A Qualitative Comparative Study between Posh and GOAL for Programming UT2004 Bots

Master’s Thesis

M.P. Korstanje
A Qualitative Comparative Study between Posh and GOAL for Programming UT2004 Bots

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

M.P. Korstanje
born in Diepenveen, the Netherlands

Interactive Intelligence Group
Department of Intelligent Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
A Variety of agent programming languages have been proposed in the literature but only few comparative studies have been performed to evaluate the strengths and weaknesses of these languages. In order to gain a better understanding of the programming features and the use of these features by programmers it is imperative to perform studies on various programming tasks in these languages. Such studies advance our knowledge of the benefits of using agent-oriented languages and may contribute positively to the evolution of these languages.

As a first step we perform an analysis of the submissions to the Multi-Agent Programming Contest. We look at common patterns, structures and methodologies used. Additionally a functional categorization is made to identify common structures in the agent program. We discover that participants favor modular abstractions and simple decision structures. Iterative methodologies seem to perform better. Based on these findings we provide a set of recommendations that agent programming languages should follow.

We also use these findings to design the study which is the primary focus of this thesis. In this study we compare the performance of programmers on tasks that use the agent programming frameworks GOAL and POSH. The aim of the study was to investigate any differences in usability of the frameworks as well as differences between novice and more advanced programmers using either framework. The results suggest that there is no significant difference for the given tasks between both frameworks. We did find that general programming experience seems to be a relatively big advantage when using a (new) programming language. Analysis of the assignment and the observations made about the use of the frameworks, however, suggest that the experimental design can be adapted in such a way that differences in usability of the frameworks can be established.
Thesis Committee:

Chair: Prof. Dr. Catholijn M. Jonker, Faculty EEMCS, TU Delft
University supervisor: Dr. Koen Hindriks, Faculty EEMCS, TU Delft
Committee Member: Dr. Andy Zaidman, Faculty EEMCS, TU Delft
Preface

This thesis is the conclusion of my Master of Computer Science with the department of Interactive Intelligence of the faculty of Electrical engineering, Mathematics and Computer Science at Delft University of Technology.

First of all I would like to thank my family and friends for all their support and encouragements. They helped me build the resolve to finish this work. You know who you are.

I’d also like to thank the people who supported me in creating this work. My thesis supervisor Koen Hindrinks for his feedback, comments and overall guidance. The Pogamut developers. Not only has their work allowed me to write this thesis; I’ve enjoyed every minute of working with and on the Pogamut Project. It was great to meet you all, I will fondly remember the two months in Prague. A special note of thanks goes out to Jakub Gemrot who not only received the brunt of my patches and bug reports but also helped me setup in Prague.

Finally I want to thank all the participants in my experiment. Not only were you willing to spend a whole day helping me with my thesis, you did so with a great enthusiasm and inspiring curiosity. I wish you all the best in your future studies.

M.P. Korstanje
Delft, the Netherlands
September 4, 2013
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

2 Preliminary 3
   2.1 Intelligent Agents ........................................... 3
   2.2 The GOAL Agent Programming Language ....................... 4
   2.3 POSH .................................................. 10

3 Related Work 17

4 Analysis of the Multi-Agent Programming Contest 19
   4.1 Multi-Agent Programming Contest .............................. 19
   4.2 Methodologies ............................................. 21
   4.3 Code Inspection ........................................... 22
   4.4 Categorization ............................................. 30
   4.5 Discussion ............................................... 38

5 Comparing Programming in POSH and GOAL 41
   5.1 Comparison of POSH and GOAL ................................ 42
   5.2 Emohawk Environment ........................................ 43
   5.3 Experimental Design ........................................ 45
   5.4 Results ................................................ 52
   5.5 Observations ............................................. 57
   5.6 Discussion ............................................... 62

6 Recommendations, Conclusions and Future Work 65
List of Figures

2.1 A partial overview of a simple GOAL program for an agent. 4
2.2 An example of the blocks world environment. 5
2.3 Structure of a GOAL program transform the environment to a specific goal state 7
2.4 Structure of a GOAL program 7
2.5 The event section of a GOAL program. 8
2.6 The knowledge section of a GOAL program. 8
2.7 The program section of the main module. 9
2.8 The action spec of a GOAL program. 9
2.9 The goals section of a GOAL program. 9
2.10 Overview of the graphical representation of POSH and low level actions and senses written in Java. 10
2.11 A visualization of a simple posh plan. 11
2.12 LISP like code of a posh plan. 12
2.13 A POSH action and sense combined by a drive element. 12
2.14 A drive collection showing the drive elements containing individual plans. 12
2.15 A drive element is expanded showing the action and sense used to trigger it. 13
2.16 A drive element containing a competence as its action. 13
2.17 A drive element containing a action pattern as its action. 14
2.18 A simplified overview of the class structure of an agent. 14
4.1 The Mars Environment. Mars abstracted as a graph. 20
4.2 The communication between server and agents. 21
4.3 The distribution of action and knowledge in the core agent categories. 36
5.1 Feature comparison of POSH and GOAL. 42
5.2 General overview of POSH, GOAL and the connection to the Emohawk Environment 44
5.3 Illustrative Scenes from Emohawk Environment 45
5.4 Time-to-complete (mins) for comprehension task 53
5.5 Time-to-complete (mins) for coding tasks 1-3 54
5.6 Scatter plots - first three coding tasks ................................. 55
Chapter 1

Introduction

Literature [34] defines an agent as;

... a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

Some examples of things such an agent can do would be sorting the email based on its priority or stock trading algorithms that read the news. Agents are also used to simulate human beings in crowd and highway simulations or provide opponents with human behavior in computer games.

To develop such agents a variety of agent programming languages have been proposed in the literature (see e.g., [6, 7]). For this thesis we’ll specifically look at rule based languages that support the implementation of decision making in (multi-)agent systems.

Only few comparative studies have been performed to evaluate the strengths and weaknesses of these languages. In order to gain a better understanding of the use of these languages by programmers it is imperative to perform studies on various programming tasks.

As such the research question this thesis tries to answer is:

For a set of typical programming tasks, which features of rule-based agent programming languages facilitate novice and advanced programmers in writing good agent programs?

In this context good agent programs are programs that execute their tasks well. Likewise both novice and advanced programmers needs to be taken quite literally. Novice programmers are programmers that had little to no experience with the languages whereas advanced programmers had received a more extensive training.

To answer this question we first introduce several important concepts in chapter 2. This chapter also introduces the concept of an agent as well as agent programming languages POSH and GOAL. We then investigate two instances of applied agent programming. The first is a preliminary analysis and comparison of the code of participants in the Multi-Agent Programming Contest in chapter 4. This chapter describes common patterns, structures and methodologies used by the different teams.
Chapter 5 describes a qualitative comparative analysis of POSH and GOAL for programming bots in an Unreal Tournament Environment. In this analysis subjects complete a set of tasks using either POSH or GOAL. The design of the tasks is informed by the results of the previous investigation. To support the analysis a connection was created that allows both languages to use the same environment. An important aspect in the design of this connection is that both languages have to work at the same conceptual level of abstraction.
Chapter 2

Preliminary

In this chapter we introduce several important concepts used in this thesis. We already introduced the basic notion of Agent in the introduction. Section 2.1 discusses the concept of agents in more detail. Section 2.2 and 2.3 introduce POSH and GOAL respectively.

2.1 Intelligent Agents

In the introduction we used a definition provided by Wooldridge[34] to define an agent. An agent is a computer system that is situated in an environment in which it can take actions to meet some objective. This is a rather broad definition. A Unix daemon or even a simple thermostat can satisfy it. Yet neither is very intelligent. Intelligence can be defined by the kinds of properties expressed by an agent. Literature provides three properties[35, 34] for intelligent agents.

Reactivity Agents are able to perceive their environment and respond to changes in a timely manner. A reactive agent can will reevaluate its current actions and goals to see if they are still valid and sensible to execute.

Proactiveness Agents are able to initiate actions on their own accord and do not wait or input. Agents that are proactive will take the initiative and work actively towards achieving their goal.

Social ability Agents can interact with other agents and humans to work towards their goals. To do this they can negotiate and cooperate with others.

Another definition comes from Russel[27] who defines an agent as follows:

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

By abstracting agents into entities with sensors and actuates we can reduces the problem of writing an intelligent agent to deciding what action to take based on the percepts that have been received. Because of this agent programs follow a common control structure[34, 27]. This is a loop in which the agent:
2.2 The GOAL Agent Programming Language

GOAL is a logic-based agent programming language for programming cognitive agents [16, 19]. GOAL agents maintain a mental state that consists of beliefs and goals and derive their choice of action from their beliefs and goals. GOAL agents also use a knowledge base to represent conceptual and domain knowledge. The version of GOAL used in this study uses Prolog, a declarative programming language, to represent the knowledge, beliefs, and
goals of an agent. A Prolog program consists of Horn clauses, which are logical rules and simple facts [28].

GOAL is a rule-based language. The design philosophy of GOAL is that writing agent programs essentially means writing rules that determine for each situation that the agent finds itself in what it should do in that situation. Rules are ordered which allows for imposing a priority on what needs to be done first by an agent.

GOAL agents execute a basic reasoning cycle that consists of two phases. The purpose of the first phase is to process all events such as percepts and messages and make sure that the agent’s mental state is up-to-date. In this phase the GOAL agent retrieves and processes all perceptual information available from the environment to update the beliefs and goals of the agent. In the second phase of the cycle agents decide what to do next. Typically, in this phase one environment action is selected and sent to an environment.

The concept of a module is a key programming construct in GOAL for structuring and writing larger agent programs. A module basically is a container for a set of rules. A GOAL agent program then is a set of modules. The event module corresponds to the first phase in the agent’s cycle and is designed to support event processing whereas the main module corresponds to the second phase and is designed to support decision making. In addition, a special init module is available for initialising the mental state and other components of an agent. A programmer can also add and write its own set of modules for structuring and organizing code. GOAL also provides support for communication between agents.

The GOAL language is distributed with an Integrated Development Environment for coding, testing, and debugging. It provides the usual program editing tools as well as tools to analyse and debug code (e.g., creating an overview of predicates used in a program). It also provides debugging tools including introspectors for inspecting agent states, stepping functionality, (conditional) breakpoints, runtime querying and modification of agent states, tracing and logging functionality at different levels of granularity. In addition, the GOAL platform fully supports the Environment Interface Standard (EIS; [3]), which provides an interface for interacting with environments. It facilitates the exchange of actions from agents to an environment and the exchange of percepts from an environment to agents.
2.2 The GOAL Agent Programming Language

2.2.1 GOAL Tutorial

GOAL abstracts the environment in which the agent is situated into percepts and actions. Percepts are facts it receives about the environment. Actions are things it can do to interact with the environment. Figure 2.2 shows an example of an environment, the blocks world. In this environment the agent consists of a gripper suspended over a table. The agent perceives the available blocks in the world and what they are standing on. The agent can pick up blocks and stack these on top of other blocks or on the table.

GOAL uses Prolog as its knowledge representation. For example all information about the blocks world can be represented by a number of facts. These take the form of block(X) which describes the available blocks. Facts of the form on(X,Y) show which block is standing on another or the the table and holding(X) to represent which block is being held by the gripper. Figure 2.2 shows a configuration of the blocks world, for this configuration the following facts apply:

\begin{verbatim}
block(a)
block(b)
block(c)
block(d)
block(e)
on(b,c)
on(c,d)
on(d,table)
on(e,table)
holding(a)
\end{verbatim}

Using Prolog rules we can derive knowledge from these facts. For example using the situation describe above in figure 2.2 we can apply the following two rules:

\begin{verbatim}
clear(table).
clear(X) :- block(X), not(on(Y,X)), not(holding(X)).
\end{verbatim}

The rules derive that block a is not clear while blocks b and e are. These rules can then be used to write simple rules. As an example the rule if clear(X) then pickup(X) will move the gripper to pick up a block that is clear. An ordered set of rules in order make a GOAL program.

The purpose of a GOAL program is to achieve a goal state. This is achieved by using a program to transform the environment from one state into another. Figure 2.3 shows the initial state of the environment and the goal state in which the agent program will try to bring this environment. It will try to build a tower of three blocks. This can be described as a Prolog conjunction on(a,b), on(b,c), on(c, table).

Figure 2.4 shows the overall anatomy of a GOAL program. A program consists of an init module, event module, main module and any number of custom modules. The init module provides knowledge rules and actions that can be accessed in every other module. It also
2.2 The GOAL Agent Programming Language

Figure 2.3: Structure of a GOAL program transform the environment to a specific goal state

```plaintext
limit module {
    knowledge { [10 lines]
    }
    beliefs {
    }
    goals { [2 lines]
    }
    actionspec { [12 lines]
    }
}
main module {
    program { [44 lines]
    }
}
read module {
    program { [9 lines]
    }
}
module custom {
    program { [1 lines]
    }
}
```

Figure 2.4: Structure of a GOAL program
2.2 The GOAL Agent Programming Language

Preliminary

Figure 2.5: The event section of a GOAL program.

```prolog
event module {
    program {
        % assumes full observability
        forall bel{ block(X), not(percept(block(X))) } do delete( block(X) ).
        forall bel{ percept(block(X)), not(block(X)) } do insert( block(X) ).
        forall bel{ holding(X), not(percept(holding(X))) } do delete( holding(X) ).
        forall bel{ percept(holding(X)), not(holding(X)) } do insert( holding(X) ).
        forall bel{ on(X,Y), not(percept(on(X,Y))) } do delete( on(X,Y) ).
        forall bel{ percept(on(X,Y)), not(on(X,Y)) } do insert( on(X,Y) ).
    }
}
```

Figure 2.6: The knowledge section of a GOAL program.

```prolog
knowledge{
    % assume there is enough room to put all blocks on the table;
    % this is true up to 15 blocks in the tower world.
    clear(table).  
    clear(X) :- block(X), not(on(_, X)), not(holding(X)).
    above(X,Y) :- on(X, Y).  
    above(X,Y) :- on(X, Z), above(Z, Y).
    tower([X]) :- on[X, table].
    tower([X, Y | T]) :- on[X, Y], tower([Y | T]).
}
```

provides initial beliefs and goals. The event modules contains rules to process percepts and
and the main and custom modules contain the decision making logic.

The GOAL interpreter uses two Prolog fact-bases. One containing goals and one con-
taining beliefs of the agent. A special subset of the beliefs are percepts. These are received
from the environment at a regular interval. The knowledge rules can query both fact-bases,
in turn the knowledge rules can be used in the event and main module. The event and main
module can both query and modify the goal and belief base.

The event module (fig 2.5) contains the rules that process percepts. The rules have the
form of forall <GOAL Query> do <Action> which executes the action for all results of
the goal query. The GOAL query in this case is a bel<Prolog Query> which queries the
belief base. The actions are insert(<Fact>) and delete(<Fact>).

An example of knowledge rules are shown in figure 2.6. These can be used in the
program section as shown in figure 2.7. The results of the query can be used as the parameter
for an action. When there are multiple solutions a random choice is made. The rules of the
program section are evaluated in order starting from the top. Evaluation stops at the first
rule that results in the successful execution of an action. After the action is evaluated the
program will start the ODA cycle again by fetching new percepts from the environment and
processing them in the event module.

Actions (fig 2.8) provided by the environment are specified in the action spec section of
the program. Actions have a pre and post condition. The precondition must be true if the
Figure 2.7: The program section of the main module.

Figure 2.8: The action spec of a GOAL program.

Figure 2.9: The goals section of a GOAL program.

the action is to be executed. The post condition is used to update the belief base after the action has been executed.

Figure 2.9 shows an example of the goals section. This describes the state of the environment the program should work towards. Goals are removed from the goal base when they are achieved. GOAL will try to build the tower described in the first goal once. Once the tower has been built the goal will be dropped, if the tower is removed it will not be reassembled again. The second goal can not be achieved so the program will try to rebuild the tower if it gets changed. To use the goal base the drop(<Fact>) and adopt(<Fact>) action can be used. To query the goal base the goal query can be used, goal(<Prolog Query>).
2.3 POSH

POSH is a reactive action selection mechanism developed by Bryson[10] to simplify the construction of action selection for modular AI (see [14] for details). A programmer used to thinking about conventional sequential programs is asked to first consider a worst-case scenario for his/her agent, then to break each step of the plan to resolve that scenario into a part of a reactive plan. Succeeding at a goal is the agent’s highest priority, the thing the agent does if it can. Then for each step leading to the goal the same process is followed: a perceptual condition is defined allowing the agent to recognize if it can take the action leading most directly to its goal. Both perceptual conditions and actions are small chunks of code (e.g. methods in Java) done in the underlaying language that is used to control the agent. These make up an agent’s sensors and actions, so-called behavior primitives.

A POSH plan organizes these primitives in a hierarchical rule-based manner. During runtime, the POSH plan is periodically evaluated to determine the action the agent should be executing[10]. This is illustrated in figure 2.10 which shows a small section of a POSH program. On the left side is the low level Java code implementing a perceptual condition and behavior primitive using functionality provided by the underlying framework. These primitives are then both slotted into the POSH program on the right.

Later, Bryson embedded POSH in a more formal development methodology called Behavior Oriented Design (BOD) [11]. BOD emphasizes the above development process, and also the use of behavior modules written in ordinary object-oriented languages (in our case, in Java) to encode the majority of the agent’s intelligence, including its memory. These modules provide the behavior and sensory primitives; method calls are the interface between a high-level POSH plan and the low-level code of the behavior modules (see Fig. 2.10). The high-level POSH plan can be organized hierarchically in a tree like structure.

Recently, a graphical editor for POSH plans has been developed. This version was used in the present study.
2.3 POSH Tutorial

We’ll now provide a brief crashcourse in POSH in combination with the Pogamut Platform.

Figure 5.2 shows a general overview of POSH and Pogamut. The agents are situated in the Emohawk Environment. This is a modification for Unreal Tournament 2004 (the creation of this environment is explained later on). To communicate with external programs the GameBots modification is used\[21, 15\]. Pogamut is a framework written in Java that can communicate with Unreal Tournament through the Game Bots modification. Programming a POSH agent is done in Netbeans. A graphical editor is used to create POSH plans. The regular Java editor is used to create the primitives (actions and senses) that connect the POSH plan with the underlying framework.

As an example of how to use POSH we’ll be modeling the behavior of a monkey. This Monkey is a fictional agent with no existing implementation. This allows us to explain the basics of POSH while avoiding the need to explain the complexities associated with the environment.

The plan shown in figure 2.11 will serve as the running example. Note that POSH originally did not have a graphical plan notation, but used a LISP-like syntax as shown in figure 2.12. A POSH plan consists of a simple tree structure that consists of five elements. These are two primitives: actions, and senses, and three aggregates: a drive collection, competences, and action patterns.

Actions and Senses are the primitive elements of a POSH plan, they form the leaves on the behavior tree. Primitives are always attached to an aggregate as shown in figure 2.13. Senses are used by an aggregate to determine if the associated action should be executed. This action can either be a primitive action or another aggregate.
Actions and Senses need to be implemented in the underlying platform. This can for example be done implementing Actions and Senses as methods that can be invoked by the POSH interpreter. When a Sense is evaluated it must return a boolean value or a number. These can be used as part of a simple boolean expression. Multiple senses can also be used as a trigger, in this case all senses have to evaluate.

The Drive Collection is the root element of a POSH plan and should be seen as a collection of plans. The Drive Collection is evaluated at every iteration of the agent logic and determines what an agent should do. It will keep on doing so until the senses attached to the goal condition evaluates as true. As shown in figure 2.14 the Drive Collection contains sev-
eral Drive Elements, these describe these individual plans. Drive Elements in turn consist of an action and a sense (see fig 2.15). The elements are evaluated from top to bottom, for each element the attached senses are evaluated. The first element for which its senses evaluated a true will be executed. Figure 2.15 shows the plan of a Monkey who’s goal is to live long and prosper. This is a sense that will evaluate to true once the monkey has demised and the execution of the plan will stop. While the monkey is alive it will try to avoid looming things, attack loud things, forage and relax in that order of priority. The avoid looming drive element has been expanded to show that it is triggered when the monkey senses something looming. When triggered it will avoid the looming thing.

