Guidelines for designing interfaces to connect environments to agent programming languages

Master Thesis IN5000

Lennard de Rijk
1308270

Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science

Multimedia and Knowledge Engineering
Man-Machine Interaction Group

August 16, 2011

Exam committee:

Prof. dr. ir. C.M. Jonker
Dr. K.V. Hindriks
Dr. L. Hollink
Delft University of Technology

Acknowledgments

First of all I would like to thank my family and friends for all their support and encouragements. They helped me gather the strength needed to put this work together.

I would especially like to thank Ot and Nick for their extremely helpful feedback, which made this thesis just that much better. I hope my misuse of commas has not caused you two any permanent brain damage.

I would also like to thank everyone in the MMI lab for dealing with all the distractions I caused. I am sure it was as much fun for you as it was for me. I wish Noeska the best with her PhD and I hope my mockery of her workplace decorations can be forgiven.

My thanks also goes out to Zoltan Papp, for sharing his expertise of the traffic domain, which helped the entire simulation reach another level. Also his feedback on the thesis was of great value.

Last - but not least -, I would like to thank the supervisors, Catholijn Jonker and Koen Hindriks, for their feedback, guidance and support during the highs and lows of my work. Their combined experience ultimately led to this thesis.

Lennard de Rijk
Delft, August 2011

Summary

Multi-agent environment frameworks have been connected to *Agent Programming Languages* (APLs) before. These connections are created to combine the computational and graphical power offered by environment frameworks with the reasoning power of agents in APLs. However, only limited reasearch has been done to determine how the interface presented to the agents needs to be structured. This thesis aims to expand the knowledge in this area by providing guidelines for designing agent-facing interfaces.

The thesis is motivated by the desire to connect GOAL (an APL) to a multi-agent environment framework. This allows us to make efficient use of the power of GOAL to represent knowledge, while at the same time useing the environment modeling capabilities of an existing environment framework. We compare three environment frameworks and find Repast most suitable for our purposes.

We create the connection between GOAL and Repast using the *Environment Interface Standard* (EIS). EIS offers a language, called *Interface Immediate Language* (IIL), which allows two-way communication between the environment and the APL. Objects written in the IIL contain *percepts* and *actions* which agents use to do their work.

To make implementing EIS more manageable we add our own extension, called *EIS2Java*. EIS2Java reduces the overhead of translating between Java and the IIL. The EIS2Java framework also enables programmers to quickly make changes to the agent interface, and ensures that the code is maintainable and portable to other projects. The connection between GOAL and Repast is used to create two simulation environments, in which we build and evaluate agent-facing interfaces.

By evaluating the design and implementation of both environments, we construct the following three guidelines for building agent-facing interfaces. First, create the interface such that agents are able to spend their reasoning cycles on accomplishing important tasks that are best suited for the explicit knowledge representation of APLs. Second, introduce unique names for objects and agents, since APLs have no notion of pointers like most other programming languages. Finally, keep computationally expensive functionality in the environment, as APLs are usually less efficient at math-like computations than languages in which the environment is written.

Finally, we also conclude that EIS2Java is a helpful framework and we aim for it to be included in the EIS distribution by default.

Contents

Part I

Introduction

Chapter 1

Introduction

Multi-agent systems are systems that consist out of multiple interacting intelligent agents. Agents are written in numerous different programming languages commonly referred to as Agent Programming Languages (APLs). APLs usually focus on clearly structuring the reasoning behind decisions that agents make. For instance, 2APL, 3APL, GOAL and AGENT0 each have their own language to represent what agents know, how they think and what they do. However, APLs usually offer no direct way to model complex environments in which these agents operate. The environments are usually modeled in languages like Java or C++, where the object-oriented approach, efficient implementation and good support for graphical user interfaces make these languages perfect candidates to model environments. Combining an APL with an object-oriented language is a known problem in the area of multi-agent research, and some solutions already exist, e.g. EIS [3] and Cartago[27]. The question remains, however, what level of *detail* agents should be provided by the interface the environment offers. In this thesis we seek to answer this question.

This thesis is motivated by the desire to connect GOAL to a multi-purpose simulation framework, allowing us to make efficient use of the power of GOAL to represent knowledge, while at the same time using the extensive capabilities of existing simulation frameworks. This would in turn enable GOAL to be used more extensively for multi-agent research.

The difficulty of connecting GOAL to a simulation framework does not lie in the way the connection to the environment is created; this problem has largely been solved [3]. Rather, we consider the higher-level issue of understanding what kind of information and capabilities should be presented to an agent, and which functionalities should stay hidden in the simulation. Therefore the research question that this thesis tries to answer is:

> *Which design guidelines provide effective rules for designing interfaces that connect simulation environments to agent programming languages?*

The interface referred to in this question is the functionality of the environment that is exposed to agents written in an APL. These functionalities include the actions that agents can perform and the sensory data (percepts) that they receive.

To answer this question we connect GOAL to a multi-agent framework called Repast. Within Repast a set of simulation environments is created, based on reallife and other environments proposed within the field of multi-agent research. We develop a connection between these environments to GOAL, which is then evaluated using the environments just mentioned. This results in design guidelines for the interface offered by the environment to participating agents.

Chapter 2 covers the basics by introducing multi-agent systems, GOAL and several frameworks that can be used to built multi-agent systems. Chapter 3 compares three different multi-agent frameworks and evaluates which one is best suited to be connected to GOAL. Chapter 4 goes into depth and designs the connection between the multi-agent framework and GOAL. Part III of the thesis then describes the implementation of a highway simulation to help construct guidelines for the agent interface. Part IV presents another environment called BlocksWorld for Teams, which is also evaluated.

Finally, Chapter 14 concludes with our findings and guidelines, and Chapter 15 lists interesting possibilities for future work.

Chapter 2

Preliminaries

In this thesis we deal extensively with multi-agent systems in general and GOAL as an agent programming language. This chapter introduce both these topics. Section 2.1 introduces multi-agent systems and Section 2.2 summarizes several frameworks for building a multi-agent system. The final section (2.3) introduces GOAL.

2.1 Multi-Agent System

This section introduces the concept of multi-agent systems. First a definition for rational agents is given in Section 2.1.1. Then software agents are described in Section 2.1.2. The classifications of the different types of multi-agent environments are given in Section 2.1.3. Finally several architectures for building software agents are introduced in Section 2.1.4.

