOAL implementation and provide an environment which is not interactive and doesn't use the EIS. You may wish to examine them if interested but this isn't necessary for learning GOAL from the programming guide.

9.7 Chapter 7

Chapter 7 examines communicating agents via two examples: an Elevator example and a Coffee Maker Example. The `chapter7` package in the AIL distribution contains only at the Coffee Maker example. This does not require a special environment and so used the default `goal.mas.GoalEnvironment`. It launches the two agents `maker` and `grinder` as the `tableagent` and `stackbuilder` were launched in our discussion of section 6.2.

Bibliography

[Bremner et al.,] Bremner, P., Dennis, L. A., Fisher, M., and d, A. F. W. On proactive, transparent and verifiable ethical reasoning for robots. *Proceedings of the IEEE special issue on Machine Ethics: The Design and Governance of Ethical AI and Autonomous Systems*.

[Courcoubetis et al., 1992] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. (1992). Memory-efficient Algorithms for the Verification of Temporal Properties. In *Formal Methods in System Design*, pages 275–288.

[Dennis et al., 2015a] Dennis, L., Fisher, M., Slavkovik, M., and Webster, M. (2015a). Formal verification of ethical choices in autonomous systems. *Robotics and Autonomous Systems*, pages –.

[Dennis et al., 2011] Dennis, L., Fisher, M., Webster, M., and Bordini, R. (2011). Model checking agent programming languages. *Automated Software Engineering*, pages 1–59. 10.1007/s10515-011-0088-x.

[Dennis, 2017] Dennis, L. A. (2017). Gwendolen semantics: 2017. Technical Report ULCS-17-001, University of Liverpool, Department of Computer Science.

[Dennis et al., 2016] Dennis, L. A., Aitken, J. M., Collenette, J., Cucco, E., Kamali, M., McAree, O., Shaukat, A., Atkinson, K., Gao, Y., Veres, S. M., and Fisher, M. (2016). Agent-based autonomous systems and abstraction engines: Theory meets practice. In Alboul, L., Damian, D., and Aitken, M. J., editors, *Towards Autonomous Robotic Systems: 17th Annual Conference, TAROS 2016, Sheffield, UK, June 26–July 1, 2016, Proceedings*, pages 75–86, Cham. Springer International Publishing.

[Dennis and Fisher, 2023] Dennis, L. A. and Fisher, M. (2023). *Verifiable Autonomous Systems:Using Rational Agents to Provide Assurance about Decisions Made by Machines*. Cambridge University Press.

[Dennis et al., 2010] Dennis, L. A., Fisher, M., Lincoln, N., Lisitsa, A., and Veres, S. M. (2010). Declarative Abstractions for Agent Based Hybrid Control Systems. In *Proc. 8th Int. Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 6619 of *LNCS*, pages 96–111. Springer.

[Dennis et al., 2014] Dennis, L. A., Fisher, M., Lincoln, N. K., Lisitsa, A., and Veres, S. M. (2014). Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering*, pages 1–55.

[Dennis et al., 2015b] Dennis, L. A., Fisher, M., and Webster, M. (2015b). Two-stage agent program verification. *Journal of Logic and Computation*.

[Dennis et al., 2012] Dennis, L. A., Fisher, M., Webster, M., and Bordini, R. H. (2012). Model Checking Agent Programming Languages. *Automated Software Engineering*, 19(1):5–63.

[Dennis et al., 2015c] Dennis, L. A., Fisher, M., and Winfield, A. F. T. (2015c). Towards verifiably ethical robot behaviour. In *AAAI Workshop on AI and Ethics (1st International Conference on AI and Ethics)*, Austin, TX.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and Modal Logic. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier.

[Ferrando et al., 2018] Ferrando, A., Dennis, L. A., Ancona, D., Fisher, M., and Mascardi, V. (2018). Verifying and validating autonomous systems: an integrated approach. In *8th IEEE International Conference on Runtime Verification*.

[Ferrando et al., 2021] Ferrando, A., Dennis, L. A., Cardoso, R. C., Fisher, M., Ancona, D., and Mascardi, V. (2021). Toward a holistic approach to verification and validation of autonomous cognitive systems. *ACM Trans. Softw. Eng. Methodol.*, 30(4).

[Fisher et al., 2013] Fisher, M., Dennis, L. A., and Webster, M. P. (2013). Verifying autonomous systems. *Commun. ACM*, 56(9):84–93.

[Gerth et al., 1996] Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. (1996). Simple on-the-fly Automatic Verification of Linear Temporal Logic. In *Proc. 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK. Chapman & Hall, Ltd.

[Havelund et al., 2000] Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., and White, J. L. (2000). Formal Analysis of the Remote Agent Before and After Flight. In *Proc. 5th NASA Langley Formal Methods Workshop, Virginia, USA*.

[Hindriks, 2014] Hindriks, K. V. (2014). *Programming Cognitive Agents in GOAL*.

[Hindriks et al., 2001] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. (2001). Agent Programming with Declarative Goals. In *Intelligent Agents VII*, volume 1986 of *LNAI*, pages 228–243. Springer.

[Holzmann, 2004] Holzmann, G. (2004). *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

[Kars, 1996] Kars, P. (1996). The Application of Promela and Spin in the BOS Project (Abstract). *http://spinroot.com/spin/Workshops/ws96/Ka.pdf*. Accessed 2013-05-30.

[Kirsch et al., 2011] Kirsch, M. T., Regenie, V. A., Aguilar, M. L., Gonzalez, O., Bay, M., Davis, M. L., Null, C. H., Scully, R. C., and Kichak, R. A. (2011). Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation. NASA Engineering and Safety Center Technical Assessment Report.

[Kwiatkowska et al., 2011] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer.

[Lindner et al., 2017] Lindner, F., Bentzen, M., and Nebel, B. (2017). The HERA Approach to Morally Competent Robots. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*.

[Rao and Georgeff, 1992] Rao, A. S. and Georgeff, M. P. (1992). An Abstract Architecture for Rational Agents. In *Proc. International Conference on Knowledge Representation and Reasoning (KR&R)*, pages 439–449. Morgan Kaufmann.

[Visser et al., 2003] Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F. (2003). Model Checking Programs. *Automated Software Engineering*, 10(2):203–232.

Index