Competences serve as basic reactive plans and can be used whenever a primitive action can be used. Much like the drive collection they consist of a number of competence elements. Each element consists of a sense and an action. The elements are evaluated from top to bottom, the first element to trigger is executed. Illustrated in figure 2.16 the forage

Figure 2.15: A drive element is expanded showing the action and sense used to trigger it.

Figure 2.16: A drive element containing a competence as its action.
Figure 2.17: A drive element containing an action pattern as its action.

 Figure 2.18: A simplified overview of the class structure of an agent.

drive element is expanded with the competence for eating bananas. If hungry the Monkey will eat a peeled banana if it has one. If the Monkey only has unpeeled ones it will peel a banana. If it does not have any bananas at all, it will get one.

Figure 2.17 shows an action pattern. Action patterns are chains of actions and can be used anywhere a primitive action can be used. The actions are executed sequentially. Action patterns are useful for reducing the complexity of actions that need to be executed as a sequence. In this case the Monkey will find a banana, peel the banana and subsequently eat it.

The actual creation of actions and senses is platform and implementation dependent. For this example we’ll use a Java implementation with the Pogamut platform.
Actions and Senses are created by annotating methods with \texttt{@POSHAction} or \texttt{@POSHSense} in an agent specific behavior class. This class should extend the MonkeyBehaviour class provided by the Platform. The MonkeyBehaviour comes with various modules that provide access to specific low level functionality which can be used in the action and sense methods (see figure 2.18). The POSH interpreter then uses the Java reflection API to invoke the actions and senses.
Chapter 3

Related Work

This chapter discusses work related to this thesis.

An analysis of the Multi-Agent contest has been done by [2]. This analysis covers high level strategies used and technical problems encountered by the teams. The analysis does not cover the actual implementation or methodologies used.

Several studies have been performed to empirically evaluate agent programming languages. In particular in the past five years more work has been done in this area. Techniques from the programming comprehension community (see e.g., [30][33]) have been borrowed but also more qualitative work has been reported that analyze the use of agent programming languages.

[23] evaluates a toolkit called ACRE for conversation management between agents. The focus of this study is on agent communication and compares performance of a group that used the toolkit with a group that did not have access to the toolkit. Results suggest that the toolkit can improve code quality and reduces the code base but not necessarily effort. The Agent Factory platform was used in the study but use of language features was not analyzed.

[32] presents a qualitative analysis of multi-agent programs for Unreal Tournament written in GOAL by first year computer science students. The main aim of the study was to gain more insight into more practical aspects of agent development and to better understand some of the problems that agent programmers face. The method used in [32] consists of analyzing the end result of MAS development, i.e., the multi-agent program code and discusses the actual use of language features. Results of this analysis were related to the performance of the multi-agent systems in a competition and the grades that students received. The study did not involve an experimental design with programmers as subjects nor a comparison with other languages.

Only a few studies have been performed that compare agent programming languages. [14, 13] empirically compared the use of the agent programming framework POSH with using Java as the language for programming agents that control virtual characters. The aim of the study was to establish whether any differences in usability for programming agents that perform relatively complex tasks in a virtual environment called Emohawk could be found. The study, however, found that subjects using POSH and those using Java performed similarly. Both studies asked subjects to write code from scratch, i.e., no code was provided
Related Work

Initially to the subjects. The second study in addition included a comprehension task where subjects were asked questions about the code they had written. Both studies lasted eight hours, making fatigue a potential factor in the results.

Apart from empirical comparison, analytic comparison of programming languages can be performed. We mention the work of Bryson [11] which presents a purely theoretical comparison between POSH and other action selection mechanisms such as Environmental Determinism, Finite State Machines and Basic Reactive Plans. POSH was found to be more expressive than these other frameworks for programming reactive agents.

As we use a virtual environment called Emohawk that runs on top of the Unreal gaming engine, we briefly mention related work in the area of programming gaming agents. [17] evaluated a tool called BehaviorShop and showed that the IDE allowed users with little to no experience in Artificial Intelligence to create high-level behaviors that control virtual characters. Subjects that were only given a brief lecture on the basic concepts of BehaviorShop were able to create fairly complex interactions. In our study we also found that given more general experience with programming a short tutorial was sufficient for subjects to code agents that control virtual characters. Our study is comparative, however, and different from [17] investigates also which language features are used.
Chapter 4

Analysis of the Multi-Agent Programming Contest

In this chapter we analyze the code submitted to the Multi-Agent Programming contest (MAPC). The challenge the MAPC presents us with is that the participants use several different languages, methodologies and frameworks to tackle a competition. Additionally there are no controls on the team size or the available time for each team. This makes it hard to point out a single factor that contributes to their performance.

As such we’ll take a broad view and try to discern common traits in successful teams. For this we look at the agent platforms, languages and methodologies used and how these assist the participants in their implementation. In the implementation we look for commonly observed patterns in the code.

To answer our main question we’re interested in knowing what a programmer is typically working on. This will allow us to better understand what a typical agent programming task is. To this end we classify sections of code in an agent program and measure the size of each category relative to whole program. The categories are based on concepts used in the literature to discuss agent programming.

4.1 Multi-Agent Programming Contest

The MAPC has been organized yearly since 2005. From 2005 to 2008 the contest was organized as part of a conference workshop. From 2008 to present the MAPC provides a scenario and simulated environment. Teams from participating universities develop a multi-agent system which will compete against each other. As part of the competition, teams publish their code and write a report on their implementation, this makes the MAPC a suitable area for study.

The background story of this year’s scenario features agents on Mars trying to find the best sources of water on Mars and hold onto these. To achieve their goal a team of ten agents can use a number of different vehicles with different abilities. The participants have to develop an agent or team of agents that can control these vehicles.

The Mars environment is abstracted as a graph as shown in figure 4.1. Each node has a
value that represents the amount of water available, each vertice a value that represents the cost of traveling over it. The overall goal is to control zones of the graph by placing agents in appropriate positions. Rewards are handed out each round for holding the largest area of water sources and for certain achievements. The team that has gathered the largest number of points at the end of the simulation wins.

The vehicles the agents are using have a value for its energy, health, strength and vision range. Energy is used to perform actions, different actions have a different cost. Health determines if an agent can perform all actions. When an agent has no health left only a few actions are possible. The strength influences the damage a sabotage action will inflict. The vision range determines how far an agent can see.

The vehicles can also execute a set of basic actions, recharge, goto, survey, buy parry and recharge. The recharge action will restore a part of the agent’s energy. Goto moves the agent across the map. Survey reveals the cost of the edges of the current node. Buy lets the agent improve its stats. Parry block an incoming attack. Finally recharge restores the vehicles energy.

There are five different types of vehicle available, each with its own special actions. The surveyor has extra energy and can probe nodes to determine how much water is available on it. The repairer can repair other vehicles. The saboteur can attack other vehicles. The sentinel has extra vision range. The inspector can inspect other vehicles to learn their details.
The competition itself uses a client server architecture that uses an XML protocol for communication. The server runs a simulation of the environment. At a regular interval the server moves the simulation a step further and sends information about the environment to the clients. The clients can then send a message to the server to indicate what their next action will be. The process is shown in figure 4.2. The participants can either write their own implementation of the client to send, receive and parse the XML messages or they can use an (Environment Interface Standard) EIS interface that has been provided by the competition.

The competition organized a series of friendly competitions during the development phases that allowed teams to test their connection to the server and test their agents against each other. At the end of the development a qualification round was held to ensure all teams were able to maintain a stable connection which was followed by the actual competition. The competition used a round robin setup in which each team faced every other team. The winner was determined by adding up the scores of all competitions.

4.2 Methodologies

A methodology describes a set of guidelines that can be used to solve a problem and thus significantly influences how each team develops their solution for the MAPC. As part of the contest each team was asked to describe the methodology they used, these answers have been collected in table 4.1.

TUB, Simburg, Sorena and Nargel have chosen to use an agent-based methodology. As a group these teams did not do very well.
4.3 Code Inspection

When inspecting the code of the participants we look for common patterns and structures. To do this efficiently we look at the overall agent structure, the implementation of the observe-decide-act cycle (ODA) and comment on the overall readability of the code. We’ll also comment on any structural oddities we find while inspecting the code.

4.3.1 HactarV2

Agent structure

The agent is written in GOAL and consists of 20 files. The general structure of the agent consists of an agent file describing two modules. One of which serves to initialize the agent. The other describes how the agent should process percepts and decide on its actions. Additionally this agent file imports several other files containing Prolog knowledge rules and specific actions for each agent role.

Cycle implementation

GOAL provides an ODA cycle on which HactarV2 implements its own ODA cycle. This cycle keeps HactarV2 in sync with the competition server.

In this cycle all percepts are processed and a limited set of percepts is communicated to other agents. Finally an action is decided upon. This cycle also to handles the reset control-percept from the competition server by executing a reset action.
Percept processing is done in two stages. First the common case for each role is processed. Then a role specific module handles the remainder of the percepts. The same applies to communication and action selection.

The actual action selection is done by a set of if-condition-then-action rules. These rules are are listed in order of priority and have been grouped together in modules which each describe a specific high-level action.

**Readability of the code**

While larger then Python-DTU the code of HactarV2 is one of the shortest, some 1826 lines, which makes reading the code a brief and uncomplicated affair. Additionally the code has been very well documented (one line of comment every two lines of code) and has a clear structure.

**Code duplication**

Overall the code has been fairly well structured and HactarV2 has attempted to remove duplication by separating the action rules into common and role specific parts. Two elements stand out. The dijkstra.pl file contains eight different implementations of Dijkstra’s path-finding algorithm. Each finds a shortest path to a different item such as a nearby enemy or unprobed node. It is unclear why this code has been duplicated.

### 4.3.2 Python-DTU

**Agent structure**

The agent is written completely in Python. The general structure of the agent consist of an Agent class which inherits functionality from the Thread, Communicator and Graph classes.

The Agent class contains the main cycle, decision system and initialization. It also contains the functionality to deal with a disabled agent. The Thread is a python standard class allowing each agent to run concurrently with other agents. The Communicator provides communication with the contest server and process percepts. The Graph contains the relevant code to maintain the graph that represents the game world and provides functions to enumerate possible actions for the agent on the graph.

**Cycle implementation**

The Python-DTU agents implements a ODA cycle in the Agent class. On each iteration this loop receives a message from the server. If this is a control message the agent is started, reset or stopped. Otherwise the message is passed on to the Communicator for percept handling.

The percept handling stores all percepts in the agents memory, updating the map and the position of all other agents on it. The percept handling also serves to share this information with other agents by inserting these percepts into their memory.

Decisions on which action to execute are made at several points in the cycle. The agent being disabled is treated as a special case and is handled separately. For regular actions
the implementation uses an auctioning system. The purpose of the auctioning system is to ensure the agents collectively make optimal decisions. To do this each agent determines 10 possible actions and their cost to the agent. An algorithm then compares all from all agents and determines for each agent the best action. It does however take special care to prioritize repair, attack and probe actions. The action selected by the auction is not directly executed. If the agent needs to recharge, buy upgrades, survey or parry it will do this instead of the action selected by the auction. This is suboptimal as an agent under attack might claim a valuable action it is unable to execute. It appears this functionality was added later on in the development.

Readability of the code

Overall the code of Python-DTU is short, a mere 1373 lines, which makes reading the code a brief and uncomplicated affair. The hardest parts of the code to understand were the handle disabled, get goals and do action functions. These were filled with deeply nested if-statements and algorithms to derive knowledge from the percepts. Additionally, a lack of comments made these hard to understand.

4.3.3 TUB

Agent structure

The team used Java and the JIAC V framework to implement their Agent. JIAC V supports a component based architecture. Components in this architecture are classes called Beans. Using XML configuration files it is possible to define what components are used in an agent and how they should connect to each other.

The agent consists of an ServerCommunicationBean and an ontology of Message classes. A PerceptionBean and a set of role specific DecisionBeans and State classes that describe actions the agent can make.

The ServerCommunicationBean class parses the XML messages to their corresponding Message classes. The PerceptionBean class processes these further.

Decisions are made by the a DecisionBean. Each agent contains a number of DecisionBeans, one for each role. Each is initialized with a list of State classes appropriate for that role. Each State class describes a high level plans and provide an method to test if the plan is applicable. The DecisionBean tests all State classes for their applicability and the first that holds is.

Cycle implementation

Rather then implementing a thread which calls each component in turn, each component calls the next component. The ServerCommunicationBean waits for a message from the server. This message is then parsed and passed onto the PerceptionBean. The perception bean handles the control percepts, shares the other percepts and updates a world object that represents the state of the world. In turn the PerceptionBean calls a DecisionBean. Which DecisionBean is called depends on the role of the agent.
Observations on Agent structure

While the agent is well structured there are several oddities when interacting with the JIAC framework.

To allow loose coupling the framework provides functionality to look up an action registered by a bean[31]. These actions can then be called without needing a direct reference to the bean. Yet in the initialization of each bean TUB implemented a loop that searches all registered beans and stores these in a class variable, essentially duplicating functionality of the framework.

The framework also provides a basic memory in which objects can be stored. This memory can be queried by using a template and is accessible from any class that extends the AbstractAgentBean class. While the World object is written to this memory, nothing is ever retrieved from it. Rather the DecisionBeans and State classes are all provided with a reference to the World object instead. Again duplicating functionality that is already provided by the framework.

The different state classes also feature a rather interesting pattern of duplication. Typically a number of percepts are queried from the World object and on this data simple search functions and queries are executed. For example finding the closest opponent or testing for presence of an bot in the same position. This pattern is repeated several times over rather then written as a function of the World object.

Readability of the code

With close to 10k lines of code the TUB has written the longest agent of all teams. However the code is generally well structured and the critical parts describing the agent behavior and decision making are well commented.

4.3.4 Bogtrotters

Agent structure

The Bogtrotters team consists of two agents. A leader and an in-situ agent. Both make use of a MapService and GroupService. These are objects shared between all agents and used to share information.

The MapService provides information about the maps, path planning and other analysis. The GroupService is used to organize the agents into groups on the maps to optimize the control of the terrain. Both services add also sensors that create additional percepts relevant to their working.

The Leader is written in AF-AgentSpeak and manages the overall strategy. The in-situ agents is written in AF-TeleoReactive. The in-situ agents either respond to requests from the leader and move to the locations needed to hold an area or follow their own role specific behavior. The Map- and GroupService are written in Java.
Cycle implementation

Agent Factory provides a ODA cycle and the Bogtrotters make use of this. Percepts are received and processed. Based on these percepts additional percepts are generated by the Map- and Group services. Based on the available percepts the agents rules are evaluated. The first to trigger is executed. The execution of the actual action is delayed until all agents have selected an action.

Which action an agent selects depends on their role and orders from the leader. Before executing actions from the leader (typically) the agent checks based on its role if a role specific action needs to be done. Only when it has nothing to do it will move to the location the leader requested. What the agent should do is described with a simple priority based rule set.

Readability of the code

With 5561 lines of code the agent contains a modest amount of code and is well structured. Even so it is hard to understand what this agent is doing. This comes in part from unfamiliarity with the Agent Factory framework to which the agent is strongly connected. The lack of useful comments make the agent even harder to understand. The only available comments are code that has been commented out.

4.3.5 HempleSofa

Agent structure

HempleSofa has written their agent in Java. The structure of the agent is centered around a StrategyBasedAgent class which extends the Agent class provided by the contest. The StrategyBasedAgent contains the main loop, a Graph used to represent the environment and an ActionGenerator which describes a several high-level actions.

The Strategy class consists of a number of variables which describe the world and a number of the agents preferences. Based on these preferences the agent decides whether to upgrade, attack, defend, ect.

The StrategyBasedAgent has subclasses for each role which are required to implement methods that define how the agent should upgrade, attack, defend, etc. These methods then typically use the ActionGenerator to invoke a high-level actions which then sorts out the details.

Cycle implementation

The cycle is implemented in the code provided by the agent contest. The agent logic is placed in a method that will be called once in each round of the competition. In this method the percepts are processed, messages sent and received. Once the state of the world is known, the agent preferences are revised. Then based on the preferences a strategy is chosen. The strategy checks a few conditions after which it choses an action.
Readability of the code

While the code is generally well commented, as a result of the chosen structure a lot of functionality has been thrown together making it hard to discern what happens where. Additionally due to the chosen system of preferences and strategies it is not clear how an agent will actually react in a certain situation.

4.3.6 Simburg

Agent structure

The Simburg agent was written in Java. The structure of the agent is based on the basic structure provided by the contest. The Agent class has been modified to include several lists of Do objects. For each role a subclass of the Agent class has been created. These subclasses fill out the lists with Do objects. Do objects represent simple actions the bot can take.

Cycle implementation

The cycle is based on the basic structure provided by the contest. First percepts are processed into beliefs and shared with team mates via messages. The messages from team members are processed into beliefs. The agent then consults the list of Do objects in order. The first action provided by a Do object is executed. Do objects determine what action should be executed based on (sometimes deeply nested) if-else-statements.

Observations on Agent Structure

The Agent class contains a number of functions with the nearly identical names and functionality as the some Do objects. The functions are no longer used in the agent. As such it would appear that some refactoring has taken place and the old code never cleaned up.

Readability of the code

Overall the structure of the agent as easy to understand. However the agent was written with little abstraction. As a result the agent has extremely long long methods and deeply nested decision structures. This made it overall hard to follow what the agent was doing.

4.3.7 Sorena

The code submitted to the Agent contest by Sorena was generated by the JIAC[20] framework. No original source code was available, because of this it is not possible to make a proper analysis of Sorena.
4.3 Code Inspection

4.3.8 D3lp0r

Agent structure

The structure of D3lp0r is hard to understand. The agent consists of two parts. One part is written in Python and handles communication with the contest server, manages the live cycle of the agents and aggregates the percepts prior to providing them to the agent. The other part is written in Prolog and contains the decision logic for the agent. Each role has it’s own Prolog implementation. Part of the Prolog implementation makes use of DeLP to decide between conflicting options.

Cycle implementation

Of all teams D3lp0r implements the most complex ODA cycle. Percepts are received from the environment and aggregated in a percept server. The percept server builds a state of the world based on the collective information. Each agent will then determine if it needs to reevaluate what it is currently doing. If this is the case it will execute the action associated with it’s current intention. Otherwise the agent will look for a new intention. This is done by evaluating the available plans and using DeLP to decide between these. The winning plan then becomes the new intention of the agent and is executed.

Readability of the code

Approximately half the code and comments are written in Spanish. For some one not speaking Spanish this made it to understand some parts of the agent. The use of Prolog difficult to understand the overall flow of program. The explanation in the report did however clear up most of the structure.

4.3.9 Nargel

Agent structure

Nargel is written in Java and makes use of JADE. JADE is framework that provides middleware for communication and cooperation of agents[4]. The structure of Nargel is rather poor. The program consist of an Agent class provided by the contest, a SimpleExplorerAgent and ten nearly identical numbered Sender# and Receiver# classes.

The Sender classes constantly check if a certain static variable contains a message. If such a message is present it is sent to a Receiver using JADE. The Receiver then places this message in another static variable. It is unclear why the sendMessage and receiveMessage functions provided by the Agent class were not used.

Contrary to expectations the SimpleExplorerAgent class contains the logic for all roles. For each role a function is defined in which the percept processing, communication and action selection is done. This results in large sections of identical code.
Readability of the code

The poor structure of the agent makes it hard to understand what responsibilities are located where. Additionally, combining percept processing, communication and action selection in a single method creates a long, deeply nested, complicated section of code.

4.3.10 Results

Overall the inspection showed a great variety in the structure and organization of the code. However, three patterns are common for all teams.

Modularization

The agents in each team have different capabilities. These require that a different behavior is implemented for each of these. Some parts of these behaviors also have much in common. These common parts are often grouped into modules which can be reused. These are variously named Modules (HacatarV2), Do (Simburg), States (TUB).