2.1.1 Rational Agents

One of the adopted definitions of a rational agent can be found in a book from Russell [29].

> A **rational agent** is one that acts to achieve the best outcome or, when there is uncertainty, the best expected outcome.

2.1.2 Software Agents

A multi-agent system is a system that consists of interacting intelligent software programs called agents. Intelligence in agents can be defined by the capabilities the agents exhibits[39]. Wooldridge [41] suggests that the following three capabilities are required for an agent to be considered *intelligent*:

Reactivity Agents are able to perceive their environment and respond to changes in a timely manner. Depending on the environment in play “a timely manner” can vary from a few milliseconds for mission-critical decisions to a number of seconds when playing a game of backgammon.

Pro-activeness Agents are able to instantiate actions independently, i.e. they do not merely react to actions made by others. They are able to take control of the situation and make things happen rather than just waiting for input and act accordingly. An example of proactive behavior is an agent that offers to help other agents without being asked to do so.

Social ability Agents are able to interact with other entities, such as other agents and humans, to meet their goals. Note that this is not only about being able to communicate, but also about being able to reason and cooperate with others that might have a different goal than the agent.

In other words the goal of any good agent architecture is to offer a way to create a reactive, proactive and social agent that act, to achieve the best possible outcome. The architectures in Section 2.1.4 aim to incorporate these three capabilities.

2.1.3 Multi Agent Environments

Multi-agent systems have to be able to operate in a range of possible environments. According to Russell these environments can be categorized along five different dimensions[29]. We use these five different dimensions for defining our own environments.

Observable versus partially observable An *observable* environment is one in which the agent can acquire correct and complete information about the state of the world at any time. If it is impossible for the agent to retrieve the information needed to make a decision at any time, or if the information might be wrong due to inaccurate sensors, an environment is called *partially observable*. In case of partially observable environments an agent usually needs to maintain an internal representation of the state of the world.

Deterministic versus stochastic In a *deterministic* environment the next state of the world is completely defined by the current state and the action taken by the agent, otherwise it is called a *stochastic* environment. The definition of a deterministic environment, as given by Russell [29], ignores the fact that other agents might be performing actions that change the state of the world. Therefore a game can be deterministic, even though the actions of the other agents are unpredictable.

Episodic versus sequential In an *episodic* environment the life of an agent can be divided in clear episodes. In each episode the agent receives percepts and performs a single action. Very important here is that the next episode does not depend on any of the previous ones. If the life of an agent can not be divided into episodes the environment is called *sequential*.

An example of an episodic environment is that of an agent in charge of a simple entrance gate. Letting the next person in does not depend on the previous person. Sequential environments are games of chess or backgammon: in these games previous moves can and usually will have consequences later on.

Static versus dynamic An environment is classified as *dynamic* when its state can change while the agent is reasoning, otherwise the environment is classified as *static*. In a static environment the agent does not have to worry about time, to a certain extent, nor does it have to keep observing the world while deciding upon an action. Building an agent for a static environment is therefore usually considered easier than building one for a dynamic environment.

A game of backgammon can be considered static since its state does not change while the agent is thinking about the next move. On the other hand an agent that is controlling elevators lives in a dynamic world where, the elevator call button on another floor can be pressed, while the elevator is still processing where to go next.

Discrete versus continuous *Discrete* and *continuous* refer to how time and percepts are handled in an environment. For instance in backgammon time is measured in turns and there are a finite number of states in which the game can be, this makes it a discrete environment. On the other hand, an agent controlling a production line at a factory deals with a continuous environment, where the speed of the conveyor belt changes over time in a smooth and continuous way.

2.1.4 Software Agent Architectures

There are numerous different architectures for building a rational software agent. Most of the popular ones can be found in a survey by Wooldridge [40]. In this section we discuss two different architectures in some detail. In Section 2.1.4.1 we discuss the BDI architecture on which the APL GOAL, introduced in Section 2.3, is based. Section 2.1.4.2 details the hybrid agent architecture which combines reactive and deliberative components.

2.1.4.1 BDI Agent Architecture

BDI stands for Belief-Desire-Intention and originates from the human reasoning model developed by Bratman [6]. It is used in this context as the basis of a software architecture for building rational agents as presented by Rao [26]. It discerns three aspects of the agent's reasoning.

Beliefs The *beliefs* stand for the informational state of the agent. They represent what the agent knows about itself and the environment. Beliefs are usually stored in a kind of database, commonly referred to as a belief base. The beliefs stored might not necessarily be true: for instance, they may have become false over time, or a faulty sensor may have led to wrong conclusions.

Desire The *desires* of an agent describe what the agent would like to accomplish. If an agent is actively trying to accomplish a desire, the active desire is also referred to as a *goal*. Goals adopted by the agent must be consistent. An example is that an agent is not allowed to simultaneously have the goals "go out to lunch today" and "stay home today" even if the agent desires to accomplish both of them.

Intention The *intention* of the agent stands for what the agent has chosen to do. It means that the agent has committed to executing certain actions to work towards one of its goals. The sequence of actions that an agent wants to perform to achieve its intention is referred to as a *plan* in the literature. Plans may be nested or partially complete, the latter helps when the environment of the agent is changing due to external factors.

The BDI architecture is a very open architecture and its exact implementation is up to the programmer. Implementations range from pure implementations (usually backed by logical programming languages [14, 17]) or ad-hoc/hybrid implementations that use the BDI mindset to structure their code [2, 28].

2.1.4.2 Hybrid Architectures

Architectures where time is almost exclusively spent on reasoning about the model of the world are called *deliberative*. On the other hand architectures where barely any time is spent thinking ahead and the agent's action are a direct response on input are called *reactive*. Many researchers have suggested that purely deliberative and reactive architectures cannot be used for building suitable agents [40]. The solution they suggest is a hybrid architecture, a marriage between a reactive and a deliberative architecture.

In such an architecture there is a reactive component that is able to react directly to events in the world, such as imminent danger or other, usually time-critical, events. The deliberative component is responsible for long-term behavior or planning to achieve the agent's goal. This bears great resemblance to how the human mind works, where the parts that are involved in survival are located in the front of the brain while those related to planning and reasoning are located deeper in the back of the brain. Examples of different hybrid architectures are those of Ferguson [9] and Georgeff [11].

Hybrid architectures are important because they draw a parallel to the problem we are trying to solve in this thesis. Most of the code responsible for reactive behavior in an environment is usually written and called from Java while the deliberative part of the system is handed over to an APL.

2.2 Multi-Agent Environment Frameworks

This section introduces different frameworks for creating multi-agent systems. All of these frameworks are different in nature and some of them are investigated further to determine their applicability for creating multi-agent simulation environments.

2.2.1 MARS

MARS stands for Multi-Agent Real-time Simulator and is a proprietary simulator built by the Dutch Organization for Applied Scientific Research (TNO). This system is capable of simulating discrete and continuous dynamical systems. A main component in this simulator has three methods it supports. The first method is *sample*, which collects data from the sensors of an agent. The second method is *output* which determines the action an agent should take. The third and final method is *step*, which updates the state of the agent. MARS supports linking with other simulators and real-time parameter changes. It has been used in an intelligent traffic system setup before [23].

2.2.2 JADE

JADE stands for Java Agent DEvelopment Framework and is a platform developed for interaction between FIPA-compliant agents. A FIPA-compliant agent has the ability to communicate in FIPA-ACL or Agent Communication Language [10]. This language is intended as a common standard for agents built in different systems such that they can share part of their ontology with each other, enabling agents to communicate. JADE supports FIPA-compliant agents and is written in Java. It is mainly used for creating an inter-agent communications platform rather than a complete simulation. Many agent frameworks are therefore built on top of JADE, e.g. [4, 24]. A more detailed description on JADE can be found in [5].

2.2.3 Repast

Repast is an interactive cross-platform Java-based agent modeling and simulation toolkit, designed to support flexible models and dynamic model adjustment. Repast is open source and free to use. The installer ships with a modified version of Eclipse, which is the IDE of choice for programming models. The simulation is based on a concurrent discrete event scheduler, meaning that time is not exactly continuous but is incremented in so called *ticks*. Methods to be executed can be scheduled at a certain time and can be repeated at set intervals. The toolkit has built-in simulation logging and graphing tools, allowing for a range of ways to export data collected during simulations.

2.2.4 MASON

MASON stands for Multi-Agent Simulator of Networks and is a discrete-event multi-agent simulation library written in Java. It is open source and free to use. It is similar to Repast in that it also works on basis of a discrete event scheduler. It has a strict framework in which agents must be molded. In other words every agent must implement the so called *Steppable* interface. Every agent has a step function which is called with the current state of the simulator. The toolkit supports discrete and continuous 2D and 3D simulation and display, as well as a built-in inspector for the agents which allows real-time inspection and modification of the agent's parameters.

2.2.5 Brahms

Brahms is "a set of software tools to develop and simulate multi-agent models of human and machine behavior" [2], developed at NASA's AMES Research Center. These tools include a compiler and a virtual machine for the Brahms Language, in which the models are defined.

One of the most distinguishing characteristics of Brahms is that it is specifically designed to allow the modeling of *work practice*. Brahms' creators define work practice as follows:

> The collective performance of contextually situated activities of a group of people who coordinate, cooperate and collaborate while performing these activities synchronously or asynchronously, making use of knowledge previously gained through experiences in performing similar activities. [32]

One of the key concepts in this definition is *activities*. Activities form the smallest unit of execution in a Brahms model, rather than plans such as in common BDI systems. To make sure models remain workable however, the right granularity must be obtained when activities are specified. The most suitable level of detail depends on the nature of the model and lies between the high level formal process models on one side and the low level cognitive models on the other. More information and the installers for Brahms can be found at [2].

2.3 Agent Programming in GOAL

We choose to program our agents in GOAL since it is a language which is developed at the Delft University of Technology and one with which the author is familiar. However as we discuss in Chapter 14 we could have chosen any other language such as 2APL, Jadex or Jason.

GOAL is an agent programming language built specifically for rational agents [14]. It takes concepts from the Believes-Desires and Intentions (BDI) model [6], which is a conceptual model developed to for writing intelligent agents. GOAL stays true to this model by having a mental state that consists of knowledge, beliefs and goals. GOAL agents receive information about the world in form of percepts, which an agent can use to for instance create beliefs or update goals.

The GOAL language is implemented on top of the Prolog language. Prolog is a general purpose logic programming language in which everything is represented in the form of facts and rules.

The downside of using Prolog is that it adds extra computational load of the environment. The complexity of programs usually increase by 25-50% when written in Prolog [33].

GOAL agents are run in environments that are written in Java. These environments comply with the Environment Interface Standard (EIS), which is introduced in Section 4.1. Environments are launched and controlled using GOAL IDE. The GOAL IDE has support for inspecting the belief base of the agent and other tools that help with debugging.

Part II

Connecting GOAL to Repast

Chapter 3

Agent Based Modeling & Simulation Platforms

This chapter investigates three different agent based modeling & simulator platforms. These platforms make it easier to program models and agents that interact with these models. We start by defining agent based modeling in Section 3.1. We then consider platforms that support agent based modeling and simulation. Since there are many such platforms, each with their own quirks and limitations, we limit ourselves to three platforms which we have access to, some form of experience with, and which are known inside our field of research.

The three platforms are MARS, Repast and MASON, all of which are introduced in section 2.2. We compare them in Section 3.2 based on a set of criteria to conclude which platform is most useful to be connected to GOAL.

3.1 Agent Based Modeling & Simulation

Agent based modeling is a method for studying systems exhibiting two properties:

1. Firstly the system is composed of interacting rational agents as defined in Section 2.1.1.
2. The system exhibits emergent properties.

These emergent properties arise from interactions of the agents and cannot be deduced simply by aggregating the properties of the agents. When the interaction of the agents is dependent on the past, and especially when the agents adapt to past experiences, mathematical analysis is limited in its ability to derive the dynamic consequences. In these cases agent based modeling might be the only practical method of analysis [35].

3.2 Comparison

The three modeling platforms to be compared all require that most of the work is done in Java. We specifically choose Java since it is multi-platform, highly interoperable and because we have experience with its programming paradigms.

The platforms are compared on six criteria which we introduce first in Section 3.2.1. The actual comparison of the different platforms is done in Section 3.2.2.