The modules used by HacatarV2 contain a list of if condition then action rules which are evaluated in order. Each action may refer to an action in the environment or another module. The modules are used to represent high-level actions or complicated behavior and routines.

The Do’s used by Simburg are all subclasses of a single Do class. These describe high level actions such as “Chase Down an Opponent” and “Inspect an Opponent”. Depending on the role, each agent is initialized with a list of these actions which are evaluated in order. When evaluating a Do it can either result in a valid action or a null if this module has no possible action. The evaluation of Do objects continues until one has produced a valid action, this action is then executed.

The States implemented by TUB are quite similar to the Do used by Simburg. The major difference is that where the applicability of a Do object is determined by its ability to produce an action, State objects will always produce an action and have an accept method to test if they are applicable to the current situation.

HempleSofa and Python-DTU did describe high level behavior such as “Obtain an achievement” and “Take an offensive action” but did not do this in a reusable fashion.

Arbitration between different actions

In each round of the competition each team tries to work out which action is to be taken and often more than one action is possible. There are several common methods found to arbitrate between these.

HacatarV2, Simburg and TUB use modules to describe high level actions and their ordering determines which is prioritized. The first module that is applicable is executed. The modules of HacatarV2 itself consists of a list of condition then action rules which are prioritized in the same fashion.

Python-DTU and HempleSofa implement an algorithm to determine which action is executed. Python-DTU uses a graph search algorithm to discover a number of possible actions.
and an auction algorithm to determine the most optimal action for each agent. HempleSofa
uses a system which maps information from the environment onto a preferences for each
action. An algorithm then decides which preference wins and executes the action associated
with that preference.

However the most common and widely used technique, along with the other two meth-
ods of arbitration, is the direct use of the control flow features of the programming language
itself. In simple terms, most teams used (deeply) nested if statements and loops, often
copy-pasted, to decide which action should be executed and how it should be executed.
A particular good example of this is the Nargel team. For each role they implements the
percept processing, communication and action selection in a single method.

4.4 Categorization

To be able to generalize how different teams used their language of choice we attempt to
categorize their code by the function it fulfills. This is done by inspecting the code that the
teams submitted to the contest and placing annotations to indicate the function it performs.
These files are then passed through a line counting script.

While the categorization was done with great care, it does depend on understanding the
structure and purpose of the code and is therefor subjective. The purpose of this analysis
is to facilitate the the design of a qualitative comparative analysis of POSH and GOAL in
chapter 5. This requires a generic indication of the prevalent type of code in an Agent. As
such a we did not use multiple raters to allow for inter-rater reliability analysis and reduce
the subjective nature of the analysis.

4.4.1 Categories

Typically agents are discussed in the literature using terms such as actions, percepts and
knowledge [26, 12]. We use these concepts to categorize the code of the different teams.
As such we can established action selection, knowledge derivation and percepts handling
as functions one would expect to be able to identify. The categories communication, initial-
ization and import were added later after inspection showed that a significant portion of the
agent code would fall into these categories.

To illustrate these categories we’ll use pseudo code to describe an agent from the agent
contest. This agent is written for the sentinel role and its goal is to attack any opponent it
sees. To keep the example brief, code has been simplified. The agent won’t recharge or
repair and mindlessly attack the nearest opponent.
Action selection

Code is categorized as action selection when it determines which action to execute. This may include code that determines how to apply the selected action. It does but does not include code which sends the action to the competition server. This example illustrates a simplified form of what can be found in most teams and is classified as action selection. When there are visible opponents, the agent will approach and attack the nearest opponent.

```python
// (action)
def selectAction():
    if not visibleOpponents.isEmpty():
        closest = visibleOpponents.first()
        for enemy in visibleOpponents:
            if distance(closest) > distance(enemy):
                closest = enemy
            if closest.position == self.position:
                attack(closest)
            else:
                goto(closest)
        else:
            wait()
```

Knowledge derivation

Code that derives some information about the environment without processing percepts or selecting actions itself is categorized as knowledge derivation. This example is functionally the same as the previous example, however, it has been re-factored such that testing for presence of visible opponents and the selection of the closest opponent is now done by separate functions. The re-factoring has made their purpose explicit and as such these functions can be categorized as knowledge derivation.

```python
// (action)
def selectAction():
    if hasVisibleOpponentsInRange():
        attack(closestVisibleOpponent())
    elseif hasVisibleOpponents():
        goto(closestVisibleOpponent())
    else:
        wait()

// (knowledge)
def hasVisibleOpponentsInRange():
    return hasVisibleOpponent() &&
```
4.4 Categorization Analysis of the Multi-Agent Programming Contest

```python
closestVisibleOpponent().position == self.position

/// (knowledge)
def hasVisibleOpponent():
    return not visibleOpponents.isEmpty()

/// (knowledge)
def closestVisibleOpponent():
    closest = visibleOpponents.first()
    for enemy in visibleOpponents:
        if distance(closest) > distance(enemy):
            closest = enemy
    return closest
```

**Percept handling**

Percept handling code is called from the main loop, typically before action selection. It takes the percepts from the environment, processes them and optionally stores them as believes. A simple case is illustrated in the example below.

```python
/// (percept)
def processPercepts():
    visibleOpponents.clear()
    for percept in getPerceptsFromEnvironment():
        if percept instance of Opponent:
            visibleOpponents.add(percept)
        else if percept instance of Self:
            self = percept
```

**Communication**

Code used to exchange information between two agents is categorized as communication. This can be accomplished by writing and reading shared memory, or by using message passing as shown below.

```python
/// (communication)
def sendMessages():
    for teamMate in teamMates:
        sendMessage(teamMate, visibleOpponents)

/// (communication)
def receiveMessages():
    for message in messages:
        visibleOpponents.addAll(message.visibleOpponents)
```
Import statements

Upon inspection it was found that a non-trivial number of lines were used to include code from other sources into an agent. This was especially noteworthy in the teams using Java. To avoid inflating the other category these were assigned their own import category. They may look something like this example.

```python
import A...
import Z
```

Initialization

Upon inspection a non-trivial amount of code was used to initialize various components. This is especially common in object oriented languages. We show a trivial example.

```python
def SimpleAgent():
    reset()

def reset():
    visibleOpponents = set()
    self = None
```

Other

Finally a category was added for code that does not fit in any of the above categories. This may include code to connect to the server, drive the ODA-cycle and, visualize or debug the agent as shown in the sample below. These were not included in their own individual categories as they are not directly related to the problem of programming an agent. Agent frameworks and languages also typically provide some or all of these features reducing the value of comparing individual features.

```python
def mainLoop():
    connectToServer()

    while True:
        processPercepts()
        sendMessages()
        receiveMessages()
        defineGoals()
        selectAction()
```
4.4 Categorization Analysis of the Multi-Agent Programming Contest

```python
drawGraph()
dumpAgentState()

// ( other )
def drawGraph():
    // Draw graph code

// ( other )
def dumpAgentState():
    // Dumps the state of the agent
```

### 4.4.2 Counting Lines of Code

Lines of Code (LOC) is a simple and widely used measure of a project's size. There are several standard tools for counting lines of code such as loc, cloc, CodeCount and sclc. These tools do not allow for categorization. A simple counter was written to do our counting. This counter identifies three types of lines, namely code, comment or empty which are defined as follows.

**Comment** Any line defined as a comment in the original language that has three or more characters. This ensures that lines containing long comment start and end symbols such as /* and */ are not counted as actual commentary.

**Code** Any line that is not a comment and is longer than three characters.

**Empty** A line is considered empty when it has three or fewer non-whitespace characters.

```python
empty: /*
comment: * Selects the action for this
comment: * agent. When there are visible
comment: * creeps it will attack the closest
empty: */
empty:

code: def selectAction():
code: if hasVisibleCreeps():
code: attack(closestVisibleCreep())
code: else:
code: wait()
empty:
```

As with any line count there are several pitfalls to take into account when comparing the results.
Conciseness Different programming languages use different LOC to do the same. For example the proverbial “HelloWorld” program in Python takes up only one line, while the same in Java needs seven.

Formatting Depending on the language and formatting used, a simple one-liner can be split out across several lines, or a complete program can be written on a single line.

Generation There is a difference between hand written and generated code. It is possible to generate a large amount of code with little actual effort.

Comments Well documented code is longer than undocumented code without increasing the complexity or work spend on the code itself.

Formatting is not assumed to be an issue, as general coding practices include the use of sensible formatting. For generated files, only the source from which they are generated will be included. To handle the differences in conciseness and account for the difference in comments the categorization will be normalized. We’ll look at the size of each category relative to the size of all lines counted as code as illustrated by formula 4.1.

\[
\text{categorynormalized} = \frac{\text{lines counted as code with category}}{\text{total lines counted as code}}
\]

4.4.3 Results

Categorizing the teams proved to be fairly straightforward. A sensible formatting was used by all teams. Functions names and comments provided enough information to assign categories to the code. The Sorena team only provided generated code and as such has been excluded from the analysis.

Two teams used a combination of two or more languages. The D3lp0r team used a combination of Python and Prolog. Python was used to initialize and control (categorized as other) the agent. While Prolog was used for the actual reasoning. The Bogtrotter use Java in combination with AF-AgentSpeak and AF-TeleoReactive. The latter two are exclusively used to select actions for the agent.

The results in table 4.2 shows the relative size of each category. The distribution hints at the different approaches taken by the teams. It can be seen that for most teams 5%-15% of their code categorizes as other. This code is used to setup the necessary infrastructure to allow communication with the contest server, start agents and facilitate debugging. The exceptions here are HactarV2 which used a platform that provided these functions and D3lp0r which created additional infrastructure in the form of a percept server to preprocess percepts to their agents.

Initialization takes up between 0.0%-10% of the code. The exception here is Simburg which defines a large number of modules that describe high level actions for each agent role. These all need to be initialized and connected to the role.

The analysis also shows that not all teams implement the same features. They may use a platform or language to abstract certain parts away or develop their own system. As such it is not possible to compare the results directly against each other. Instead we’ll consider
4.4 Categorization Analysis of the Multi-Agent Programming Contest

Figure 4.3: The distribution of action and knowledge in the core agent categories.

The results for only the core agent categories listed in 4.3 again are indicative for different approaches. Most teams used between 1%-3% of the code for communication. This is due to the relative simplicity of the communication. For most teams communication consists of simply sharing all or a specific set of percepts with the other agents in the team. This is done either by broadcasting all percepts or using a shared memory. HactarV2 uses a more selective approach, communicating only those percepts that are of interest, resulting in a larger code base. D3lp0r uses a percept server that aggregates all percepts for all agents before passing them to the agents, eliminating the need for communication. Bogtrotters use a white-board and have their communication otherwise closely integrated with the percept processing. Nargel has duplicated its communication code for each agent role.

For most teams percept processing accounts for 8%-13% of the code. The exceptions here are the Bogtrotters for which the platform makes the percepts available and Nargel which has duplicated the percept processing code for each role.

Overall the knowledge and action categories take up most of the code, around 90% for each. When looking at their distribution in figure 4.3 it can be seen that code written for action selection and knowledge derivation have apparently inversely linear relationship. Five teams are clustered between 35%-50% knowledge derivation and 35%-50% action selection.

The Bogtrotter are an outlier as they use Java in combination with AF-AgentSpeak and AF-TeleoReactive. The latter two are exclusively used to select actions for the agent. Both two languages are more concise then Java. Nargel is an outlier due to it’s structure that
<table>
<thead>
<tr>
<th>Team</th>
<th>HactarV2</th>
<th>Python-DTU</th>
<th>TUB</th>
<th>Bogtrotters</th>
<th>HemplesSofa</th>
<th>Simburg</th>
<th>Sorena</th>
<th>D3lp0r</th>
<th>Nargel</th>
</tr>
</thead>
<tbody>
<tr>
<td>knowledge</td>
<td>38.1%</td>
<td>33.9%</td>
<td>25.3%</td>
<td>40.4%</td>
<td>36.7%</td>
<td>10.5%</td>
<td>N/A</td>
<td>21.5%</td>
<td>1.8%</td>
</tr>
<tr>
<td>communication</td>
<td>9.3%</td>
<td>1.3%</td>
<td>1.7%</td>
<td>0.0%</td>
<td>1.8%</td>
<td>1.3%</td>
<td>N/A</td>
<td>0.0%</td>
<td>36.2%</td>
</tr>
<tr>
<td>initialization</td>
<td>4.5%</td>
<td>9.7%</td>
<td>5.3%</td>
<td>0.4%</td>
<td>1.2%</td>
<td>18.6%</td>
<td>N/A</td>
<td>5.2%</td>
<td>22%</td>
</tr>
<tr>
<td>believes</td>
<td>0.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.1%</td>
<td>N/A</td>
<td>0.0%</td>
<td>0.1%</td>
</tr>
<tr>
<td>percept</td>
<td>10.3%</td>
<td>10.1%</td>
<td>7.6%</td>
<td>1.3%</td>
<td>6.1%</td>
<td>5.3%</td>
<td>N/A</td>
<td>4.7%</td>
<td>30.5%</td>
</tr>
<tr>
<td>other</td>
<td>0.0%</td>
<td>7.1%</td>
<td>13.5%</td>
<td>33.8%</td>
<td>17.6%</td>
<td>6.2%</td>
<td>N/A</td>
<td>50.7%</td>
<td>5.5%</td>
</tr>
<tr>
<td>action</td>
<td>34.5%</td>
<td>35.4%</td>
<td>37.0%</td>
<td>17.1%</td>
<td>32.2%</td>
<td>44.3%</td>
<td>N/A</td>
<td>16.0%</td>
<td>22.0%</td>
</tr>
<tr>
<td>import</td>
<td>2.0%</td>
<td>2.5%</td>
<td>9.5%</td>
<td>7.3%</td>
<td>4.2%</td>
<td>13.8%</td>
<td>N/A</td>
<td>0.3%</td>
<td>1.7%</td>
</tr>
<tr>
<td>total lines</td>
<td>1826</td>
<td>1373</td>
<td>9745</td>
<td>5554</td>
<td>8482</td>
<td>5945</td>
<td>N/A</td>
<td>8377</td>
<td>3553</td>
</tr>
</tbody>
</table>

Table 4.2: Percentage of code in each category for each team.

<table>
<thead>
<tr>
<th>Team</th>
<th>HactarV2</th>
<th>Python-DTU</th>
<th>TUB</th>
<th>Bogtrotters</th>
<th>HemplesSofa</th>
<th>Simburg</th>
<th>Sorena</th>
<th>D3lp0r</th>
<th>Nargel</th>
</tr>
</thead>
<tbody>
<tr>
<td>knowledge</td>
<td>41.0%</td>
<td>42.0%</td>
<td>35.3%</td>
<td>68.8%</td>
<td>47.7%</td>
<td>17.1%</td>
<td>N/A</td>
<td>51.0%</td>
<td>2.0%</td>
</tr>
<tr>
<td>communication</td>
<td>10.0%</td>
<td>1.6%</td>
<td>2.4%</td>
<td>0.00%</td>
<td>2.4%</td>
<td>2.1%</td>
<td>N/A</td>
<td>0.00%</td>
<td>40.0%</td>
</tr>
<tr>
<td>believes</td>
<td>0.7%</td>
<td>0.00%</td>
<td>0.0%</td>
<td>0.00%</td>
<td>0.0%</td>
<td>0.2%</td>
<td>N/A</td>
<td>0.00%</td>
<td>0.1%</td>
</tr>
<tr>
<td>percept</td>
<td>11.0%</td>
<td>12.5%</td>
<td>10.6%</td>
<td>1.7%</td>
<td>8.0%</td>
<td>8.6%</td>
<td>N/A</td>
<td>11.3%</td>
<td>33.7%</td>
</tr>
<tr>
<td>action</td>
<td>37.2%</td>
<td>43.9%</td>
<td>51.7%</td>
<td>29.1%</td>
<td>42.0%</td>
<td>72.1%</td>
<td>N/A</td>
<td>27.7%</td>
<td>24.3%</td>
</tr>
</tbody>
</table>

Table 4.3: Percentage of code in categories that make up the core agent.
4.5 Discussion

The code categorization seems to work well to summarize the structure of an agent program. Differences in the size of each category in the results are indicative for different approaches. Both HactarV2’s selective communication and the Bogtrotters lack of percept processing showed up clearly. The outliers we found on action selection were due to genuine abnormalities as revealed in the code inspection. Both Nargel and Simburg duplicated large sections of code. The method resulted in false positives when different languages were used in a single agent such as the Bogtrotters.

A major disadvantage is that the categorization needs to be done by hand making it time consuming. This could be automated in a language with a stronger separation between the different functionalities.

The action and knowledge categories describe most of the agents. Looking at the distribution of code between those two categories we can see a cluster of five teams. This cluster is centered around approximately 45% knowledge derivation and 40% action selection. Knowledge derivation is a category of code that derives knowledge about the environment. Code categorized as action selection decides which actions to take based on this knowledge. This strongly suggests that the main problem for all teams lies in writing the code for action selection and knowledge derivation. The next question is how they handled this.

When looking at the methodologies used by the teams using agent based methodologies were not very successful. This is hard to understand. It is reasonable to expect that teams using a methodology specifically made for agent programming to outperform teams that use a generic or no specific methodology at all. Taking a second look however shows that with the exception of JIAC used by TUB all agent methodologies are based on a waterfall model of development. As a group, teams using iterative methodologies were clearly more successful. This suggests that iterative methodologies are better suited for agent development.

While the participants recognize the utility of using abstraction, they prefer to avoid it. This is evidenced by the direct use of programming language control flow features as the primary means to make decisions. The most common abstraction found was the use of modules to describe high level actions. Most teams also use simple priority based arbitration to decide which module goes first. This suggests that an abstraction most participants will find easy to understand is one based on modules and simple prioritisation. This is in line with findings from a previous study[18].

The importance of action selection can also be seen in the way participants documented their code. Code involved in action selection was consistently better documented then other sections of code. This suggests that the use of a secondary notation is important and effective in clarifying how actions are selected.
Now we can focus on the main question. We have established that the primary function in agents programs is action selection and knowledge derivation. We found that iterative methodologies are more effective. So to effectively compare two agent programming languages we need to measure how they compare on tasks that stress these aspects. So we need to design tasks designed that focus on creating code that is used for action selection and knowledge derivation. These tasks need to do so in an iterative fashion where each task continues on the result of the previous task. We’ll discuss the specifics of these tasks in the next in section 5.3.2.
Chapter 5

Comparing Programming in POSH and GOAL

In this chapter we report on a study we performed in which the agent programming frameworks POSH [9] and GOAL [19] are compared empirically. Both languages take a different approach to agent programming. As part of Behavior Oriented Design POSH is based on a two layer system that consists of a high level rule-based planning layer on top of a layer that provides support for executing low level actions and senses. GOAL on the other hand is based on the BDI agent paradigm and provides a rule-based language on top of a logical language for representing knowledge (using Prolog).

The main aim of the study has been to analyze whether the different sets of language features these frameworks offer impact the performance of programmers on the same set of tasks. Moreover, we are interested in gaining a better understanding of the learning curve associated with agent-oriented programming languages. Given that the POSH framework builds directly on top of Java whereas the GOAL language uses Prolog for representing knowledge, we hypothesized that the learning curve would be less steep for POSH than for GOAL. More in particular, we expected novice programmers in POSH to perform better than novice GOAL programmers. For the coding tasks that we designed, we expected this difference to disappear for more advanced programmers that received more extensive training in either framework.

The study consists of a comprehension task and a set of increasingly more complex coding tasks. The tasks require subjects to program an agent that controls a character in a virtual environment called Emohawk run on top of the Unreal engine. Emohawk provides a fairly complex environment which however is not as complex as the Unreal Tournament gaming environment.

Subjects were provided with a simple agent program and asked to inspect and extend it. Extending the agent was done in a number of iterative steps. This limited the time requirement as participants didn’t have to create functionality from scratch while also allowing progress to be measured in a simple fashion. During the experiment we collected data on the time it took to complete each assignment and asked participants to rate the usefulness of a feature, provided it was used.