3.2.1 Comparison Criteria

This section introduces the six criteria on which the agent based modeling and simulation platforms will be compared. These criteria have been formulated based on personal experience.

Modeling Tools Does the platform offer tools for modeling the environment? In other words what primitives, like basic geometry, are available for the programmer to model the environment, or does everything have to be modeled from scratch.

Simulation Tools Does the platform provide tools to support the simulation? These tools can range from simple controls to start/stop/pause the simulation to complete visualization frameworks or tools that allow aggregation of data from the simulation.

Viability How well equipped is the platform to model different environments? E.g. ranging from a microscopic traffic simulation to an ant farm. To evaluate this criterion we consider available environments and other research surrounding the platform to gauge its capability.

Ease of use How easy is the platform to understand? What kind of learning curve is needed to implement a simple multi-agent scenario? Environments that are not easy to use might take more time to get implemented, the extra time investment required might not weigh against the goals of the research.

Documentation Documentation is vital when it comes to choosing a platform to work with since it will save you time if the platform is well documented. To evaluate documentation investigate the quality of the documentation of the code and any user guides or tutorials that might be available.

Openness How open is the platform? I.e. is all the source code available, how many other researchers have openly published their code and is there an active community around the platform in general that can offer support?

These six criteria are used to evaluate whether a platform has the potential to be a useful addition for researchers that want to work with GOAL.

3.2.2 Platform Comparison

We base the score of the various platforms on our personal findings, advice from experts in personal communication and the paper by Railsback et al.[25]. Table 3.1 shows how each of the three platforms scored on the different criteria.

On the first criterion MARS scores lowest, because of its lack of built-in representations of 2D and 3D environments. Both Repast and MASON do have such modeling tools available, such as a framework for describing movement in

	MARS	Repast	MASON
Modeling Tools	+/−	+	+
Simulation Tools	−	++	+
Viability	+/−	+/−	−
Ease of use	+/−	+	−
Documentation	−	+	+/−
Openness	−	+	+

Table 3.1: Comparison of different modeling & simulation platforms

2D/3D space and a clear way of representing the objects that are present in the world.

Simulation tooling is not really available in MARS, it only allows starting and stopping of the simulation and it therefore scores the lowest on this criteria. Both MASON and Repast support visualization, screenshots, videos and checkpoints. However Repast has built-in support for data collection, export and graph display, it therefore scores higher than MASON.

MASON is highly suited for discrete environments and networks unlike the highway environment where continuous coordinates play a big role. It therefore scores the lowest since it would take serious molding of continuous problems to make it fit into MASON. MARS and Repast have similar capabilities, however they both lack examples of serious multi-agent environments.

Of the three platforms Repast is easiest to get started with, the setup time is minimal and one can jump right into modeling without having to worry about it costing too much work. MASON and MARS both require several software packages to be installed separately, making it harder to get started.

Repast scores the highest on documentation, the framework code is highly documented and there are tutorials available for a range of different use cases. MASON's current documentation is not written clearly, the tutorials seem to lack focus as well.

Repast and MASON both score high on the openness factor since they are both open source and are actively being used in research papers over the last years according to a library search. MARS on the other hand is closed source and has a limited number of known publications, roughly twenty times less than Repast and MASON over the last two years.

Looking at the summary in table 3.1 we conclude that the best agent based modeling and simulation platform to connect to GOAL is Repast.

Chapter 4

Connecting Repast and GOAL

In this chapter we create a connection between GOAL and Repast to facilitate the use of different Repast environments for research on multi-agent systems.

To achieve this we first introduce the Environment Interface Standard (EIS) in Section 4.1, this is the standard we use to connect GOAL to Repast.

Section 4.2 introduces a way to enhance the use of EIS in environments such that the overhead of creating new actions and percepts is significantly reduced. We conclude in Section 4.3 with the design and implementation of the connection between GOAL and Repast that makes use of EIS.

4.1 Environment Interface Standard

The *Environment Interface Standard (EIS)* is a proposed standard for connecting different agent-platforms (2APL, GOAL, Jadex and Jason) and environment models, created by Behrens et. al. in [3].

EIS assumes that the environment models entities that can be controlled. These *controllable entities* are then controlled by agents. Controllable entities must provide two things. First, they provide capabilities that can change the state of the environment, so called *actions*. Second, they provide *percepts* which forms the sensory information that is made available to the agent.

It is the job of EIS to offer a way to translate these actions and percepts coming from the environment into something that can be understood at the agent-platform level. This is achieved with a language that is used on both the agent-platform side as well as the side of the environment. This language is called *Interface Immediate Language (IIL)* and is the common ground that both the agent-platform and the environment use to communicate. The IIL models two basic structures namely *Percepts* and *Actions*, which are the two aspects of controllable entities. Percepts and actions are defined as having a name and containing an ordered list of *Parameters*, i.e. the action $goTo(1, Fast)$ is an action named $goTo$ with 1 as its first parameter and $Fast$ as its second parameter. Parameters model primitive data types of the IIL, they define concepts such as numbers, strings and lists.

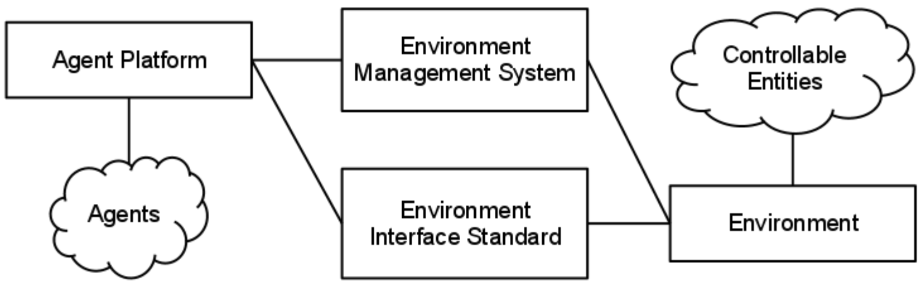

Figure 4.1: The EIS sits between the Agent Platform and the Environment.

Now that the ideas behind EIS have been introduced we take a look at the actual implementation. EIS is currently implemented in Java and operates at the level between the environment and the agent programming language (see also Figure 4.1). We discuss the different components which EIS connects to and what kind of sofware we use to implement them.

The *Agent Platform* is defined as the platform on which software entities, called *agents,* run. These agents are able to process percepts and initiate actions. As an agent platform in this thesis we use GOAL, which has been introduced in Section 2.3.

The *Environment* is software that contains controllable entities. In this thesis we use environments that are built on top of Repast.

The *Environment Management System (EMS)* allows the agent platform to manage the state of the environment. It offers the functionality to stop, pause and resume the environment. This component is implemented by us since it differs for every environment.

The *Environment Interface Standard (EIS*) is the core component which allows the agent platform to perceive and act upon the environment. For every environment the implementation of EIS is slightly different. However some functionality defined in EIS is the same across different environments. For this reason the authors of EIS created the abstract Java class *EIDefaultImpl*, which contains that common functionality. In Section 4.2 we discuss this default implementation and how to extend it to make it easier to use with different kinds of environments. Section 4.3 details how we create the connection between EIS and Repast. The connection between GOAL and EIS is already implemented by the authors of GOAL and is not detailed in this thesis [15].

4.2 Simplifying the use of EIS

This section aims to simplify the way Java environments are made compatible with EIS. In order to qualitatively improve the code on a software engineering level. This simplification also makes it easier to make small adjustments to the interface exposed to agents.

The current strategy of implementing EIS for environments, especially when it comes to generating percepts and actions, is introduced in Section 4.2.1. Section 4.2.2 explains the downsides of this strategy. A new method for making environments EIS compatible, called EIS2Java, is introduced and evaluated in

Code Snippet 4.1 Entity in the *HelloWorldEnvironment*.

```java
public int getPrintedTextAmount() {
  return printedTextAmount;
}

public String getLastPrinted() {
  return lastPrinted;
}

public void printText(String text) {
  System.out.println(text);
  lastPrinted = text;
  printedTextAmount++;
}
```

Section 4.2.3.

To visualize the differences between the old strategy and EIS2Java's approach we use an example called the *HelloWorldEnvironment*. An *Entity* in this environment is capable of printing text to the standard output in Java. The entity is also able to sense how many times it has written something to the output and what was the last string it wrote. The actions and percepts available are public methods in the *Entity* class. A partial implementation of the entity can be found in Snippet 4.1.

4.2.1 Actions and percepts directly in EIS

In this section we go through Snippet 4.2 which is a simplified version of what currently needs to be done to support percepts and actions in an environment. This approach is evaluated in Section 4.2.2.

All of the functions written are part of the Environment Interface and are therefore present in a single class that also deals with other things like actually coupling an agent to an entity, see also EIS in Figure 4.1.

To support actions a programmer has to implement two methods, *performEntityAction* and *isSupportedByEntity*. The latter of these methods checks whether an entity actually supports the given action. It involves at least a check on the name of the action and the number of arguments present. The perform method unwraps the action and tries to translate the arguments into something that can be understood by the entities. If it is successful in translating it tries to perform the requested action. In this example the action does not return a percept, but this is possible, and in that case it means that the return value would need to be translated into a Parameter so that it can be processed by the agent platform side of EIS.

Dealing with percepts is similar to actions in some sense: the programmer needs to implement a method called *getAllPerceptsFromEntity* in the environment. As the snippet shows this method has to locate the entity with the given name and create percepts for each concept that is deemed interesting. This involves giving values a name and translating them into IIL.

Code Snippet 4.2 Code for actions and percepts directly in EIS.

```
protected Percept performEntityAction(String entity,
    Action action)
     throws ActException {
  Entity e = entityMap.getEntity(entity);
  if(action.getName().equals('printText')) {
    Parameter param = action.getParameters().get(0);
    if(!(param instanceof Identifier)) {
      throw new ActException(...)
    }
    String text = ((Identifier)param).getValue();
    e.printText(text);
  } // else if(action.getName().equals('
     someOtherAction')) { ... }
  return null;
}

protected boolean isSupportedByEntity(Action action,
    String entity) {
  if(action.getName().equals('printText')) {
    return action.getParameters.size() == 1;
  } // else if(action.getName().equals('
     someOtherAction')) { ... }
  return false;
}

protected LinkedList<Percept> getAllPerceptsFromEntity(
    String entity) throws ActException {
  LinkedList<Percept> percepts = new LinkedList<Percept>();

  Entity e; //Retrieve the entity in some way
  int printedTextAmount = e.getPrintedTextAmount();
  percepts.add(new Percept(
    "printedText", new Numeral(printedTextAmount)));
  String lastPrinted = e.getLastPrinted();
  percepts.add(new Percept(
    "lastPrintedText", new Identifier(lastPrinted)));

  return percepts;
}
```

4.2.2 Evaluating programming directly in EIS

The previous section explained what needed to happen to implement the environment side of EIS. While this approach does works for small environments there are three downsides that become apparent when the environment grows.

First, the implementation of actions and percepts directly in EIS does not scale nicely with regards to the size of the environment. Imagine having an extensive environment which usually has a lot more actions and percepts available. For instance the Unreal Tournament 2004 environment that can be connected to GOAL has around ten different percepts and ten different actions [15]. If this would be done using the current approach where the code is put directly into EIS, this would greatly increase the amount of code present in *performEntityAction* and *getAllPerceptsFromEntity,* up to a point where it is hard to maintain and test the correctness of the code. Imagine as well having support for different types of entities each with their own set of available actions, this has the potential to make the code even more cluttered.

While actions and percepts for different entities can be delegated to separate methods and objects, the entire implementation still has the potential to grow excessively. Such is the case for the Unreal Tournament environment. After manual inspection of the Unreal codebase over two thousand lines of code are involved in gathering percepts, executing actions and make translations to and from the IIL.

A second downside is the duplicated code in the example. While the names of the actions/percepts can be factored out, it is clear that if for instance the number of arguments of a function are changed, the code would need to be changed in at least two different places. This means that the definition of what an action exactly entails (the name, number of arguments and type of arguments) is made in two different places. This can lead to programming errors and is bad for the maintainability of code. This breaks one of the rules of the Don't Repeat Yourself (DRY) principles, namely, *every piece of knowledge must have a single, unambiguous, authoritative representation within a system* [18].