The chapter is organized as follows. Section 5.1 informally compares the two agent pro-
5.1 Comparison of POSH and GOAL

From the introduction in 2 it may not be immediately clear that both GOAL and POSH provide similar features for writing agent programs. Both programming frameworks are rule-based. Rules are used in POSH as well as in GOAL to decide which action the agent
Comparing Programming in POSH and GOAL

5.2 Emohawk Environment

has to perform in a particular situation. The terminology used, however, differs considerably. Still it is possible to conceptually link various language features of both languages. To this end we introduce five key concepts that relate to general aspects of programming an agent and discuss how each language covers these aspects. To further support this breakdown we use an agent implemented in both languages. This agent is situated in a simple environment consisting of a graph. The agent has primitive actions that allow it to move between connected nodes. Its purpose is to learn the graph.

The agent’s state A state represents everything that the agent knows about its environment. A POSH agent maintains a state by means of various Java modules for processing 'senses' provided by Pogamut. A GOAL agent maintains a state by processing percepts received from an environment by means of an environment interface and updating its beliefs and goals accordingly.

The plan An agent’s plan consists of condition-action rules for making decisions on what to do next based on the agent’s current mental state. In POSH such rules are called POSH plans. For GOAL the concept of a module, which consists of a set of rules, matches with the POSH plan concept. This could include the program section of any module with the exception of the event module as GOAL does not differentiate between low level and high level actions. The plan has been highlighted green in figure 5.1.

Rule conditions Rules in both POSH and GOAL are condition-action rules. The conditions of such rules in GOAL consist of Prolog queries on the agent’s beliefs and/or goals. POSH on the other does not have the ability to directly query the mental state from a plan but has to do this through 'senses'. POSH senses can be chained like conditions in a goal query. As such conditions can be defined queries in GOAL and POSH senses. These have been highlighted blue in figure 5.1.

Rule actions POSH plans contain primitive and aggregate actions. Through primitive actions POSH has access to a large number of high and low level actions provided by Pogamut. GOAL only has access to actions provided by means of the environment interface. That is, a GOAL program cannot access actions available in Pogamut directly if they are not explicitly made available through the interface. As such an action is either a native Pogamut action, a primitive action, aggregate actions, module actions or an environment action. In figure 5.1 these have been highlighted red.

Perception In GOAL the event module can be used to process percepts and store facts in the mental state. POSH has no such feature and relies on modules provided by Pogamut to do this. As such perception comes from either the event module or a module provided by Pogamut.

5.2 Emohawk Environment

A challenging aspect of designing a comparative study for agent languages concerns the choice of environment with which the agents interact. Toy examples, such as the Blocks
5.2 Emohawk Environment

Comparing Programming in POSH and GOAL

Figure 5.2: General overview of POSH, GOAL and the connection to the Emohawk Environment

World discussed above and in [32], for example, do not call for the use of some of the more advanced features of a language, whereas the use of complex gaming environments such as Unreal Tournament may pose too many challenges for a programmer to solve within the necessarily limited time available [14, 13].

In addition, in our study we want to compare two agent languages which also poses technical challenges as both agent frameworks have to be able to interact with the same environment. This technical aspect is not trivial as differences in how to connect to an environment may pose different challenges again to programmers. In fact, apart from the complexity of the Unreal Tournament gaming environment, the difference in how GOAL and POSH connect to it also rules out the use of this environment as it is made available currently.

To compare POSH and GOAL an environment is needed that both APLs can access in a comparable fashion. Emohawk is a modification for the Unreal Engine that replaces the violence with a civilized environment featuring a small town with several locations of interest and several characters that can display text, emoticons and animate[5]. Like Unreal Tournament, Emohawk is accessible through Pogamut allowing an easy connection to be made with both GOAL and POSH. POSH can connect directly with Pogamut; in order to connect GOAL an environment interface had to be created that provides similar functionality. For a complete overview see figure 5.2.

Pogamut provides functionality to manage bots in the Emohawk environment through various modules. These modules provide a wide range of functionality such as various forms of locomotion, displaying emotions, communication with other bots. These modules also contain information about the world and allow an agent to query it. A combination of these can then be used to make the agent do something. For example an agent can ask the Players module for the nearest visible player and then tell the Navigation module to bring the bot to that player.

An important concern for our study is to make sure that the experimental tasks are the same apart from the different programming languages that needed to be used. Most importantly, the same functionality provided by Pogamut should be available for both POSH
5.3 Experimental Design

This study compares the usability of two agent programming languages and their IDE’s. This experiment was carried out in the context of the Emohawk Environment which finds a comparable use in both POSH and GOAL as described in section 5.2.

Participants were given an agent and two assignments to measure their ability to comprehend and modify an agent. The second assignment was divided into several incremental sub-tasks to measure participants progression. The assignments are further described in Section 5.3.2. To gather feedback participants were given three questionnaires in total. One before the experiment and one after completing each task as described in 5.3.3. During the experiment participants were observed.

Subjects for the study were recruited from students attending Delft University of Technology with experience in GOAL and students from Charles University in Prague having experience with POSH. To compensate for potential differences between participants drafted from either University were divided into four groups. Advanced GOAL program-
mers, GOAL Novices, Advanced POSH programmers and POSH Novices as described in section 5.3.4.

5.3.1 Aim of the Study and Hypotheses

The aim of the study was to establish any differences in the usability of the programming frameworks POSH and GOAL. In particular, we were interested in the learning curve related to both frameworks and we investigated differences between novice programmers that just started learning the languages and more advanced programmers that had received more extensive training and practice.

The first hypothesis we formulated relates to the question how quickly programmers are able to familiarize themselves with either POSH or GOAL and to use the languages to write working code. We speculated that because POSH requires programming in Java, a language familiar to all participants, and GOAL requires programming in Prolog, novices would do better when using POSH rather than GOAL. This seems a reasonable hypothesis given that Prolog is perceived as difficult to learn for novices [24]. Moreover, POSH plans, a key feature of the language, are represented visually which is generally considered to facilitate novice programmers. We thus hypothesized that there would be a difference in the learning curve which would make it easier for POSH programmers to perform well on the tasks of our study.

Hypothesis I POSH novices perform better on all tasks than GOAL novices.

Here, performance is measured in terms of successfully meeting the criteria of the task (meeting the criteria formulated, see below) and the time-to-complete. Time-to-complete is considered to be the most important measure for establishing usability differences between the framework.

As discussed above, both POSH and GOAL provide similar though distinct features for programming an agent at a conceptual level. More importantly, both languages provide the features needed for solving the tasks set in the study. For any user that has been relatively well familiarized with either language, given the quite reasonable complexity of the tasks, it should therefore be possible to perform well on these tasks. We saw no specific differences that suggest advanced programmers would perform very differently on the tasks in our study. Therefore we hypothesized that more experienced users will be able to achieve similar results in more or less the same time. In case we would observe different performance, we would take this as an indicator that there would be a real and significant difference between the languages.

Hypothesis II Advanced POSH and GOAL programmers perform similarly on all tasks.

Finally, given that novice and more advanced programmers participated, we hypothesize that advanced programmers outperform novices. This does not have to be case as advanced programmers may complicate things more than novices but seems a very reasonable assumption to make.

Hypothesis III Advanced programmers outperform novices on all tasks.
5.3.2 Tasks

In task-based evaluations, the relevance of the results depends directly on the relevance of the tasks with respect to the purpose of the approach. Given that we want to compare two agent programming frameworks, the design of relevant tasks is a challenge. In order to be able to perform a reasonable comparison of the frameworks, we need to avoid very basic or trivial tasks but at the same time design tasks that can be performed within a reasonable amount of time. [14] illustrates that it is difficult to strike the right balance and report that in their comparative study of POSH and Java subjects became fatigued during an 8 hour session and could not successfully complete the given task. To avoid such problems, we decided that the duration of the experiment should not take longer than 4 hours. Multiple tasks were designed that aim to measure performance on different skills, including program comprehension and code writing. The design of our experiment is a between-subjects design.

Due to time constraints in general and the inexperience of the novice users, for the coding task, we did not consider it feasible for subjects to create a rather complex agent from scratch. Therefore the choice was made to extend an existing agent, by means of a set of incremental tasks, starting with trivial and ending with complex behavior.

At the start of the experiment, subjects were provided with some time to familiarize themselves with the development environment of either framework and were asked to run a given agent connected to the Emohawk environment. This allowed subjects to try the software they need to use for completing the tasks. At this stage, we could provide additional explanation where needed, address questions, or resolve potential other issues and make sure that very basic issues would not have any impact on the results.

Subjects were provided with an initial agent program that already is able to enact a basic scenario in the Emohawk environment. The behavior of this agent consists of an attempt of the virtual character controlled to work at home for a little while before deciding to get a cup of coffee. The attempt to work and the need for coffee are graphically displayed in the environment by means of available emoticons. After deciding to get coffee, the agent walks to the coffee shop, uses emoticons to order a double espresso, and then returns home to make a renewed attempt to work again. This agent was used in the comprehension task and provided the starting point for the coding tasks. To ensure comparable difficulty of the tasks subjects needed to complete, the agent was written in both POSH and GOAL following a predefined specification. We also made sure that the GOAL agent was able to provide similar functionality as the Pogamut framework provides (on top of which the POSH agent runs) by means of adding several Prolog rules to this agent (knowledge).

Subjects were also provided with the documentation that is distributed with POSH and GOAL. For POSH, this material consists of the Pogamut source code and documentation accessible through the IDE. For GOAL a manual of the environment is provided.

The study has been divided into two parts. The focus of the first part is on program comprehension whereas the second part focuses on code writing.
5.3 Experimental Design

Comparing Programming in POSH and GOAL

Comprehension Task

For the comprehension task, subjects were provided with an agent that controls an entity in the Emohawk environment written in either POSH or GOAL. They were asked to examine the agent code and to answer a number of related questions. These included questions about the behavior that the agent will display, the flow of control of the agent, the structure of the program code, and the interpreter that executes the agent. The comprehension task has two main goals. It allows us to measure how well subjects comprehend programs written in POSH or GOAL, and it allows subjects to familiarize themselves with the program code that is used in the subsequent tasks.

Coding Tasks

In the coding tasks subjects were asked to extend the agent program they examined during the comprehension task in various ways. The tasks were presented in order of increasing complexity and each task builds on results obtained in the previous task. The aim of this incremental set of tasks is to create an agent that implements the following scenario in the Emohawk environment: two people (entities controlled by agents) meet in a park, have a conversation, and thereafter visit the cinema together.

The scenario is similar to the one used in [14] in which a Guide agent had to find a Civilian agent, communicate with the Civilian to make the Civilian follow and then lead the Civilian to its home. To make the time needed to test this scenario more predictable the Civilian and Guide meet up in a predetermined position near their final destination. The background story of the scenario has been altered to make this more plausible.

The five coding tasks asked subjects to write code that:

Task 1  The agent has to walk to location of the meeting in the park. A high level action is provided by the platform that makes the agent walk to a given destination. The location is provided as either a knowledge rule in GOAL or a Java function in POSH. The participants have to write checks to see if the action has finished and has achieved the desired result.

Task 2  Continuing from the meeting location in the park, the agent has to walk to cinema and from there on continue with its existing routine. The challenges are identical to the previous task however now the subjects have to make the agent switch between these tasks.

Task 3  A second agent executing the same program is introduced. Both agents have to wait for each other in the park before continuing individually to the cinema. This requires that they are both standing in their assigned spot and are facing each other. Both have to do so long enough to allow the other agent to take notice. Knowledge rules and functions have been provided to test if an agent is standing in a location and is able to see an other location. As such the challenge for the subjects lies in correctly composing the success condition for the wait task.
Task 4  After waiting in the park but prior to walking to the cinema each agent has to simulate a part of a conversation. This is done by displaying 3 sets of 3 emoticons in a row using a provided action. It is not required that the agents actually react to each other, but the sets displayed are different for each agent. The challenge lies in correctly creating a branch in the behavior of the agent.

Task 5  Rather then making the agents walk from the park to the cinema individually, they now have to walk along side of each other. A walk-along action has been provided but the action is prone to failure. As such the challenge is to correctly identify failure of the action and either restart the action or walk to the cinema individually.

The challenges can be summarized as implementing a simple task, switching between tasks, creating a task with a complex success condition, creating a branching task and handling action failure.

We measured the time it took each subject to answer each question and to complete each task. Subjects were asked to write down the time they finished a question or task. This provided us with a measure of progress for all subjects throughout the experiment. To check whether subjects successfully completed a task we ran the agent, observed its behavior, and checked whether this behavior met the criteria described in the assignment.

5.3.3 Questionnaire

Subjects were asked to complete three questionnaires to collect qualitative data. One questionnaire was given before the actual experiment, one questionnaire was given in between the two main parts of the experiment, and one questionnaire was given after completing the coding tasks (or when time ran out). Each questionnaire took about 15 minutes.

Experience Questionnaire

The first questionnaire we designed to obtain information about the subject (e.g., age), about prior programming experience in relevant languages, and about their experience with popular IDEs (e.g., Eclipse). Subjects were asked to rate their own skill in programming with Java, Prolog, POSH, and GOAL. Java is required for programming a POSH agent but also gives a subjective rating of programming experience in a common and popular language that is used extensively in education in both Delft as well as Prague (all subjects were recruited from students taken part in curricula in Delft and Prague). Experience with Prolog is needed to program a GOAL agent.

The results from this questionnaire also provided us with a means to check whether the different groups that got different treatments (based on the various experimental conditions) were balanced in the following sense. Groups should have more or less the same experience in programming, should be familiar with at least one popular IDE, and age should be similarly distributed over groups.
Post Comprehension Questionnaire

The second questionnaire was given to subjects after they finished the comprehension task. This questionnaire asked subjects to rate on a five point likert scale (strongly disagree to strongly agree) how useful they found selected language features for understanding the agent program provided to them. Questions were posed in the form “was feature X helpful”. Subjects could also select a “not applicable” option for each feature. Additionally, they were asked to explain their rating in words.

Post Coding Questionnaire

After completing the coding tasks, a final questionnaire was handed out to subjects. This questionnaire asked subjects to rate the usefulness of a language feature for completing the coding tasks. For each feature they were asked to rate their agreement with the question “was feature X helpful” on a five point likert scale, with 1 being strong disagreement and 5 being strong agreement. For each question there also was the option to select “not applicable”. Additionally, subjects were asked to explain their rating.

5.3.4 Treatments and Subjects

As discussed, we are interested in comparing the usability of two agent programming languages and the learning curve associated with these languages. To this end, we designed four different treatments. These treatments were obtained by combining the conditions of using either POSH or GOAL and the level of experience with either of these languages. This gives four treatments that we labeled as follows:

- POSH novice,
- GOAL novice,
- advanced POSH programmer, and
- advanced GOAL programmer.

Both ‘novice’ and ‘advanced’ needs to be taken quite literally here in the sense that subjects that we labeled ‘novice’ had little to no experience with the languages whereas ‘advanced programmers’ had received a more extensive training. Subjects assigned to the novice condition were given an introductory tutorial on either POSH or GOAL of about two hours before the experiment took place. The tutorial included a basic lecture on the programming language features required by the experiment and a practical tutorial on how to use the IDE and programming language.

Subjects for the study were recruited from students attending Delft University of Technology with experience in GOAL and students from Charles University in Prague having experience with POSH.1 None of the Delft students were familiar with POSH and only

1All of the Delft students participated in the Multi-Agent Systems course in which GOAL is taught, see http://ii.tudelft.nl/trac/goal/wiki/Education. All of the Prague students participated in the course Artificial Beings [8] in which POSH is taught.
Comparing Programming in POSH and GOAL

5.3 Experimental Design

<table>
<thead>
<tr>
<th></th>
<th>GOAL Novice</th>
<th>GOAL Expert</th>
<th>POSH Novice</th>
<th>POSH Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std.</td>
<td>Mean</td>
<td>Std.</td>
</tr>
<tr>
<td>GOAL Usage (Months)</td>
<td>0.2</td>
<td>0.3</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td>GOAL Skill</td>
<td>1.5</td>
<td>0.6</td>
<td>2.8</td>
<td>1.1</td>
</tr>
<tr>
<td>Prolog Usage (Months)</td>
<td>1.8</td>
<td>2.2</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td>Prolog Skill</td>
<td>2.2</td>
<td>1.2</td>
<td>3.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.1: Self-reports on experience and skill

very limited experience with GOAL was reported by the Prague students. Therefore, we could assign Prague students to the GOAL novice treatment and Delft students to the POSH novice treatment. GOAL novices should at least have some experience with Prolog whereas POSH novices should at least have some experience with Java. Results shown in Table 5.1, obtained from the first questionnaire where subjects were asked to report on their own programming experience in terms of man months a language had been used, demonstrate that this was indeed the case. From self-reports on the use of either language and the skill rating we also can conclude that the subjects assigned to the expert treatments indeed were more advanced than the other students (e.g., skill is rated significantly higher; $p_{GOAL} < 0.005$, $p_{POSH} < 0.005$).

28 students from Charles University participated. For 18 of those, participating in the experiment was part of their final exam and students were graded on the performance of their agent. These students were assigned to the advanced POSH programmers condition. The other 10 students voluntarily signed up for the experiment and were assigned to the GOAL novice condition. 11 students from Delft University of Technology participated, all of which volunteered. These were divided equally over the POSH novice (6 students) and advanced GOAL programmers (5 students) condition, stratified by the results of a Java skill test. The Delft students were given an incentive with an equivalent value of 20 Euros.

The total of 39 students that participated included 4 women and 25 men. Table 5.2 describes the distribution of the participants age. Analysis of age in table shows there is a clear difference in age distribution (one way anova, $p < 0.001 ; F = 15.01$) between groups assigned to different conditions. This was the case because, contrary to the expectations, a large number of senior students enrolled in the course in Prague from which subjects were recruited.

Finally, table 5.4 covers experience with IDEs. Analysis shows that almost all students except for a few students assigned to the GOAL expert treatment had experience with either the Eclipse, Netbeans or Visual Studio IDE.
5.4 Results

In this Section, we report on the use of language features by and the performance of subjects in the experiment. For both language usage as well as performance (time-to-complete) self-reports of subjects were used. The data obtained from Likert scale questions is presented by the associated mean and standard deviation which indicates the general tendency of a group assigned to one of the four treatments. Comparisons between treatments are analyzed using chi-square or Fishers-Exact test whenever the categorical minimum was not reached.

5.4.1 Comprehension Task

The comprehension task provided little difficulty for any of the subjects. Figure 5.4 shows that all participants finished this task in roughly the same time (one way anova, \( p = 0.707; F = 0.46 \)). Due to the low number of subjects in the POSH novice and advanced GOAL groups, we report the average and individual completion times. Most questions were answered cor-

<table>
<thead>
<tr>
<th>Table 5.2: Breakdown of the participants by age.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
</tr>
<tr>
<td>GOAL</td>
</tr>
<tr>
<td>Average Age (std.)</td>
</tr>
<tr>
<td>POSH</td>
</tr>
<tr>
<td>Average Age (std.)</td>
</tr>
<tr>
<td>Over All</td>
</tr>
<tr>
<td>Average Age (std.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.3: Breakdown of the participants by experience.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
</tr>
<tr>
<td>GOAL</td>
</tr>
<tr>
<td>Average years enrolled (std.)</td>
</tr>
<tr>
<td>Average years programming (std.)</td>
</tr>
<tr>
<td>POSH</td>
</tr>
<tr>
<td>Average years enrolled (std.)</td>
</tr>
<tr>
<td>Average years programming (std.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.4: Self reported experience with various major IDE’s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
</tr>
<tr>
<td>GOAL</td>
</tr>
<tr>
<td>Eclipse</td>
</tr>
<tr>
<td>Netbeans</td>
</tr>
<tr>
<td>Visual Studio</td>
</tr>
<tr>
<td>Best of three</td>
</tr>
</tbody>
</table>
Comparing Programming in POSH and GOAL

5.4 Results

Figure 5.4: Time-to-complete (mins) for comprehension task

rectly. This is also reflected in the high self-rated satisfaction with the completion of the task (3.9 ± 1.1) for all groups.

To both POSH novices and experts question 4, involving specific behavior of the interpreter, proved to be rather difficult. The difficulty for POSH experts can most likely be attributed to the difference between the mental model and the actual implementation of the interpreter.

5.4.2 Coding Tasks

The coding tasks were considerably more difficult and not all participants were able to complete all tasks in time. The difficulty of these tasks is also reflected in the slightly lower satisfaction ratings (3.4 ± 1.1) for these tasks. In particular, some subjects in the POSH novice and advanced GOAL programmers groups were not able to complete the 4th and 5th task. As we do not have completion times for all subjects, these tasks are not included in the discussion below.
The 4th coding task, simulating a conversation using emoticons, was found to be difficult for quite a number of subjects. Most of these subjects had problems with coding the right duration for which the emoticons should be displayed. The 5th task, where one agent should be coded to follow another, was difficult for all subjects. Most subjects had problems with establishing in their code that the first agent should start following the other agent. Given that all subjects had trouble with completing this task, the task may have been too challenging.

Figure 5.5 shows the cumulative completion times for the first three tasks. Due to the low number of participants in the POSH Novice and GOAL Expert groups the individual completion times are shown for the sake of clarity. Analyzing the cumulative completion times, the advanced POSH and GOAL novice groups from Prague use significantly less time than the groups from Delft ($p < 0.001; F = 9.31$). This difference may be explained by the difference in age, years of enrollment, and years of programming experience, but each of these separately only weakly correlate ($R^2$ of 0.31, 0.26 and 0.17, respectively; see also Figure 5.6 for related scatter plots).
Comparing Programming in POSH and GOAL

5.4 Results

Figure 5.6: Scatter plots - first three coding tasks
5.4 Results

Comparing Programming in POSH and GOAL

Table 5.5: Use of POSH features in the comprehension tasks

<table>
<thead>
<tr>
<th>Feature</th>
<th>Novice Usage</th>
<th>Rating (Std.)</th>
<th>Advanced Usage</th>
<th>Rating (Std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Senses</td>
<td>100%</td>
<td>4.2 (0.4)</td>
<td>94%</td>
<td>4.2 (0.7)</td>
</tr>
<tr>
<td>Actions</td>
<td>100%</td>
<td>4.0 (0.6)</td>
<td>100%</td>
<td>4.2 (0.8)</td>
</tr>
<tr>
<td>Plans</td>
<td>100%</td>
<td>4.8 (0.4)</td>
<td>100%</td>
<td>4.6 (0.8)</td>
</tr>
<tr>
<td>Textual</td>
<td>100%</td>
<td>3.8 (1.0)</td>
<td>94%</td>
<td>1.8 (0.8)</td>
</tr>
<tr>
<td>Java</td>
<td>100%</td>
<td>3.0 (1.2)</td>
<td>83%</td>
<td>2.4 (1.2)</td>
</tr>
<tr>
<td>Overall</td>
<td>4.0 (0.0)</td>
<td>4.0 (0.7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Use of POSH features in the coding tasks

<table>
<thead>
<tr>
<th>Feature</th>
<th>Novice Usage</th>
<th>Rating (Std.)</th>
<th>Advanced Usage</th>
<th>Rating (Std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Senses</td>
<td>100%</td>
<td>3.8 (0.4)</td>
<td>100%</td>
<td>3.9 (0.9)</td>
</tr>
<tr>
<td>Actions</td>
<td>100%</td>
<td>3.6 (0.5)</td>
<td>100%</td>
<td>3.9 (0.9)</td>
</tr>
<tr>
<td>Plans</td>
<td>100%</td>
<td>3.8 (0.4)</td>
<td>100%</td>
<td>3.8 (1.1)</td>
</tr>
<tr>
<td>Textual</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>100%</td>
<td>4.4 (0.5)</td>
<td>100%</td>
<td>3.9 (1.3)</td>
</tr>
<tr>
<td>Overall</td>
<td>3.8 (1.1)</td>
<td>3.7 (0.8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Looking more closely at the scatter plots in Figure 5.6 shows that there are clear outliers for each of the three plots including in total 5 subjects part of the POSH novice and advanced GOAL programmers groups (subjects from Delft).

While there is a difference in performance between the groups from Delft and Prague, further analysis with Tukeys HSD shows there is little difference within these groups ($p_{Delft} = 0.77; p_{Prague} = 0.99$). The programming language thus does not appear to influence the completion times for the subjects from Prague. A similar observation applies to the subjects from Delft but due to the low number of subjects no firm conclusion can be drawn here.

5.4.3 POSH Language Feature Usage

Tables 5.5 and 5.5 show the average ratings given by subjects for how useful they found a POSH feature (see Section 5.1) for the comprehension and coding tasks. Most subjects used almost all POSH language features while completing a task. Both novices and advanced programmers rated features of the language close to a rating of 4.0 indicating that all features were found to be useful. There is one exception. The textual representation of the POSH plans was not found to be useful by advanced programmers but novices did report they found it useful. Interestingly, all features were used in the coding tasks and were rated equally useful by both novices and advanced programmers. It seems to be the case though that novices rely more on the Java code when they are asked to program more complex agent behavior.
Comparing Programming in POSH and GOAL

5.5 Observations

<table>
<thead>
<tr>
<th></th>
<th>Novice</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Usage</td>
<td>Rating</td>
</tr>
<tr>
<td>Specs</td>
<td>90%</td>
<td>2.4 (1.2)</td>
</tr>
<tr>
<td>Modules</td>
<td>90%</td>
<td>4.9 (0.3)</td>
</tr>
<tr>
<td>event</td>
<td>70%</td>
<td>2.3 (1.1)</td>
</tr>
<tr>
<td>Knowledge</td>
<td>100%</td>
<td>2.8 (1.1)</td>
</tr>
<tr>
<td>Prolog</td>
<td>100%</td>
<td>4.3 (0.7)</td>
</tr>
<tr>
<td>Overall</td>
<td>4.6 (0.5)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Use of GOAL features in the comprehension tasks

<table>
<thead>
<tr>
<th></th>
<th>Novice</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Usage</td>
<td>Rating</td>
</tr>
<tr>
<td>Specs</td>
<td>22%</td>
<td>4.5 (0.7)</td>
</tr>
<tr>
<td>Modules</td>
<td>89%</td>
<td>4.6 (0.5)</td>
</tr>
<tr>
<td>Event</td>
<td>22%</td>
<td>4.0 (0.0)</td>
</tr>
<tr>
<td>Knowledge</td>
<td>67%</td>
<td>4.2 (1.2)</td>
</tr>
<tr>
<td>Prolog</td>
<td>100%</td>
<td>4.6 (0.7)</td>
</tr>
<tr>
<td>Overall</td>
<td>4.4 (0.7)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Use of GOAL features in the coding tasks

5.4.4 Language Features GOAL

Tables 5.7 and 5.8 show the ratings given for the usefulness of GOAL features (see Section 5.1) for the tasks. There are clear differences on how the usefulness of GOAL language features is reported compared to those for POSH. Whereas all main GOAL language features were used by completing the comprehension task, this is no longer the case for the coding tasks for both novices and advanced programmers. Generally speaking advanced programmers use more GOAL language features except maybe for Prolog, the language on top of which GOAL runs. It is not quite clear how to explain these findings but they might suggest that there may be a difference in learning curve between POSH and GOAL. Recall that all subjects that used GOAL had initial experience with Prolog, and as in the case of POSH, novices may more quickly resort to what they already know than more advanced programmers. However, due to the low number of subjects as well as the presence of outliers it is difficult to draw any conclusion and it may even be the case that not all language features were required for completing the tasks.

5.5 Observations

During the development of the environment used in the experiment by us and during the experiment several observations were made on the usability of POSH and GOAL. Additionally, students provided feedback on POSH and GOAL on their questionnaires.
5.5 Observations Comparing Programming in POSH and GOAL

5.5.1 GOAL

Exceptions and warnings are hidden  GOAL has been designed such that it’s actions are passed via an environment interface. Underneath this interface it is possible for actions executed by the agent to fail due to a programmer error. For example, when provided with bad parameters for the action, the action will not be executed. Currently GOAL prints the details of the exception to a console and continues the program. After the exception has occurred a fixed number of times, it will no longer be printed to the console. While this creates the impression of robustness, it was found to be very confusing by some subjects. This is an inherently bad practice (see #10 in [24]).

- Observed subjects trying to find a problem that was caused by an exception or warning. The GOAL program would continue to run but the bot did not show the expected behavior. The subjects assumed they had made a mistake in their program and were unable to solve the problem until it was pointed out that GOAL was throwing an exception.

- Observed a subject realizing there was an exception but being unable to find it in the logs. The subject was flabbergasted that the exception could not be found. The exception could not be found because it had occurred several times after which it would no longer be shown. As a result the exception was then pushed off the screen by other log messages from the still running agent.

This issue can be solved by clarifying when a GOAL program has failed. Exceptions thrown by action or percepts should be considered failures of the program and should be clearly communicated to the user.

Issues with environment initializations are ignored  When an MAS is launched, it starts an environment for the agent to live in. This environment may try to configure itself using the arguments supplied to it by the user. When these arguments are incorrect the environment will throw an exception to inform the user that the arguments were not correct. GOAL will however ignore any such exceptions and launch the MAS regardlessly. This was found to be confusing by some subjects. This is an inherently bad practice (see #10 in [24]).

- Observed subjects searching for agents in their environment. According to GOAL The MAS had launched and the environment was running. Yet agents were nowhere to be found as due to incorrect initialization.

- Observed subjects expressing confusion about the message that GOAL had ignored issues with the environment initialization.

This issue can be resolved by properly acknowledging an exception thrown by the environment during initialization and terminating the startup off the MAS.
Comparing Programming in POSH and GOAL 5.5 Observations

**A GOAL program have a lack of locality**  Locality is a measure of the proximity of related items, e.g. the declaration and use sites of a variable specifically. A GOAL program is divided into several sections and references to each of these sections are used in others. This results in a large amount of scrolling and searching for most subjects.

- Observed subjects mashing the search button to find what they needed to know.
- Observed subjects searching by carefully reading line by line to find what they needed to know.
- Observed subjects searching for a predicate they needed, trying the first one that seemed applicable but overlooking any of the more suitable alternatives.
- Subjects inquired after Eclipse style "find all usages" and "goto declaration features". Subjects also commented that they "[...] could use go-to-declaration or find-module IDE action".

As suggested both by subjects and [24] this issue can be alleviated by using modern environments that use powerful navigation features. Another solution might be to reorganize GOAL such that related elements are used close together.

**GOAL IDE is not responsive** The GOAL interface directly uses the events from the AWT-Event dispatcher to execute various actions, some of which are time consuming such as launching or stopping a multi agent system. This means that while the MAS is launching the GOAL interface is not responsive. This also means that any information provided by MAS during its launch will be provided all at once after the MAS has been launched. This resulted in confusion and frustration by some subjects.

- Observed several subjects repeatedly clicking the play button to launch the environment.
- Observed subjects expressing frustration with not knowing what GOAL was doing, being unsure if it had just crashed or was doing something.

This issue can be resolved by improving how GOAL handles long running tasks. For example using worker threads to handle long running tasks [1] and improving the interface to show what GOAL is currently working on.

**Improve debugging** GOAL offers a debugger that outputs the individual steps the interpreter makes through the program.

- Subjects asked if it was possible to step through the program line by line and did not recognize the debugger output on the console as an actual debugger. When this was pointed out the student requested if it was possible to step through the code line by line.

This does indeed make sense as it is a feature supported by all major IDEs.
5.5 Observations

**It is not clear which MAS is started**  The GOAL IDE allows multiple MAS and their agents to be viewed in a tree like structure. This is useful to distinguish which agent belongs to which MAS and quickly switch between different environments. When the environment is started the IDE will start the MAS that was selected last. However after starting a MAS there is no indication which MAS has actually been started.

- Observed during the GOAL introduction. When there are agents from multiple MAS available such as towerworld.goal from towerworld.mas2g and towerworld.goal from DALTowerworld.goal it is not clear which agent is the currently active agent.

This issue can be solved by providing visualize clues to show which MAS is running.

**Ctrl-B for code completion is unusual**  A typical function for an IDE is to predict what users will type and offer this as a suggestion to the user. The user can then complete what he is typing without actually typing it. The GOAL IDE offers a limited form which allows words to be completed.

Users will typically expect that well known features are in a well known place. Common IDEs such as Eclipse, Netbeans and Visual Studio all use CTRL-Space. GOAL uses CTRL-B for this, making it hard to find.

- Comment from a subject. The standard for code completion is CTRL-Space. He also noted that word completion is not code completion.

This issue can be solved by using the defacto standard of CTRL-Space.

**Several issues with syntax colouring**  Another typical function of IDE’s is to provide syntax highlighting. GOAL provides syntax highlighting by parsing the text using regular expressions. Regular expressions are not powerful enough to separate between all cases which may lead to instance where correct syntax is colored wrongly. This negatively affect the IDE as users become confused about the correctness of the syntax.

- Students commented on the fact that when e, pi and other constants occurred at the start of another word they were colored differently.

This issue can be solved by using a modern IDE.

**Typo prevention**  GOAL programs are prone to typos which can result in difficult to spot bugs.

- Subjects commented that they would like to see all similar words light up when he selects a word.

- Observed subject writing a function to query ID from a person percept using the name of that person `getPersonID(Name, ID) :- person(ID, Name, \_\_\_\_\_\_\_\_\_\_)`. The arguments were reversed and so when used this resulted in a bug that was hard to find without knowing how to debug a GOAL program.
Auto completion would help here. It can be used to fill out predicates in the correct order and by suggest existing variables. This would replace typing out the entire predicate by selection from a list of options, reducing the space of possible mistakes.

**Difference between edit and debug mode are unclear** The GOAL IDE provides two modes. A debug mode and an edit mode. This leads to various forms of confusion.

- Observed a subject being unable to edit his program. Subject had switched from the debug mode to the edit mode and promptly forgot that the environment was still running. When in edit mode the interface provided no clues that the environment was still running.

- A subject requested to see the source code while debugging, he could not easily locate the edit button.

A remedy here could be showing the code with the running environment removing the need to return to the edit mode.

**POSH**

**Provide visual queues when using drag and drop** The POSH editor provides a drag and drop interface for its plans. It does however not provide any hints on where a plan element might be dropped. This led to some searching and confusion.

- Observed subjects dragging a plan element around the plan. As if they were looking for a location to make it drop.

A good way to improve POSH editor would be highlighting the available spots. This would not only make dropping a plan element easier, it would also provide a visual cue to the available options.

**POSH Debugger** The POSH interpreter does not come with a debugger. It does provide a log of all actions it takes. However this log is hard to read. It makes it difficult to relate what the interpreter is doing with the actions the agent executes in the environment. It was easier to simply observe the behavior of the agent and reason from that where the plan had gone wrong.

- Observed that subject did not read the interpreter log when debugging but rather observed the agents behavior to conclude what was going wrong.

To remedy this a visual trace of the current path from the drive collection to the current plan element would visually show in what part of the plan the interpreter is evaluating.


5.6 Discussion

Comparing Programming in POSH and GOAL

Names become too long  POSH Actions and Senses all have a unique name. When having multiple actions and senses that do similar things their names become longer. e.g. going-ToLocationAWithFuzzyB, goingToLocationAWithFuzzyC, goingToLocationDWithFuzzyE, etc. These make graphical plans harder to read.

- Observed several solutions with extremely long names.

POSH plans do not allow parameters to be passed with actions. Implementing this would allow more generic methods to be written.

POSH plans are inert  POSH plans can only be build by adding one plan element at a time. It is not possible to move or copy subbranches of the plan into other parts of the plan. Additionally the graphical representation of POSH plans increase in size rather quickly. These are known problems with most graphical representations. The textual view of POSH uses a LISP based syntax featuring superfluous parenthesis.

- Subjects commented on the size of their plans.
- Observed subjects switching to the textual view to copy a part of the plan they wanted to reuse.
- Subjects commented on the difficulty of using the textual version of POSH.

These issues could be resolved by using a textual version of POSH with a simpler syntax.

5.6 Discussion

Somewhat surprisingly the advanced POSH programmers and GOAL novices (subjects from Prague) completed tasks much faster than advanced GOAL programmers or POSH novices. This means that all three hypotheses are rejected. In our study, GOAL novices performed better than POSH novices (and not vice versa), advanced POSH programmers performed better than advanced GOAL programmers (and not similarly), and advanced GOAL programmers did not clearly outperform GOAL novices.

Our findings, however, seem to suggest an alternative interesting conclusion because the better performance by subjects from Prague corresponds with the fact that these subjects were more experienced. Although more work is needed, generally speaking this suggests that having more programming experience improves performance more than additional training in any particular programming language. One might speculate whether more experienced programmers have developed a mental model that is more generally applicable to arbitrary programming language. However, a simpler explanation may be that they are more experienced in using tools for developing programs. We observed that the more advanced programmers made more use of various IDE features. Better handling of the tooling provided with a language may be quite beneficial for programming in general, which would point towards the importance of tool support. As development of most agent
programming languages has focused more on the language features than on the tooling to make these languages accessible to programmers, a shift in focus may be called for.

Given these results and the fact that differences between GOAL novices and advanced POSH programmers and between advanced GOAL programmers and POSH novices are small, we can conclude that it remains difficult to establish relations between language features and programming performance. In this as well as previous studies that compared POSH with Java[13, 14] no significant effects of the programming language on the programmer’s performance was found.

As discussed above, it is and remains difficult to design experimental tasks for comparative studies of (agent) programming languages. One issue of the experimental design we used is the necessarily short amount of time that is available for completing tasks. Advantages of programming languages that might not show up in a short study may still manifest themselves in much larger programming projects which take weeks or even months. Other issues may be more specific to the particulars of agent programming frameworks. Tasks used in this and the studies cited above used the same Emohawk environment. It may be that successfully completing tasks that involve virtual environments such as Emohawk require that more time is spend on observing and testing the agent’s behavior in that environment (if only because virtual characters take more time to perform actions). A significant amount of time thus may be used to this end compared to the time that is actually spend on programming. As a consequence, the use of a virtual environment may make it more difficult to measure the effect of a programming language on completing a task.

That time spend testing is included in the measurement of performance is not a threat to the validity of the comparison. The typical development pattern for both POSH and GOAL is a cycle of writing code, executing the code and observing the results. Given that agents in both languages are build on top of the same framework and given that the subjects are working on the same assignments, the execution and observation phases of the cycle can be expected to be comparable. Differences here can be attributed to a subjects personal skill and/or the language.

Some language specific conclusions may still be drawn based on the language feature usage that we observed. An interesting finding has been that subjects that programmed in POSH use the plan, senses and actions language features for completing the coding tasks. Subjects that programmed in GOAL, however, can make due with just the modules (consisting of rules) and the knowledge section. These language features offer more or less the same expressivity and therefore allow programmers to provide the same functionality by only using these features. This suggests that for this particular study an agent programming language that provides an efficient way to encode and execute plans that consist of condition-action rules may be sufficient. Moreover, a likely reason that explains why the event module of GOAL was not used is that completing the tasks only required writing of a relatively simple reactive agent that can derive its behavior directly from the information it perceives. It may therefore be interesting in future work to design tasks that require more complicated decision logic and/or persistent memory in order to determine when specific language features are actually used. Such tasks might reveal differences between e.g., POSH and GOAL, as tasks in which an agent is required to infer persistent beliefs from percepts might be harder to write in POSH than in GOAL.
Chapter 6

Recommendations, Conclusions and Future Work

In this chapter we answer the research question defined in chapter 1: For a set of typical programming tasks, which features of rule-based agent programming languages facilitate novice and advanced programmers in writing good agent programs? In section 6.1 we discuss the recommendations based on an evaluation of teams in the Multi-Agent Programming Contest. In section 6.2 we conclude the comparison of POSH and GOAL.

6.1 Recommendations

Looking back at the evaluation of the agents in the multi-agent contest in chapter 4 it is possible to derive several recommendations.