Finally there many of manual translations defined between the IIL, especially the *Parameter*, and Java objects. Although the example in the previous section is quite small, you can already see that it is not very convenient to have to write the same five lines of code over and over again for each environment that is made just to translate a parameter to a string. These kinds of translations are not happening in a localized place, meaning that code is duplicated across projects. This once again increases the maintenance cost of the code and allows for errors between different versions.

4.2.3 EIS2Java: A framework to assist with EIS-ifying an environment

Section 4.2.2 shows that the current way of implementing EIS for an environment has serious drawbacks when it comes to scalability, testability and maintainability. For this thesis we have implemented a framework, called EIS2Java, that reduces these drawbacks and makes it easier to have an environment EIS compliant. Note that while this report focuses specifically on the connection between Repast and GOAL, this framework is designed so that it can be used for *any* environment that can interface with Java.

The idea behind EIS2Java is to automatize the discovery of actions and percepts that are made available to an agent without imposing additional restrictions on the environment itself, as well as ensuring that EIS still is non-intrusive in the code of the environment. To achieve automated discovery and a low level of intrusiveness we have opted for an approach that involves annotations. Annotations are a way of adding meta-information to methods inside Java[1], in our case whether a method is an action or whether it should should be seen as a producer of percepts.

The *AsPercept* annotation can be applied to any method which has no arguments and has a return value of any type. For the *AsAction* annotation there are no restrictions on the signature of the method.

The values passed into methods annotated with AsAction and the value returned by those annotated with AsPercept can be any Java object. We do not leak any IIL objects to the environment, unless the environment itself is defined using them. This is because EIS2Java automatically handles translation from Java objects to Parameters and vice versa for common Java primitives and structures, such as Integers, Strings and Collections.

If an environment wants to use other Java objects, a translator can easily be written by implementing the right translation interface (see Code Snippet 4.3 and 4.4). These interfaces allows translations of objects to be defined at a single place and be tested separately from other code.

The automatic discovery of actions and percepts in EIS2Java is based on the fact that environments in EIS are responsible for registering controllable entities. Instead of only passing a string identifying the object, the actual object itself is passed along to the method which exposes entities in EIS. At this stage EIS2Java uses reflection to find all annotated methods and cache them for later use when percepts or actions need to be executed. The implementation of the code that processes the annotations, retrieves percepts and performs actions can be found in Appendix A.

Code Snippet 4.5 shows what the implementation of the *HelloWorldEnvironment* looks like with EIS2Java. Note that only three lines of code were added to the entity. The rest of the code is hidden in the reusable EIS2Java implementation of the EIS interface and takes care of all the translations and argument checking.

EIS2Java addresses the concerns posed in Section 4.2.2. It is easy to extend an environment with new percepts and actions, a single annotation is all that is needed.

Names of percepts and definitions of actions are found in a single place. When the signature of an action needs to be changed only the actual Java method signature would need to be touched, any code in EIS remains unchanged. Moreover, since all translations happen inside EIS2Java by use of the *Java2Parameter* and *Parameter2Java* interfaces, translations for a single type are defined in one location and can be modularly reused across multiple projects.

Finally we estimate that the percept and action related code for the Unreal Tournament environment mentioned in Section 4.2.2 can be reduced by as much

[1]More information can be found at `http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html`.

Code Snippet 4.3 Interface for translating from Java to Parameters.

```
/**
 * Interface for translating Java objects of type T to
     a Parameter.
 *
 * @param <T> The type of the Java object to translate
    .
 */
public interface Java2Parameter<T> {
  /**
   * Translates the object into an array of Parameter.
   *
   * @param o The object to translate.
   * @return The array of Parameter that represents a
      translated object.
   * @throws TranslationException if the translation
      can not be made.
   */
  Parameter[] translate(T o) throws
     TranslationException;

  /**
   * @return The class that is being translated.
   */
  Class<? extends T> translatesFrom();
}
```

Code Snippet 4.4 Interface for translating from Parameters to Java.

```java
/**
 * Interface for translating a Parameter into a Java
   object of Type T.
 *
 * @param <T> The type of the Java object to translate
   .
 */
public interface Parameter2Java<T> {
  /**
   * Translates the Parameter into an object of type T
     .
   *
   * @param <T> The type of object to translate the
      parameter to.
   * @param parameter The parameter to translate.
   * @return The Parameter that represents a
      translated object.
   * @throws TranslationException if the translation
      can not be made.
   */
  T translate(Parameter parameter) throws
     TranslationException;

  /**
   * @return The class that is translated to.
   */
  Class<T> translatesTo();
}
```

Code Snippet 4.5 The entity when EIS2Java is used.

```java
@AsPercept(name = "printedText")
public int getPrintedTextAmount() {
  return printedTextAmount;
}

@AsPercept(name = "lastPrintedText")
public String getLastPrinted() {
  return lastPrinted;
}

@AsAction(name = "printText")
public void printText(String text) {
  System.out.println(text);
  lastPrinted = text;
  printedTextAmount++;
}
```

as 50% if it would use EIS2Java.

More information about how to use EIS2Java can be found in Appendix B.

4.3 The connection between GOAL and Repast

In this section we design the connection between GOAL and Repast. First we explain how we deal with Repast's discrete event scheduler when it is combined with EIS in Section 4.3.1. Section 4.3.2 deals with the problem of getting GOAL to launch an environment written for Repast.

4.3.1 Dealing with Repast's discrete event scheduler

EIS makes no assumptions on the scheduling of either the agent platform or the environment. This presents a challenge because Repast itself is based on a concurrent discrete event scheduler. We decided to interfere with this setup as little as possible and leave the responsibility of how to deal with concurrent data access to the programmer. This means that percepts can be requested at any time during the simulation and it is the programmer's job to decide what kind of consistency is guaranteed for the returned data. Actions are quite similar in that they may be executed during the time other scheduled methods in the environment are being executed. A programmer who wishes to circumvent this can wrap the execution of the actions from agents in a method that can be queued with the discrete event scheduler for execution at the earliest convenience.

4.3.2 Launching environments from GOAL

GOAL uses Java programs packaged in JARs to launch environments[16]. The main-class specified in the JAR's manifest should implement the EIS; it is instantiated by GOAL using the empty public constructor.

Since Repast is built on the Simphony Application Framework (SAF[2]), it has a peculiar way of loading classes during runtime. The classes that need to be loaded are defined in XML files. This loading mechanism does not allow any Repast classes except those in the runtime package to be available on the classpath at start up. This makes creating a single JAR containing both Repast and the actual environment a near impossible task and it would require a lot of manual effort when new versions of Repast would be released. We therefore opted for an approach in which the user is required to install Repast separately.

With Repast installed separately there are two ways of enabling the support of EIS. First we could run Repast in another VM and use Remote Method Invocation (RMI) to remotely connect to GOAL through the EIS framework it exposes or we could run Repast in the same VM as GOAL.

Early investigation showed that using RMI is sub-optimal since EIS does not directly support the separation of concerns that is needed to make this work. For instance, environment management and agent management are all cluttered into a single class, which makes it difficult to setup an RMI version of EIS without having to write a complicated layer in-between. RMI should however be the eventual goal since it makes it easier to deal with the complicated loading mechanism of Repast. This can by accomplished by re-writing the

[2]`http://old.nabble.com/saf-source-code-td19221478.html`

current interface into a more modular structure where the only information traveling between the APL and environment are actions, percepts and simple management code.

Because of the complicated loading framework we need to call the main method of the Repast framework in the implementation of EIS to start the environment. To accomplish this, GOAL needs to have the Repast runtime on its classpath. More information on how to create an Repast environment that works with GOAL is given in Appendix C. We allow the initialization parameters of EIS to contain the location of the Repast scenario that should be loaded on start-up. The user can also manually load a scenario in Repast by using the menu in the Repast interface. For action and percept definitions we use the EIS2Java framework that is defined in Section 4.2.

The environment management component is implemented by subscribing the Repast environment to changes in its own state. These state changes are then communicated to EIS, where they are handled appropriately.

4.4 Conclusion

In this chapter we built a connection between GOAL and environments written in Repast. While doing so we constructed a framework to make it easier to create environments that are compatible with EIS, even if those environments are not written in Repast. We call this framework EIS2Java and it uses advanced concepts of Java such as reflection and annotations to do its work. The connection between Repast and GOAL does not impose any restrictions on the functionality of Repast. This means that the data analysis tools and other functionality of Repast can be unchanged while GOAL controls the entities.

Now it is time that we put the connection between GOAL and Repast in use to construct guidelines on creating agent-facing interfaces. We do this by implementing two different environments in Repast, they can be found in Parts III and IV of this thesis.

Part III

Highway Simulation

Chapter 5

Environment

In this chapter the highway environment is detailed. Section 5.1 lays out the details of the environment. Finally Section 5.2 analyzes the tasks agents need to be able to perform in the environment.

5.1 The highway environment

Section 2.1.3 showed that the environments of multi-agent systems can be classified amongst five dimensions. In this section we describe the environment according to these five dimensions.

The goal is to build an agent that can drive successfully from point A to point B. Since this is a very broad objective, we restrict ourselves to a simple environment: a highway. The highway is one-way and has at least two lanes. The agent is situated in a vehicle in which it can perform certain actions, namely steering and application of the throttle and brakes. The vehicle is equipped with sensors allowing it to sense the position of other cars surrounding the vehicle. Communication equipment is also present so that communication can take place between other vehicles and road-side units.

The classification of the environment, according to the five dimensions in Section 2.1.3, is as follows. First, the environment can be classified as partially observable, since the agent is not able to obtain information about the current state of the entire road network at any time. Nor can the agent detect what other automated vehicles are thinking.

Second, the environment can be considered stochastic, since it is hard to predict the behavior of traffic. For example, tires may blow out at any time or other mechanical failures might occur.

Third, driving is clearly a sequential task, where one action may have future consequences. Short term actions such as a lane change may result in more lane changes in the future.

Fourth, an agent controlling a car is situated in a dynamic environment, since during deliberation the car itself and the other cars keep moving, thereby changing the state of the world.

Finally, the environment is dynamic, because percepts such as speed and steering angle are measured on a continuous scale.

5.2 Task Analysis

In this section a description is given of the different tasks a driver has to accomplish, these tasks are used in the next chapters to formulate a design of the system. Section 5.2.1 introduces measurable concepts of safety on the road. The first task that is introduced in section 5.2.2 is very basic but common task which is changing the lanes. Section 5.2.3 discusses merging and Section 5.2.4 discusses overtaking.

5.2.1 Safety

Safety is very important when human lives are at stake. Not only do accidents cause personal injury but they can also severely damage the economy. It is estimated that traffic jams hurt the Dutch economy for over one billion Euros each year[12]. It is estimated that 13% of those traffic jams are related to accidents. Since this chapter analyzes the high level tasks a driver has to perform it is good to define some measures that can help us define how safe a situation is.

The first measurement is called Time-HeadWay (THW), it defines the time it takes for the vehicle to cover the distance to the vehicle in front. Formula 5.1 shows how THW is calculated. t_2 is the tail of the vehicle in front, h_1 is the head of the vehicle for which THW is being calculated and v_1 and v_2 are the speed of the vehicles.

$$THW_1 = \frac{t_2 - h_1}{v_1} \tag{5.1}$$

The second measurement is called Time-To-Collision (TTC), it defines the time it takes for the gap between two cars to close if they keep traveling at their current speed. Formula 5.2 shows how this is defined.

$$TTC_1 = \frac{t_2 - h_1}{v_1 - v_2} \tag{5.2}$$

In highway situations humans are usually expected to keep the THW above two seconds and the TTC larger than five seconds[22].

5.2.2 Changing lanes

A basic task that occurs in highway situations is changing lanes. This lateral movement is usually performed by gradually steering the car into the other lane and then straighten it out to stay on the other lane.

Goal The goal of this action is to change the lane the car is currently occupying l_c to l_n where $l_c \neq l_n$.

Preconditions The car is allowed to drive on lane l_n.

Lane l_n is considered safe. This can be determined from using the formulas from section 5.2.1. If the car would only change lateral coordinates to lane l_n the safety conditions should still hold. Also, the back area of the car should be clear since you don't want to cut cars off.

Action sequence

1. Indicate to which lane you want to change.
2. Steer in the direction of lane l_n.
3. When the car is almost completely in lane l_n, the car should steer as to level out in lane l_n.

Abort conditions When during the lane change the safety conditions in lane l_n are violated, the lane change should be aborted and the car should return to the original lane l_c.