By inspecting and categorizing the code based on its functionality we found that teams used a wide variety of solutions. The core functionality action selection and knowledge derivation take up the majority of the code. A common strategy to organize this code was to create modules of code that would execute a high level task. Action selection was generally done by writing rules in a simple linear order. This suggest the primary problem of an agent comes down to writing code for action selection and knowledge derivation in a clear fashion. Of note was also the density of comments around code involving action selection. Participants clearly valued the information provided by the secondary notation. As such the platform/language should:

- Use metaphors based on modules and simple prioritization.
- Provide a clean, brief and esthetically pleasing syntax for the above.
- Provide a secondary notation to clarify actions.

The methodologies used were either agile, agent based or none at all. Agent based methodologies were not very successful but not clearly so. Separating all methodologies...
into either iterative and waterfall showed that iterative teams and teams without a methodology performed better. This suggests that it is is important to use a methodology that supports small increments and regular testing.

This conclusion is primarily based on the evaluation of teams in a competitive setting. Though we believe it generalizes to other settings of agent programming as well. The waterfall model relies on creating thorough, correct and clear specifications before development. This is often impractical as there are many details that can only be discovered once implementation is started; and even if all details were known people can only master a limited amount of detail; and even if it could be mastered, the requirements may change and invalidated prior decisions\[25, 22]. Iterative methodologies account for these limitations and therefore will work better.

The platform and language used should support iterative methodologies and thus:

- Avoid premature commitment to a solution.
- Allow for progressive evaluation.
- Have a low viscosity.

We did not conduct a full evaluation of these properties on agent languages. However the observations made in section 5.5 show that both POSH and GOAL programs can be viscous, they can be resistant to change. Specifically we observed that GOAL programs have a lack of locality. A lack of locality may increase the viscosity of a program when a small change in one place requires a programmer to locate other places that may be affected by the change. The observations on POSH show that it was hard to reuse parts of a POSH plan. This can increase the amount of work that needs to be done to make changes. For both POSH and GOAL these issues can be remedied by improving the development tools of the language.

In some instances the platform/language also limited participants. It was not easy to implement general purpose algorithms. As such:

- Remain flexible enough to implement general purpose code.

6.2 Conclusions

In chapter 4 we introduced a method to categorize the code of an agent program and measure the size of different categories. We applied this measure to the participants of the MAPC. Comparing these results showed several differences between groups. Cross referencing these results with those of a code inspection shows that differences in the size of each category are indicative of different approaches. Extreme differences were also good indications of serious problems within the agent. For example we were able to see HactarV2’s relatively complex communication and Bogtrotters lack of percept processing. Likewise Nargel and Simburg were clear outliers due to their duplicated code. This suggests that categorization is a good way to measure the quality of an agent program. The current process of categorization is however rather labor intensive. We hope to resolve this in future work.
In chapter 5 we reported on a comparative study of the agent programming languages POSH and GOAL. We discussed the design of an experiment that required subjects to complete both comprehension as well as coding tasks for an agent that controls a character in the virtual environment Emohawk. The tasks were designed with the recommendations from section 6.1 in mind. The tasks were setup such that subjects would implement functionality in several iterations and primarily required participants to work on knowledge derivation and action selection. The tasks were also designed so that they could be completed using either POSH or GOAL.

As in other related studies we did not yet obtain conclusive results that suggest the agent programming language used has a significant influence on a programmer’s performance. We did find that general programming experience seems to be a relatively big advantage when using a (new) programming language. This may in part be due to the experience of using tooling support to develop code, which highlights once more the importance of this aspect when developing agent programming languages.

6.3 Future work

This thesis has several possible areas for improvement and future work. We discuss future comparisons and improvements to the Emohawk Environment. IDE Development, Code categorization, and the MAPC are also discussed.

6.3.1 Future Comparisons

The experiment did not yield any differences in the performance of participants between POSH and GOAL. The current tasks required programming a purely reactive agent. This left updating the mental state (processing percepts and setting goals) under utilized. As such the resolving power of the task was low, it could not tell either language apart. We could improve this by creating tasks that deliberately attempt to exploit differences in either language. A task that requires percept processing could exploit the difference between the implicit percept processing used by POSH and explicit processing by GOAL. Likewise a task that requires complicated mental state queries could exploit the perceived complexity of Prolog against Java.

A scenario in which this can be tested could be this. An agent in the Emohawk environment has a gizmo delivery service. Today he has to deliver exactly one gizmo to every customer in the environment. This might sound like a trivial job were it not for the fact that gizmos appear at random places around town during the day. Our agent can only carry one gizmo at the time. Our customers are also rather picky about the gizmo they want, a complication that requires interviewing them first.

The need to interview customers and provide them with a specific gizmo provides multiple dynamic goals. The random appearance of gizmos in the environment requires explicit percept processing. The combination of both also means the agent is working with incomplete information and may make blind commitment non-optimal.

Another avenue of improvement would lie in increasing the relevance of the tasks by more accurately specifying the domain they are drawn from. The current scenario was
6.3 Future work

Recommendations, Conclusions and Future Work

essentially an escort mission, based on prior research[14]. This mission was made with gaming in mind. To increase the relevance with regards to gaming we could look at common scenarios in games and serious gaming. Using these scenarios it might be possible to more accurately describe what is actually needed from a language.

As a final suggestion it might be possible to increase the relevance of the tasks by increasing their complexity. The current tasks, while not trivial are relatively simple for the Unreal Tournament domain. As part of their curriculum students in Delft and Prague create agents teams that are far more complicated. It would certainly be interesting to compare these teams. We’ve shown that it is possible to setup a comparable environment for the Emohawk environment and as such we can expect to do the same with Unreal Tournament 2004.

However the non technical differences might proof to be a bigger problem for this approach. The differences in the curriculum, differences in the nature of the participants, differences in project setup could render such a comparison irrelevant fairly quickly. Due to external constraints and limitations these are also hard to equalize as this thesis already experienced. Mitigating these by splitting the subjects into two groups is also not feasible as POSH and GOAL are part of a fixed curriculum. That the students work in groups to develop their agents adds another complicating factor. It makes it impossible to correlate the performance of an agent to a single statistic.

6.3.2 Emohawk Environment

The Emohawk environment can be used by both POSH and GOAL. The implementation however does not include all functionality provided by the Emohawk modification. The environment could be expanded to include these features and implement the scenario mentioned in the previous section.

The purpose of the Emohawk environment was to provide both languages with identical functionality in a simple environment. This is however a fairly niche functionality. It would be unexpected for the environment to see major use. As an alternative it might be useful to provide identical functionality for the Unreal environment. The Unreal environment is more actively used. This translates into a larger pool of material for future comparisons. Though the objections mentioned in the previous section would first have to be solved.

6.3.3 IDE Development

During the experiment we have received a huge amount of comments and feedback on the implementation of POSH and GOAL. This feedback can be used to improve the working of the IDE. For GOAL the general tendency seems to be that it might be better to use an IDE framework such as Eclipse or NetBeans rather then a custom IDE.

6.3.4 Code categorization

Categorizing code was done manually, this made it a fairly labor intensive process. Automating this would reduce the work load of the process. It is not feasible to automate the categorization in a generic purpose language. This can be done using a language that clearly
marks the functionality of code such as GOAL. Future work could focus on measuring these aspects in GOAL.

6.3.5 Multi-Agent Programming Contest

This year the MAPC used a new scenario and as such all teams created a new implementation. This excludes the influences and problems of working with legacy code. The next competition will re-use the same scenario and it would be interesting to evaluate if and how teams reused their code.

Acknowledgement

This research -specifically the study conducted in chapter 5- is supported by the Czech Science Foundation under the contract P103/10/1287 (GACR), by student grant GA UK No. 655012/2012/A-INF/MFF.
Bibliography


Appendix A

Materials

This appendix covers the materials used during the experiment. This includes the questionnaires, assignments, code and manuals provided to the participants. The Java documentation and source code for Emohawk is available through the Pogamut SVN repository at svn://artemis.ms.mff.cuni.cz/pogamut/trunk/project/Main/PogamutEmohawk.

A.1 Assignments and Questionnaires
PreTest

Name: ________________________________________

Question 1: What is the value of the expression:

\[(2/4)*4 + 4\%2\]

1. 0  
2. 2  
3. 4  
4. 6

Question 2: Which line will print the letter h.

String s = "hello";

1. System.out.print(charAt(s,0));  
2. System.out.print(s.charAt(0));  
3. System.out.print(charAt(s,1));  
4. System.out.print(s.charAt(1));

Question 3: What will be the output of the program?

String x = "xyz";
x.toUpperCase();
String y = x.replace('Y', 'y');
y = y + "abc";
System.out.println(y);

a) abcXyZ  
b) abcxYZ  
c) xyzabc  
d) XyZabc

Question 4: Which statement is true for the class java.util.HashSet<Double>?  
a) The elements in the collection are ordered.  
b) The collection is guaranteed to be immutable.  
c) The elements in the collection are guaranteed to be unique.  
d) The elements in the collection are accessed using a unique key.
Question 5: What will be the output of the program?

```java
public class CommandArgs {
    public static void main(String[] args) {
        String s1 = args[1];
        String s2 = args[2];
        String s3 = args[3];
        String s4 = args[4];
        System.out.print(" args[2] = " + s2);
    }
}
```

the command-line invocation is

```
> java CommandArgs 1 2 3 4
```

a) `args[2] = 2`
b) `args[2] = 3`
c) `args[2] = null`
d) An exception is thrown at runtime.

Question 6: What will be the output of the program?

```java
class Super {
    public String text = "Hello 0";
    public Super(String text) {
        this.text = text +" " + 1;
    }
}
class Sub extends Super {
    public Sub(String text) {
        this.text = text +" " + 2;
    }
    public static void main(String args[]) {
        Sub sub = new Sub("Hello");
        System.out.println(sub.text);
    }
}
```

a) Hello 0
b) Hello 1
c) Hello 2
d) Compilation fails.
Question 7: Which of the following would compile without error?

a) int a = Math.abs(-5);
b) int b = Math.abs(5.0);
c) int c = Math.abs(5.5F);
d) int d = Math.abs(5L);

Question 8: You want subclasses in any package to have access to members of a superclass. Which is the most restrictive access that accomplishes this objective?

a) public
b) private
c) protected
d) transient

Question 9: What will be the output of the program?

```java
public class Question7{
    private int i = j;
    private int j = 10;
    public static void main(String args[]){
        System.out.println((new Question7()).i);
    }
}
```

1. Compiler error complaining about access restriction of private variables of Question7.
2. Compiler error complaining about forward referencing.
3. The output is 0.
4. The output is 10.

Question 10: An Interface can never be private or protected.

1. True
2. False

Question 11: A Class implementing this interface should...

```java
public interface Question{
    public abstract void someMethod() throws Exception;
}
```

1. ...necessarily be an abstract class.
2. ...should have the method public abstract void someMethod();
3. ...should have the method public void someMethod() which has to throw an exception which is a subclass of java.lang.Exception.
4. ...should have the method public void someMethod() which need not throw an Exception.
Question 12: Read the code below. What will be the result of attempting to compile and run the code below.

```java
public class Question {
    public void method(Object o){
        System.out.println("Object Version");
    }
    public void method(String s){
        System.out.println("String Version");
    }
    public static void main(String args[]){
        Question question = new Question();
        question.method(null);
    }
}
```

1. The code does not compile.
2. The code compiles cleanly and shows “Object Version”.
3. The code compiles cleanly and shows “String Version”
4. The code throws an Exception at Runtime.
Question 13: Read the code below. Will be the result of attempting to compile and run the code below.

```java
public class Question{
    public void method(StringBuffer sb){
        System.out.println("StringBuffer Version");
    }
    public void method(String s){
        System.out.println("String Version");
    }
    public static void main(String args[]){
        Question question = new Question();
        question.method(null);
    }
}
```

1. The code does not compile.
2. The code compiles cleanly and shows “StringBuffer Version”.
3. The code compiles cleanly and shows “String Version”
4. The code throws an Exception at Runtime.

Question 14: Which of the following statements are true?

Statement A: A method can throw an Exception
Statement B: A method can return an Exception

1. only A
2. only B
3. both A and B
4. Neither.

Question 15: The range of a byte is from -127 to 128

1. True
2. False

Question 16: Which two statements are true about comparing two instances of the same class, given that the `equals()` and `hashCode()` methods have been properly overridden?

A: If the `equals()` method returns true, the `hashCode()` comparison `==` must return true.
B: If the `equals()` method returns false, the `hashCode()` comparison `!=` must return true.
C: If the `hashCode()` comparison `==` returns true, the `equals()` method must return true.
D: If the `hashCode()` comparison `==` returns true, the `equals()` method might return true.

1. A and B
2. C and D
3. A and D
4. B and C
Question 17: What will be the output of the program?

```java
public class Foo {
    public static void main(String[] args) {
        try {
            return;
        } finally {
            System.out.println( "Finally" );
        }
    }
}
```

1. Finally
2. Compilation fails.
3. The code runs with no output.
4. An exception is thrown at runtime.

Question 18: What will be the output of the program?

```java
int i;
for (i = 0; i < 4; i += 2) {
    System.out.print(i + " ");
}
System.out.println(i);
```

1. 0 2 4
2. 0 1 2 3 4
3. 0 2 4 5
4. Compilation fails.
Question 19: Please implement the `getUniqueElements` method to return a set of elements that can be found only in set a or set b, but not in both Sets. The method should not change the content of the parameters. You may use pseudo code.

```java
import java.util.*;

public class Question {
    public static Set<Object> getUniqueElements(Set<Object> a, Set<Object> b) {
    }
}
```
Question 20: Please implement the reverse method to reverse a given linked list. You may use pseudo code.

```java
public class Question {
    // Please do not change this interface
    interface ListNode {
        int getItem();
        ListNode getNext();
        void setNext(ListNode next);
    }

    public static ListNode reverse(ListNode node) {
    }
}
```
**Startup**

The intention of this assignment is to allow some time to familiarize yourself with the IDE, documentation and experiment setup. Additionally before the experiment proper can be begin some software has to be installed first.

If you run into any problems, please give a shoutout.

**Installation**

1. Login with your netid.
2. Copy J:\ewi\courses\poshgoal to D:\{UserName}\poshgoal

For further reference start all programs using the batch files in D:\{UserName}\poshgoal

3. Start Emohawk Server
4. Start UE Runtime (and wait 10 seconds.)
5. Try moving around. Try spectating.

**POSH**

This parts assumes you've been assigned to the POSH group. If you are not in the POSH group continue to the next section.

1. Start Nebeans
2. If it asks about JDK say yes.
3. Load project D:\{UserName}\poshgoal\Programs\PoshComphrehension
4. Start the agent. (Press run)
5. Wait for download (Only once).
6. Observe agent in the world.

**GOAL**

This part assume you've been assigned to the GOAL group. If you are not in the GOAL group continue to the next section.

1. Start GOAL
2. Open comprehension/comprehension.mas2g
3. Start agent
4. Observer agent in world.

**Documentation**

Documentation can be found in D:\{UserName}\poshgoal\Documentation. Please review the relevant material. For the POSH group additional material is available as JavaDoc through Netbeans.
Questionnaire 1

Basic Information

Name: ____________________________________________

Age: _______ Gender:  M/F (Strike out inapplicable)

Language: GOAL / POSH (Strike out inapplicable)

Experience

1. How many years have you been enrolled at a university?

2. How many years have you been programming regularly.
   
   *E.g. Jack learned his first programming language at age 15 but only started to program regularly after enrolling at university this year. He answers 1.*

3. Which programming languages do you know and how long have you been using these? Please also rate your skill with each from 1 (very poorly) to 5 (very good) or N/A (never used).
   
   *A few examples to consider: Java, Posh, Goal, Prolog, C, C# or C++. 1 man-month = 20 man-days = 160 hours of work.*

<table>
<thead>
<tr>
<th>Language</th>
<th>Man-Months</th>
<th>Skill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prolog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Posh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOAL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Outside of the university courses, do you have any experience programming or modifying virtual agents or artificial intelligence? Please describe.

5. Please rate your how well you are acquainted with Netbeans.

<table>
<thead>
<tr>
<th>Novice</th>
<th>Expert</th>
<th>Never used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6. Please rate how well you are acquainted with Eclipse.

<table>
<thead>
<tr>
<th>Novice</th>
<th>Expert</th>
<th>Never used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

7. Please rate how well you are acquainted with Visual Studio.

<table>
<thead>
<tr>
<th>Novice</th>
<th>Expert</th>
<th>Never used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

8. Are you familiar with other IDE's? Please list and add a 1/5 rating for each.
Comprehension

The intention of this assignment is to measure comprehension of an agent program by answering questions about it. The agent given is a simple agent that will follow a simple plan.

This assignment is divided into several sub assignments. Before starting and after finishing each assignment please write down the time in the table below. Try to complete each assignment to the best of your abilities before moving on to the next part. Should you return to an assignment later on, please also write down the second start time and second end time.

Questions for each assignment either have a GOAL, POSH or POSH & GOAL question. Please answer the version for the language you are using.

Name: ________________________________________

Language: GOAL / POSH (Strike out inapplicable)

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Start time (HH:mm)</th>
<th>End time (HH:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comprehension 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comprehension 1
1. Please record the starting time.
2. POSH: Which drive will be executed, meeting or workCoffeeCycle?
   GOAL: Which module will be executed, meeting or workCoffeeCycle?

3. Please record the time of completion

Comprehension 2
1. Please record the starting time.
2. POSH: If the hadCoffee sense returns true but the agent is not at home, what will the agent do?
   GOAL: If the agent has the goal work, but is not at home. What will the agent do?

3. Please record the time of completion

Comprehension 4
1. Please record the starting time.
2. POSH: How many iterations does it take to complete the orderCoffee action pattern?
   GOAL: How many cycles does it take to complete all the orderCoffee module?

3. Please record the time of completion
Comprehension 5

1. Please record the starting time.
2. POSH: Describe which conditions must be true to make the agent visit the restaurant. Include all necessary senses and their outcome in your description.
   GOAL: Describe which conditions must be true to make the agent visit the restaurant. Include all necessary mental state conditions.

3. Please record the time of completion

Comprehension 6

1. Please record the starting time.
2. GOAL & POSH: If the agent is at the restaurant. What additional information do you need to determine the next action of the agent? For all possible variations of this information. What will the next action be?

3. Please record the time of completion
Comprehension 7

1. Please record the starting time.

2. POSH: Which competence is included in the plan but not actually used?
   GOAL: Which module is included in the agent program but not actually used?

3. Please record the time of completion

Comprehension 8

1. Please record the starting time.

2. POSH: If you wanted to change the agent such that it were to (attempt to) work at at the cinema rather then at home, which plan elements would have to be changed? Please write this down in the format oldLine → newLine.
   GOAL: If you wanted to change the agent such that it were to (attempt to) work at at the cinema rather then at home, which lines would have to be changed? Please write this down in the format oldLine → newLine.

3. Please record the time of completion.
Questionnaire 2

Basic Information

Name: ________________________________________

Language: GOAL / POSH (Strike out inapplicable)

Exercise

1. Did you understand the assignment? If not, please explain.

2. I am satisfied with the answers I have given.

   O ----- O ----- O ----- O ----- O ----- O O
   1 2 3 4 5 N/A

   Strongly disagree Neither agree nor disagree Strongly agree

Please explain:
3. Please rate the ease of use of the language.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Hard Easy

Please explain:

4. Please rate the ease of use of the IDE.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Hard Easy

Please explain:

5. Please rate the ease of use of the environment

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Hard Easy

Please explain:
IDE

1. Auto completion was a useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

2. Syntax highlighting was a useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

3. The error reporting was useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

4. The (code) search feature was useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

5. The output/console was useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

6. The debugger was useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence

7. Breakpoints were useful to me when making the assignment.
   Strongly disagree Neither agree nor disagree Strongly agree Not used Not aware of existence
8. Being able to start / stop / step through the execution of the agent were a useful features.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>Strongly disagree</td>
<td>Neither agree nor disagree</td>
<td>Strongly agree</td>
<td>Not used</td>
<td>Not aware of existence</td>
<td></td>
</tr>
</tbody>
</table>

9. The logging of the program execution was a useful feature.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>Strongly disagree</td>
<td>Neither agree nor disagree</td>
<td>Strongly agree</td>
<td>Not used</td>
<td>Not aware of existence</td>
<td></td>
</tr>
</tbody>
</table>
1. Please rate how well POSH actions aid you with understanding POSH plans.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

- Not at all
- Somewhat
- Very much
- Did not use

Please explain:

2. Please rate how well POSH senses aid you with understanding POSH plans.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

- Not at all
- Somewhat
- Very much
- Did not use

Please explain:
3. Please rate how well the graphical representation of POSH plans aids your understanding.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>Somewhat</td>
<td>Very much</td>
<td>Did not use</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

4. Please rate how well the textual representation of POSH plans aids your understanding.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>Somewhat</td>
<td>Very much</td>
<td>Did not use</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
5. Please rate how well Java aids you in understanding POSH plans.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>-----</td>
<td>O</td>
<td>-----</td>
<td>O</td>
<td>-----</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Not at all | Somewhat | Very much | Did not use

Please explain:

6. Please rate the overall ease of understanding POSH plans.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>-----</td>
<td>O</td>
<td>-----</td>
<td>O</td>
<td>-----</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hard | Easy

Please explain:
**Language: GOAL**

1. Please rate how much Prolog aids you with understanding GOAL agents.

<table>
<thead>
<tr>
<th>Not at all</th>
<th>Somewhat</th>
<th>Very much</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

   Please explain:

2. Please rate how much the knowledge section helps you understanding GOAL agents.

<table>
<thead>
<tr>
<th>Not at all</th>
<th>Somewhat</th>
<th>Very much</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

   Please explain:
3. Please rate how well action specification aids you with understanding GOAL agents.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Somewhat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very much</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

4. Please rate how well the program rules aids you with understanding GOAL agents.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Somewhat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Very much</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
5. Please rate how well modules aid you with understanding GOAL agents.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>Somewhat</td>
<td>Very much</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

6. Please rate how well the event module aids you with understanding GOAL agents.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>Somewhat</td>
<td>Very much</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
7. Please rate how well the whole GOAL program aids you with understanding GOAL agents.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not at all</td>
<td>Somewhat</td>
<td>Very much</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
Documentation

1. Which documentation have you used during the exercise? Think of manuals, javadoc or google. If so, what did you look for? Which questions did you need to address?

Did the documentation help in answering your questions?
Assignment Creation

In the previous assignment you've been examining an agent. In this assignment your task is to extend this agent such that it will first play out a scenario. Please use the meeting module as a starting point for this.

In this scenario Jan and Pier are meeting each other in the park and, engaging in a conversation. After this conversation is finished they visit the cinema. Then they both part ways and resume their original behaviour.

This assignment is divided into several sub assignments. Before starting and after finishing an assignment please write down the time. Use the table below for this. Try to complete each assignment to the best of your abilities before moving on to the next part. Should you return to an assignment later on, please mark the second start time and second end time.

Name: ________________________________________

Language: GOAL / POSH *(Strike out inapplicable)*

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Start time (HH:mm)</th>
<th>End time (HH:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creation 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creation 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creation 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creation 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Creation 1**

The goal of this part is to make the agent move to a specific location.

1. Please record your starting time for this part.

2. Modify the agent such that the agent will walk to the meeting place. The location of the park is provided as a knowledge rule parkMeeting(Location) for GOAL and as a getParkMeeting() function for POSH.

3. Please record your time of completion for this part and back up your solution.

**Creation 2**

The goal of this part is to make the agent move to one specific location and then another.

1. Please record your starting time for this part.

2. Modify the agent such that after walking to the park, the agent will walk to the cinema. The location of the cinema is provided as a knowledge rule cinemaMeeting(Location) for GOAL and as a getCinemaMeeting() function for POSH.

3. Please record your time of completion for this part and back up your solution.

**Creation 3**

The goal of this part is to have two agents who'll walk to the park, wait for one another againt and then walk to the cinema.

1. Please record your starting time for this assignment.

2. Modify the startup code such that two agents can be launched. This can be done by uncommenting the relevant bits in creation.mas2g or creationLogic.java. Start your agents to make sure both launch.

3. Modify your agents such that after walking to the meeting spot in the park, both agents wait for the other to arrive before they continue walking to the cinema. For GOAL the knowledge rule otherAtParkMeeting can be used, for POSH make use of the sense otherAtPark(). Note that for this to work the agent must look at the other. So you must also make sure that the agents both look in the right direction.

5. Please record your time of completion for this assignment and back up your solution.
**Creation 4**

The goal of this assignment is to simulate a conversation between two agents using emoticons while waiting at the park.

1. Please record your starting time for this assignment.

2. Modify your agent such that after both agents have arrived at the park they display three sets of emoticons. After all sets have been displayed they may continue.

The three sets of emoticons must be displayed for their full duration. You can see which emoticons are currently active by either looking at the self percept or the appropriate method on Self object as provided by info.getSelf().

3. Please record your time of completion for this assignment and back up your solution.

The groups are:

JAVA

EmoticonType.

<table>
<thead>
<tr>
<th>Group</th>
<th>Emoticon Type 1</th>
<th>Emoticon Type 2</th>
<th>Emoticon Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>SECOND_GIRL_66</td>
<td>MONSTER_39</td>
<td>EMOHAWK_RIFLE_20</td>
</tr>
<tr>
<td>Group 2</td>
<td>BOY_67</td>
<td>KITTY_43</td>
<td>EMOHAWK_RIFLE_21</td>
</tr>
<tr>
<td>Group 3</td>
<td>FIRST_GIRL_68</td>
<td>HAMSTER_46</td>
<td>ENERGY_DRINK_22</td>
</tr>
<tr>
<td>Group 3</td>
<td>SECOND_GIRL_66</td>
<td>BOY_67</td>
<td>FIRST_GIRL_68</td>
</tr>
<tr>
<td>Group 4</td>
<td>MONSTER_39</td>
<td>KITTY_43</td>
<td>HAMSTER_46</td>
</tr>
<tr>
<td>Group 6</td>
<td>EMOHAWK_RIFLE_20</td>
<td>EMOHAWK_RIFLE_21</td>
<td>ENERGY_DRINK_22</td>
</tr>
</tbody>
</table>

**GOAL**

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Emoticon Type 1</th>
<th>Emoticon Type 2</th>
<th>Emoticon Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1</td>
<td>second_girl_66</td>
<td>monster_39</td>
<td>emohawk_rifle_20</td>
</tr>
<tr>
<td>Group 2</td>
<td>boy_67</td>
<td>kitty_43</td>
<td>emohawk_rifle_21</td>
</tr>
<tr>
<td>Group 3</td>
<td>first_girl_68</td>
<td>hamster_46</td>
<td>energy_drink_22</td>
</tr>
<tr>
<td>Group 4</td>
<td>second_girl_66</td>
<td>boy_67</td>
<td>first_girl_68</td>
</tr>
<tr>
<td>Group 4</td>
<td>monster_39</td>
<td>kitty_43</td>
<td>hamster_46</td>
</tr>
<tr>
<td>Group 6</td>
<td>emohawk_rifle_20</td>
<td>emohawk_rifle_21</td>
<td>energy_drink_22</td>
</tr>
</tbody>
</table>
**Creation 5**

Up until now both agents have been walking independently to the cinema. To improve this we'll make them walk side by side.

1. Please record your starting time for this assignment.

2. Modify your agent such that when Jan is going to the cinema, he will follow Pier. You can use the walkAlong action for this. The walkAlong action only works when Jan was able to see Pier when. Should Jan at some point get stuck or loose sight of pier, he should run to the theatre alone.

3. Please record your time of completion for this assignment and back up your solution.
Questionnaire 3

Basic Information

Name: ________________________________________

Language: GOAL / POSH (Strike out inapplicable)

Exercise

1. Did you understand the assignment? If not, please explain.


2. I am satisfied with the agent I have written.

   O ----- O ----- O ----- O ----- O ----- O ----- O
   1 2 3 4 5 N/A

   Strongly disagree  Neither agree nor disagree  Strongly agree

Please explain:
3. Please rate the ease of use of the language.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:


4. Please rate the ease of use of the IDE.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:


5. Please rate the ease of use of the environment

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:


IDE

1. Auto completion was a useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Syntax highlighting was a useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. The error reporting was useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. The (code) search feature was useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. The output/console was useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. The debugger was useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. Breakpoints were useful to me when making the assignment.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither agree nor disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly agree</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not used</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not aware of existence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8. Being able to start / stop / step through the execution of the agent were a useful features.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A N/A</td>
</tr>
</tbody>
</table>

Strongly disagree  Neither agree nor disagree  Strongly agree  Not used  Not aware of existence

9. The logging of the program execution was a useful feature.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>N/A N/A</td>
</tr>
</tbody>
</table>

Strongly disagree  Neither agree nor disagree  Strongly agree  Not used  Not aware of existence

10. If you could pick 3 things to improve or add to the IDE, which would you improve? Please also explain why.

9. Any other comments about the IDE?
1. Please rate the ease of creating and editing POSHSenses.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>Easy</td>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

2. Please rate the ease of creating and editing POSHActions.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>Easy</td>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
3. Please rate the ease of creating and editing POSH plans.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

---

4. Please rate the ease of using Java for writing actions and sense.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
5. Please rate the overall ease of creating agents with POSH.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hard: Easy: Did not do

Please explain:
Language: GOAL

1. Please rate the ease of creating and editing action specifications.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hard     Easy     Did not do

Please explain:

2. Please rate the ease of creating and editing your own modules.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hard     Easy     Did not do

Please explain:
3. Please rate the ease of creating and editing rules for the event module.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>Easy</td>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:

4. Please rate the ease of creating and editing rules for knowledge section.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>Easy</td>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
5. Please rate the ease of creating and editing rules in the program section.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

Hard          Easy          Did not do

Please explain:

6. Please rate the ease of using Prolog in GOAL programs.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
</tbody>
</table>

Hard          Easy          Did not do

Please explain:
7. Please rate the overall ease of creating and editing an agent with GOAL.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>Easy</td>
<td>Did not do</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please explain:
POSH & GOAL

1. Which features from POSH would you like to see in GOAL?

2. Which features from GOAL would you like to see in POSH?
Documentation

1. Which documentation have you used during the exercise? Think of manuals, javadoc or google.

If so, what did you look for? Which questions did you need to address?

Did the documentation help in answering your questions?
Other
Anything else you want to share? Comments, suggestions, frustrations, anything goes.
A.2 Results

The anonymized data and calculations are available online at:
https://docs.google.com/spreadsheet/ccc?key=0AheILoAs5VI8dHd3M1FYTjJ3NTQyOPFqb0V1eXRURmc.

A.3 Documentation

The documentation describes the environment, how to use it and the available actions and percepts. The full documentation is provided to the subjects using GOAL. The first page is provided to subjects using POSH which had access to the Java Documentation of Pogamut (included through the Netbeans IDE).
Environment Manual

The environment is a simplified version of the Unreal Tournament using the Pogamut Emohawk mod for the Unreal Development Kit. Featuring a small town and several actors which can move about the world, animate and emote.

Organisation

**Drawing 1: Organisation of the Environment. POSH and GOAL talk with Unreal over the internet using Pogamut.**

Operation

The environment is started by launching "Emohawk Server.bat" and can be viewed using "UE Runtime.bat".

When in the environment it is possible to move around using WASD + Mouse. It is also possible to go into spectator mode by pressing escape and selecting spectator mode. When in spectator mode it is possible to follow other players by changing the camera perspective by pressing LMB repeatedly. If it also possible to go to a free moving camera which can be controlled with WASD.
Environment EIS API

Types

ILocated A Location, a NavPointID or a PlayerID

PlayerID The UnrealID of a player

NavPointID The UnrealID of a navpoint

Location A function of the form location(x,y,z). z and y lie in the horizontal plane, z is up

Rotation A function of the form rotation(p,y,r)

Integer A whole number


**Actions**

**Action** stop
**Description** The agent stops moving. The next navigation percept will be navigation(waiting).

**Action** runTo(ILocated destination)
**Action** walkTo(ILocated destination)
**Description** The agent will run or walk to the given destination. The next navigation percept will be navigation(navigating). If the destination is a Player, the agent will try to follow that player. This action can be called repeatedly with the same location but this is not required.

**Action** walkAlong(PlayerID person)
**Description** Walks next to the given person, following him around the map. The next navigation percept will be navigation(following).

**Action** emote(Emoticon left, Emoticon center, Emoticon right)
**Description** Displays the given emoticons. If none is used for any emoticon, it will not be displayed. The emoticon fades out after approximately 5 seconds. While an emoticon is displayed by this agent it will also be reflected in the self percept and person percept.

**Action** turn(Integer degrees)
**Description** Turns the given number of degrees.

**Action** turnTo(ILocated location)
**Description** Turns towards the given location

**Action** jump
**Description** The agent will hop a little in the air.
Action: skip
Description: Does nothing. Intentionally.

**Percept**

- **Percept**: navigation(<NavigationState>)
  - **Type**: Always
  - **Description**: The current state of the navigation.
- **NavigationState**
  - **following**: Indicates the agent is following another player.
  - **stuck**: Indicates the agent got stuck trying to walk or run to a location.
  - **path_unavailable**: Indicates the agent was unable to find a path to the location.
  - **destination_reached**: Indicates the agent reached the location as given by runTo or walkTo.
  - **traveling**: Indicates the agent is underway.
  - **waiting**: Indicates the agent is not traveling anywhere.

*Drawing 2: State diagram presenting the possible state transitions of the navigation.*

- **Percept**: place(<ID>, <Location>, <Radius>)
  - **Type**: Once
  - **Description**: Describes a place in the world as a circle with a position and a radius.
- **Percept**: person(<PlayerID>, <Name>, <Location>, <Rotation>, <Emoticon>, <Emoticon>, <Emoticon>)
  - **Type**: Always
  - **Description**: A person the agent can see. When this percept is not provided, the person cannot be seen.
Percept  self(<PlayerID>,<Name>,<Location>,<Rotation>,<Emoticon>,<Emoticon>,<Emoticon>)
Type  Always
Description  Information about this agent.

Percept  emoticon(<Emoticon>)
Type  Once
Description  The ID of an Emoticon.
A.4 Code

A.4.1 GOAL

The agent provided to the subjects using GOAL. The agent consists of a MAS definition and a GOAL Program containing the agent logic. Both are shown below.

**MAS Definition**

```plaintext
environment {
  "Environment−goal−emohawk−environment−3.3.2−SNAPSHOT.jar".
  init [
    botNames = ["Jan"],
    %botNames = ["Jan", "Pier"],
    botServer = "ut://127.0.0.1:3000",
    logLevel = "OFF"
  ].
}
agentfiles {
  "comprehension.goal" [name=unrealgoalfile] .
}
launchpolicy {
  when [type=bot,max=3]@env do launch bot:unrealgoalfile .
}

**GOAL Program**

%%
% An agent that meets another for a brief visit to the cinema and drinks allot of coffee.
%
% @Author M.P. Korstanje
%%
init module {
  knowledge{
    % The name of this agent.
    myName(Name):− self(_,Name,_,_,_,_,_,_).

    % Current location
    myLocation(Location) :− self(_,_,Location,_,_,_,_,_).

    % Aliases for quick lookup.
```
home(Location, Range) :- place(barbara_home, Location, Range).
restaurant(Location, Range) :- place(restaurant, Location, Range).
cinema(Location, Range) :- place(cinema, Location, Range).
park(Location, Range) :- place(park, Location, Range).
sphere(Location, Range) :- place(sphere, Location, Range).

% The location for the meeting in the park
parkMeetingJan(Location) :- place(park, ParkLocation, Range), locationAdd(ParkLocation, location(100,0,0), Location).
parkMeetingPier(Location) :- place(park, ParkLocation, Range), locationAdd(ParkLocation, location(-100,0,0), Location).

% The location for this agent.
parkMeeting(Location) :- myName('Jan'), parkMeetingJan(Location).
parkMeeting(Location) :- myName('Pier'), parkMeetingPier(Location).

% The location where the other guy will stand
parkMeetingOther(Location) :- myName('Jan'), parkMeetingPier(Location).
parkMeetingOther(Location) :- myName('Pier'), parkMeetingJan(Location).

% The location for the meeting in the cinema.
cinemaMeetingJan(Location) :- place(cinema, CinemaLocation, Range), locationAdd(CinemaLocation, location(100,0,0), Location).
cinemaMeetingPier(Location) :- place(cinema, CinemaLocation, Range), locationAdd(CinemaLocation, location(-100,0,0), Location).

% Location where this agent will stand.
cinemaMeeting(Location) :- myName('Jan'),
cinemaMeetingJan(Location).
cinemaMeeting(Location) :- myName('Pier'),
cinemaMeetingPier(Location).

% Info about the other guy
personAt(Name, Location) :- person(_, Name, PersonLocation, _, _,_), distanceLt(Location, PersonLocation, 80).
otherAtParkMeeting :- myName('Jan'), parkMeetingPier(Location), personAt('Pier', Location).
otherAtParkMeeting :- myName('Pier'), parkMeetingJan(Location), personAt('Jan', Location).

%Adds first two locations, returns result in third
locationAdd(location(X1, Y1, Z1), location(X2, Y2, Z2),
            location(X, Y, Z)) :- X is X1 + X2, Y is Y1 + Y2, Z is Z1 + Z2.

% Measures distance between two locations.
distance(location(X1, Y1, Z1), location(X2, Y2, Z2), Distance)
             :- Distance is sqrt((X1 - X2) ** 2 + (Y1 - Y2) ** 2 + (Z1 - Z2) ** 2).

% Holds if the distance between both locations is less then the MaxDistance.
distanceLt(Location1, Location2, MaxDistance) :- distance(Location1, Location2, Distance), Distance < MaxDistance.

% Holds if the agent is within 200 units of the location
atLocation(Location) :- myLocation(MyLocation),
distanceLt(MyLocation, Location, 200).

% Alias for barbara_home to make the code easier to read
at(home) :- at(barbara_home).

beliefs{
% Stores the destination of the last walkTo and runTo actions.
destination(none).
% The state of the navigation. Indicates if the agent is navigating towards
% a destination, got stuck or is waiting for a new location to go to. Will be
% updated by percepts.
navigation(unknown).
% Information about ourself. Will be updated with percepts.
sel(f(unknown, unknown, unknown, unknown, unknown, unknown, unknown, unknown).

129
goals{
  % This agent wants to have some coffee.
  coffee.
}

actionspec{
  % Stops the agent in place. Next navigation state
  % percept will be navigation(waiting).
  stop {
    pre {true}
    post {true}
  }

  % Makes the agent run to the destination. Next navigation
  % state percept will be navigation(traveling).
  runTo(Destination) {
    pre {destination(PreviousDestination)}
    post {not(destination(PreviousDestination)),
           destination(Destination)}
  }

  % Makes the agent walk to the destination. Next navigation
  % state percept will be navigation(traveling).
  walkTo(Destination) {
    pre {destination(PreviousDestination)}
    post {not(destination(PreviousDestination)),
           destination(Destination)}
  }

  % Makes the agent follow the player with the given ID.
  % Next navigation state percept will be navigation(traveling).
  walkAlong(PlayerID) {
    pre {}  
    post {}  
  }

  % Makes the agent display the given emoticons. The next
  % self percepts will
  % contain the emoticons.
emote(Left, Center, Right) {
    pre {}
    post {}
}

% As it says on the tin.
jump {
    pre {true}
    post {true}
}

% Turns the given numbers of degrees.
turn(Degrees) {
    pre {true}
    post {true}
}

% Turns to the given location.
turnTo(Location) {
    pre {true}
    post {true}
}

% Does nothing.
skip {
    pre {true}
    post {true}
}

%
% The comprehension agent has two activities. A work-coffee cycle and a friendly
% meeting at the cinema. The work-coffee cycle has been implemented. The meeting
% hasn’t been yet.
%
main module {

program{
    if bel(false) then meeting.
    if bel(true) then workCoffeeCycle.
}
% Describes the scenario of two agents meeting each other.
module meeting {

    program {
        % TODO: Not yet implemented. See the assignment.
        if true then skip.
    }
}

% This module describes an everlasting work-coffee cycle.
% Brief moments of work
% interrupted by a walk to the coffee machine.
module workCoffeeCycle {

    program {
        % If we are going places, don’t bother.
        if bel(navigation(traveling)) then skip.

        % If we are at home, we work.
        if a-goal(work), bel(at(home)) then work.
        % If we want to work but are not at home, we go home.
        if a-goal(work), bel(home(Location,Range)) then runTo(Location).
        % We can order coffee at the restaurant.
        if a-goal(coffee), bel(at(restaurant)) then orderCoffee.
        % If we want coffee, we get it at the restaurant.
        if a-goal(coffee), bel(restaurant(Location,Range)) then runTo(Location).
    }
}

% Simulates the agents attempt at work. Only to result in a desire for more coffee.
module work {

program{
  if bel(not(triedWork)) then emote(none, hammer_179, none) 
  + insert(triedWork).
  if true then drop(work) + adopt(coffee) + delete(
    triedWork).
}

%%
% Takes care of ordering coffee. After the coffee has been
% ordered
% the agent will feel refreshed and inclined to work. (A
% goal work will be added).
%%
module orderCoffee {

  program{
    % Order a coffee by emoting one. (typos in emot are in
    % source).
    if bel(not(orderedOne)) then emote(none, coffee_36, none) 
    + insert(orderedOne).
    % No, actually make that a double.
    if bel(not(orderedDoubleEspresso)) then emote(coffee_36, 
      none, coffee_36) + insert(orderedDoubleEspresso).
    % That was some good coffee!
    if bel(not(jumped)) then jump + insert(jumped).
    % Time to get back to work!
    if true then drop(coffee) + adopt(work) + delete(
      orderedOne, orderedDoubleEspresso, jumped).
  }
}

%
% The agent will turn towards the counter.
%
module turnToCounter {

  program{
    % Turn towards the Counter.
    if bel(lookingAt) then exit-module.
    if bel(counter(_, Position, _)) then turnTo(Position).
  }
}
event module{
  program {
    % Store all places.
    forall bel(percept(place(ID, Location, Range))) do insert
        (place(ID, Location, Range)).

    % Update the state of the navigation.
    forall bel(navigation(OldState), percept(navigation( NewState)))
    do delete(navigation(OldState)) + insert(navigation( NewState)).

    % Update self percept.
    forall bel(self(OldID, OldName, OldLocation, OldRotation, 
                     OldEmoteL, OldEmoteC, OldEmoteR), percept(self(ID, Name, 
                                                         Location, Rotation, EmoteL, EmoteC, EmoteR)))
    do delete(self(OldID, OldName, OldLocation, OldRotation, 
                     OldEmoteL, OldEmoteC, OldEmoteR)) + insert(self(ID, Name, 
                                                                  Location, Rotation, EmoteL, EmoteC, EmoteR)).

    % Clear out percepts of the people we have seen and
    % replace them with those we see now.
    forall bel(person(ID, Name, Location, Rotation, EmoteL, 
                      EmoteC, EmoteR)) do delete(person(ID, Name, Location, 
                                                      Rotation, EmoteL, EmoteC, EmoteR)).
    forall bel(percept(person(ID, Name, Location, Rotation, 
                        EmoteL, EmoteC, EmoteR))) do insert(person(ID, Name, Location, Rotation, EmoteL, EmoteC, EmoteR)).

    % Where are we?
    forall bel(at(Place)) do delete(at(Place)).
    forall bel(place(Place, Location, Range), myLocation( 
                  MyLocation), distanceLt(Location, MyLocation, Range))
    do insert(at(Place)).
  }
}

A.4.2 Pogamut

The agent provided to the subjects using Pogamut. This agent consists of three parts. A bot plan that describes the high level logic. A behavior which implements low level actions.
And a bot logic that connects both. All three are listed below.

**Bot Plan**

```java
(documentation "Bc." "Rien" "Wandering agent.")
(C workCoffeeCycle
  (elements
    ((tryToWork (trigger ((atHome true ==) (hadCoffee true))) work))
    ((goHome (trigger ((hadCoffee true))) runToHome))
    ((orderCoffee (trigger ((atRestaurant true))) orderCoffee))
    ((visitRestaurant (trigger ((atRestaurant false ==))) runToRestaurant))
  )
)
(C turnToCounter
  (elements
    ((turn (trigger ((lookingAtCounter true))) doNothing))
    ((turnTo (trigger ((succeed))) doNothing))
  )
)
(AP orderCoffee (emoteCoffee emoteDoubleEspresso jump setHadCoffee))
(AP work (emoteWork doNothing emoteSleep oneMoreCoffee))
(DC PoshBot (goal ((fail)))
  (drives
    ((meeting (trigger ((fail true))) doNothing))
    ((workDayRoutine (trigger ((succeed true))) workCoffeeCycle))
  )
)
)
```

**Bot Logic**

```java
package nl.tudelft.sposh.comprehension;

import cz.cuni.amis.pogamut.base.agent.impl.AgentId;
import java.io.IOException;
```
import cz.cuni.amis.pogamut.sposh.executor.BehaviorWorkExecutor;
import cz.cuni.amis.pogamut.sposh.executor.IWorkExecutor;
import cz.cuni.amis.pogamut.sposh.ut2004.SposhLogicController;
import cz.cuni.amis.pogamut.ut2004.agent.module.sensor.AgentInfo;
import cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004Bot;
import cz.cuni.amis.pogamut.ut2004.bot.params.UT2004BotParameters;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbcommands.Initialize;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.BotKilled;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.ConfigChange;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.GameInfo;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.InitedMessage;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.Self;
import cz.cuni.amis.utils.exception.PogamutException;

/**
 * @author Honza
 */

public class ComprehensionLogic extends SposhLogicController
        <UT2004Bot, IWorkExecutor> {
    private String SPOSH_PLAN_RESOURCE = "sposh/plan/BotPlan.lap";

    private ComprehensionBehaviour behaviour;

    /**
     * Note that this method was originally "createBehaviour (UT2004Bot bot)".
     * Also, if your behavior is relying on some modules (e.g. { @link AgentInfo } ),
     * you have to call { @link UT2004Behaviour#initializeBehaviour (cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004Bot) }
     */
* in which the modules are initialized. {@link PreyBehaviour} is calling the method in the constructor.
* @param bot
*/

@Override
public void initializeController(UT2004Bot bot) {
    super.initializeController(bot);
    behaviour = new ComprehensionBehaviour("comprehensionBehavior", bot);
}

@Override
protected String getPlan() throws IOException {
    return getPlanFromResource(SPOSH_PLANRESOURCE);
}

/**
 * Note that this method was introduced in Pogamut 3.1 as we have switched from Python SPOSHEngine to Java SPOSHEngine
 * that allows you to have your own IWorkExecutor implementation (such as {@link StateWorkExecutor}).
 * <p>
 * Because we wanted to maintain old-fasion "behaviour" style of Prey sample, we're using {@link BehaviorWorkExecutor} here that
 * is configured with 'behaviour' previously created by {@link PreyLogic#initializeController(cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004Bot)},
 * which means that SPOSHEntities (actions/senses) are taken from this object (behaviour).
 * @return
 */

@Override
protected IWorkExecutor createWorkExecutor() {
    return new BehaviorWorkExecutor(behaviour);
}

@Override
public Initialize getInitializeCommand() {
    return new Initialize();
}
A.4 Code Materials

```java
@Override
public void botInitialized(GameInfo gameInfo, ConfigChange currentConfig, InitiatedMessage init) {
    behaviour.botInitialized(gameInfo, currentConfig, init);
}

@Override
public void botFirstSpawn(GameInfo gameInfo, ConfigChange currentConfig, InitiatedMessage init, Self self) {
    behaviour.botFirstSpawn(gameInfo, currentConfig, init, self);
}

@Override
public void botKilled(BotKilled event) {
    behaviour.botKilled(event);
}

/**
 * Create an {@link ExternalBot} with custom made logic
 * and try to connect to Unreal Server at localhost:3000
 * @param args the command line arguments
 */
public static void main(String[] args) throws PogamutException {
    new UT2004BotRunner(ComprehensionLogic.class).setMain(true).startAgents(
        new UT2004BotParameters(){
            setAgentId(new AgentId("Jan"));
        },
        new UT2004BotParameters(){
            setAgentId(new AgentId("Pier"));
        }
    );
}
```
Bot Behavior

```java
package nl.tudelft.sposh.comprehension;

import SteeringProperties.WalkAlongProperties;
import cz.cuni.amis.pogamut.base3d.worldview.object.ILocated;
import cz.cuni.amis.pogamut.base3d.worldview.object.Location;
import cz.cuni.amis.pogamut.emohawk.agent.module.sensomotoric.EmoticonType;
import cz.cuni.amis.pogamut.emohawk.agent.module.sensomotoric.Place;
import cz.cuni.amis.pogamut.sposh.SPOSHAction;
import cz.cuni.amis.pogamut.sposh.SPOSHSense;
import cz.cuni.amis.pogamut.sposh.ut2004.SposhLogicController;
import cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004Bot;
import cz.cuni.amis.pogamut.ut2004.bot.impl.UT2004BotModuleController;
import cz.cuni.amis.pogamut.ut2004.communication.messages.gbinfomessages.*;
import java.util.logging.Level;
import nl.tudelft.pogamut.emohawk.EmohawkBehaviour;

/**
 * This behaviour describes an agent with desires for food, shelter,
 * entertainment and leisure. The agent tries to satisfy these desires by
 * visiting specific places.
 *
 * When ever an agent is at a location some of these desires will be slowly be
 * satisfied (e.g. the agent desires them less). For example
 * , going to the {@link Place.CINEMA} will provide the agent with shelter and entertainment.
 *
 * When an agent is not at a location all {@link Desire}s slowly increase.
 *
 */
```
public class ComprehensionBehaviour extends EmohawkBehaviour

<UT2004Bot> {

    // Did we have any coffee?
    private boolean hadCoffee = false;

    public ComprehensionBehaviour(String name, UT2004Bot bot) {
        super(name, bot);
        // IMPORTANT: modules won't work without call of this method.
        initializeBehaviour(bot);
        // IMPORTANT: this method must be called from here in order to work!
        prepareBehaviour(bot);
    }

    /**
     * Called after the agent is initialized.
     *
     * @param info
     * @param config
     * @param init
     */
    @Override
    public void botInitialized(GameInfo info, ConfigChange config, InitiatedMessage init) {
        // This is a good place to setup SPOSH engine log level
        bot.getLogger().getCategory(SposhLogicController.SPOSH_LOG_CATEGORY).setLevel(Level.ALL);
    }

    /**
     * Called when the bot first spawns. Use this to initialise additional data.
     *
     * @param gameInfo
     * @param currentConfig
     * @param init
     * @param self
     */
    void botFirstSpawn(GameInfo gameInfo, ConfigChange currentConfig, InitiatedMessage init, Self self) {

}
/**
 * Standard sense that has to be implemented everywhere.
 * @return false
 */
@SPOSHSense
public boolean fail() {
    // it is advised to always log your senses in order for you to get a
    // grasp on how SPOSH is evaluating the plan
    log.info("fail() = false");
    return false;
}

/**
 * Standard sense that has to be implemented everywhere.
 * @return true
 */
@SPOSHSense
public boolean succeed() {
    // it is advised to always log your senses in order for you to get a
    // grasp on how SPOSH is evaluating the plan
    log.info("succeed() = true");
    return true;
}

/**
 * @return the value of hadCoffee
 */
@SPOSHSense
public boolean hadCoffee() {
    log.info("hadCoffee() = " + hadCoffee);
    return hadCoffee;
}
```java
/**
 * Senses that we are at home.
 * @return true iff we’re at a home
 */
@SPOSHSense
public boolean atHome() {
    Place place = places.getCurrentPlace();
    boolean atPlace = place != null && place.equals(Place.BARBARA_HOME);
    log.info("atHome() = " + atPlace);
    return atPlace;
}

/**
 * Senses that we are at the restaurant.
 * @return true iff we’re at a park
 */
@SPOSHSense
public boolean atRestaurant() {
    Place place = places.getCurrentPlace();
    boolean atPlace = place != null && place.equals(Place.RESTAURANT);
    log.info("atRestaurant() = " + atPlace);
    return atPlace;
}

/**
 * Senses that we are at the cinema.
 * @return true iff we’re at a cinema
 */
@SPOSHSense
public boolean atCinema() {
    Place place = places.getCurrentPlace();
    boolean atPlace = place != null && place.equals(Place.CINEMA);
    log.info("atCinema() = " + atPlace);
    return atPlace;
}
```
/**
 * Senses that we are at the park.
 * 
 * @return true iff we’re at a park
 */
@SPOSHSense
public boolean atPark() {
    Place place = places.getCurrentPlace();
    boolean atPlace = place != null && place.equals(Place.PARK);
    log.info("atPark() = \"" + atPlace);
    return atPlace;
}

/**
 * Senses that the other person is at the Park meeting.
 * Requires that the other person is visible.
 * 
 * @return true iff the other is at the park.
 */
@SPOSHSense
public boolean otherAtPark() {
    Location park = getParkMeetingOther();
    Player other = players.getNearestVisiblePlayer();
    if (other == null) {
        log.info("otherAtPark() = \"false\" (no\_player\_visible)\");
        return false;
    }
    double distance = park.getDistance(other.getLocation());
    log.info("otherAtPark() = \"" + (distance < 200));
    return distance < 200;
}

//
// =============
// SPOSH ACTIONS
// =============
//
/**
 * Standard action that has to be implemented everywhere.
 * Sleep for 250ms.
 *
@SPOSHAction
def doNothing():
    log.info("doNothing()")
    try:
        Thread.sleep(50)
    except InterruptedException:
        return True
    return True

/**
 * Shows a coffee emoticon.
 * @return True
 */
@SPOSHAction
def emoteCoffee():
    log.info("emoteCoffee()")
    emoticons.setCenterEmoticonType(EmoticonType.COFE_36)
    return True

/**
 * Shows two coffee emoticons.
 * @return True
 */
@SPOSHAction
def emoteDoubleEspresso():
    log.info("emoteDoubleEspresso()")
    emoticons.setDoubleEmoticon(EmoticonType.COFE_36, EmoticonType.COFE_36)
    return True

/**
 * Shows a work emoticon.
 */
* @return true
*/
@SPOSHAction
public boolean emoteWork () {
    log.info ("emoteWork () ");
    emoticons.setCenterEmoticonType (EmoticonType.
        HAMMER_179);
    return true;
}

/**
 * Shows a sleep emoticon.
 * @return true
 */
@SPOSHAction
public boolean emoteSleep () {
    log.info ("emoteSleep () ");
    emoticons.setCenterEmoticonType (EmoticonType.
        YAWN_254);
    return true;
}

/**
 * Sets that this agent had coffee.
 * @return true
 */
@SPOSHAction
public boolean setHadCoffee () {
    log.info ("setHadCoffee () ");
    hadCoffee = true;
    return true;
}

/**
 * Sets that this agent had coffee.
 * @return true
 */
@SPOSHAction
public boolean oneMoreCoffee () {
    log.info ("oneMoreCoffee () ");
}
hadCoffee = false;
        return true;
    }

    /**
     * The agent will run to his home.
     * @return true
     */
    @SPOSHAction
    public boolean runToHome() {
        if (navigation.isNavigating()) {
            log.info("runToHome() \w==\w true");
            return true;
        }
        runTo(Place.BARBARA_HOME.getPlaceLocation());
        log.info("runToHome() \w==\w true");
        return true;
    }

    /**
     * The agent will run to the restaurant.
     * @return true
     */
    @SPOSHAction
    public boolean runToRestaurant() {
        if (navigation.isNavigating()) {
            log.info("runToRestaurant() \w==\w true");
            return true;
        }
        runTo(Place.RESTAURANT.getPlaceLocation());
        log.info("runToRestaurant() \w==\w true");
        return true;
    }

    /**
* When the agent can see another player, he’ll walk along side of this player. Note that the steering will remain engaged even if walking failed. Handling this is up to other components.
* @return true if there is a person we can walk along with.
* /
@SPOSHAction
public boolean walkAlong() {
    if (steering.isNavigating()) {
        log.info("walkAlong() == true (already steering)");
        return true;
    }

    Player partner = players.getNearestVisibleFriend();
    if (partner == null) {
        log.info("walkAlong() == false (nobody to walk with)");
        return false;
    }

    log.info("walkAlong() == true (just started steering");
    return true;
}

/**
 * Bot Jumps Up!
 *
 * @return
 */
@SPOSHAction
public boolean jump() {
    log.info("jump()");
    move.jump();
    return true;
}

/**
* Stops both the navigator and the steering. Bot will not move.
*
* @return
*/

@SPOSHAction
public boolean stop() {
    log.info("stop()");
    navigation.stopNavigation();
    steering.stopNavigation();
    return true;
}

/*
* Initiates walkAlong steering for the given partner.
*/

@SPOSHAction
protected void walkAlong(Player partner) {
    navigation.stopNavigation();
    move.setRun();

    WalkAlongProperties walkAlongProperties = new
    WalkAlongProperties();
    walkAlongProperties.setPartnerName(partner.getName());
    steering.setWalkAlongSteering(walkAlongProperties);
    steering.startNavigation();
}

//
// =====================
// Helper functions
// =====================
//

/**
* Provides the location for the meeting..
* *
* @return the location of the meeting for Jan or Pier.
*/

public Location getParkMeeting() {
    if (info.getName().equals("Jan")) {
        return getParkMeetingJan();
    }
}
public Location getPARKMeeting() {
    if (info.getName().equals("Pier")) {
        return getPARKMeetingPier();
    }
    return null;
}

/**
 * Provides the location for the meeting in the park for Jan and Pier.
 *
 * @return the location of the meeting for Jan or Pier.
 */
public Location getPARKMeetingOther() {
    if (info.getName().equals("Jan")) {
        return getPARKMeetingJan();
    }
    if (info.getName().equals("Pier")) {
        return getPARKMeetingPier();
    }
    return null;
}

public Location getPARKMeetingJan() {
    Location offSet = new Location(0, 100, 0);
    return Place.PARK.getPlaceLocation().add(offSet);
}

public Location getPARKMeetingPier() {
    Location offSet = new Location(0, -100, 0);
    return Place.PARK.getPlaceLocation().add(offSet);
}

/**
 * Provides the location for the meeting in the park for Jan and Pier.
 *
 * @return the location of the meeting for Jan or Pier.
 */
public Location getCinemaMeeting() {
    if (info.getName().equals("Jan")) {

A.4 Code

```java
return getCinemaMeetingJan();
}
if (info.getName().equals("Pier")) {
    return getGinemaMeetingPier();
}
return null;
}
public Location getGinemaMeetingJan() {
    Location offSet = new Location(0, 100, 0);
    return Place.CINEMA.getPlaceLocation().add(offSet);
}
public Location getGinemaMeetingPier() {
    Location offSet = new Location(0, -100, 0);
    return Place.CINEMA.getPlaceLocation().add(offSet);
}

/**
 * Runs the agent to the given location.
 * @param location
 */
public void runTo(ILocated location) {
    steering.stopNavigation();
    move.setRun();
    navigation.navigate(location);
}

/**
 * Walks the agent to the given location.
 * @param location
 */
public void walkTo(ILocated location) {
    steering.stopNavigation();
    move.setWalk();
    navigation.navigate(location);
}

//
// =====================
// ADDITIONAL JAVA LOGIC
```
/**
 This method can be used to execute anything that needs to be done BEFORE
 the plan evaluation takes place. E.g. { @link UT2004BotModuleController#logic () }
 * method.
 */
@Override
public void logicBeforePlan () {
    log.info ("−−− LOGIC ITERATION −−−");
}

/**
 This method can be used to execute anything that needs to be done AFTER
 the plan evaluation takes place. E.g. { @link UT2004BotModuleController#logic () }
 * method.
 */
@Override
public void logicAfterPlan () {
    log.info ("/// LOGIC END ///");
}