5.2.3 Merging

Merging is the act of changing lanes when forced to because the lane you are currently occupying ends, because it is an entrance ramp or an accident has occurred. It therefore differs from normal overtaking in that lane x ends soon.

Goal The goal of this action is to change the lane the car is currently occupying l_c to lane l_n where $l_c \neq l_n$ before the vehicle reaches the end of lane l_c.

Preconditions The same as in Section 5.2.2.

Action sequence The same as in Section 5.2.2.

Abort conditions When the safety conditions in lane l_n are violated the lane change should be aborted and the car should return to lane l_c. The speed of the car might need to be adjusted since lane l_c might end soon.

5.2.4 Overtaking

Overtaking is a task where a faster vehicle passes a slower vehicle that is in front of him by first switching lanes, then passing the vehicle and finally switching back to the original lane.

Goal The goal of overtaking is to pass the car in front of you on lane l_c while using lane l_o to the left of lane l_c to pass that car.

Preconditions The speed of the car in front is lower than the speed you would like to drive at.

There is a lane l_o directly to your left which you are allowed to use.

And any conditions from Section 5.2.2.

Action sequence

1. Execute a lane change as indicated in Section 5.2.2.
2. Increase speed to an amount that allows you to overtake the car.
3. When you have overtaken the car execute a lane change as indicated in Section 5.2.2 to lane l_c. Unless there is another car on lane l_c that you want to overtake during the same task, then perform the second action again.

Abort conditions The conditions in Section 5.2.2 apply here as well. Also if the car you want to overtake suddenly decides to speed up and you are unable to safely/legally overtake it then you should abort and try to change lanes back to lane l_c.

Chapter 6

Conceptual Design

In this section the design of our simulation is conceptually explored. Section 6.1 introduces the separation between the driver and the vehicle. Section 6.2 discusses automating the driver and finally Section 6.3 introduces how human behavior can be simulated using an automated driver.

6.1 Separating the driver and the vehicle

When a vehicle is moving from one place to another we can in essence distinguish two separate entities that together accomplish this goal. Namely the physical vehicle itself which uses the laws of motion to propel itself towards its destination. And a person, the driver of the vehicle, or in other words the entity that controls the vehicle's actuators. The driver is responsible for short-term and long-term planning. In other words the driver is responsible for deciding when to overtake or perform other maneuvers and is also responsible for planning the route to the actual destination.

6.2 Automation of the driver

In the future we would like roads to be used more efficiently and also more safely. Since computers have proven that they can do certain tasks better than us it seems to be a good idea to look into vehicles where the driver is being replaced by a computer program. These automated vehicles would be able to drive safely, using a set of advanced sensors to gather knowledge about the surrounding area. Combining this with navigational aids such as GPS the system should be able to drive long distances with ease. In fact such a system has already been proven and used to drive over 160.000 miles in total[20].

6.3 Introducing human behavior in automated drivers

To fully understand the interaction between humans and computers we simulate a scenario in which human driven and automated vehicles share the road. This is done by adding personality to the automated vehicle. To accomplish this we

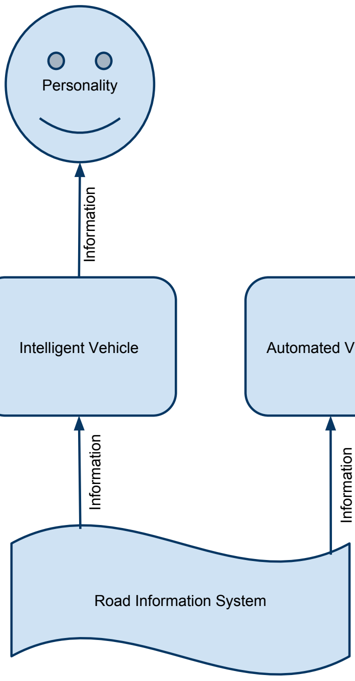

Figure 6.1: Drawing of the different drivers in the simulation.

further separate what constitutes an automated vehicle and putting part of it in a component called the Intelligent Vehicle (IV). The IV has the ability to safely execute driving maneuvers such as changing lane and keeping distance based on the personality of the driver. The higher level functionalities such as route planning and initiating lane changes are left to the automated driver or which in our case is analogous to the human driver.

The IV is connected to an Information Structure (IS) which allows it to sense the world around it. The information it needs to receive is very minimal: only direct environment information such as who is around me can be enough to make the vehicle drive safely. More sensor information can be delivered to the automated driver so that look ahead can be improved, allowing better/efficient choices to be made (for instance dynamic routing). Also the human driver has his or her own sensors (mostly eyes and ears) which can be used to gather more information to be input into the IV.

Chapter 7

System Architecture

In Chapter 6 we conceptually explored a design for our agents. In this chapter we take a more practical look at how we can put those concepts to use in an architecture that should be used for agents in our simulation. We base the architecture on MASQ which is introduced in Section 7.1. Section 7.2 introduces the models we use to create intelligent/human behavior in traffic. Finally Section 7.3 explains the architecture in which we build our agents.

7.1 Multi-Agent Systems based on Quadrants (MASQ)

MASQ helps to give structure to complex environment such as the ones in the traffic simulation. In this section MASQ is described, a more detailed description can be found in[7, 34].

MASQ is based, as the name might suggest, on a 4-quadrant framework consisting of two axes. The horizontal axis is one that differentiates between interior and exterior, which distinguishes between opinions and facts: *facts* are things we observe in the environment and the *opinion* is the mental representation of such an environment and anything else that deals with interpretation. The vertical axis distinguishes between the individual and the collective. The individualistic quadrants deal with single components in the system while the collective quadrants deal with the relationship between all of these components.

This results in four quadrants that are each mapped to a basic construct to describe complex systems such as our highway scenario or any other complex social system as seen in Figure 7.1.

Mind (Interior-Individual) The *mind* is the component that makes decisions on what the agents wants to do (intentions) but not on what it will do. The inputs for this component are mostly the percepts received from the environment. In our simulation the role of mind is played by a GOAL agent.

Body/Object (Exterior-Individual) An *object* is a tangible representation of things in the world. If the object is a physical representation of an agent with a mind it is also referred to as a *body.* Objects obey the rules of the

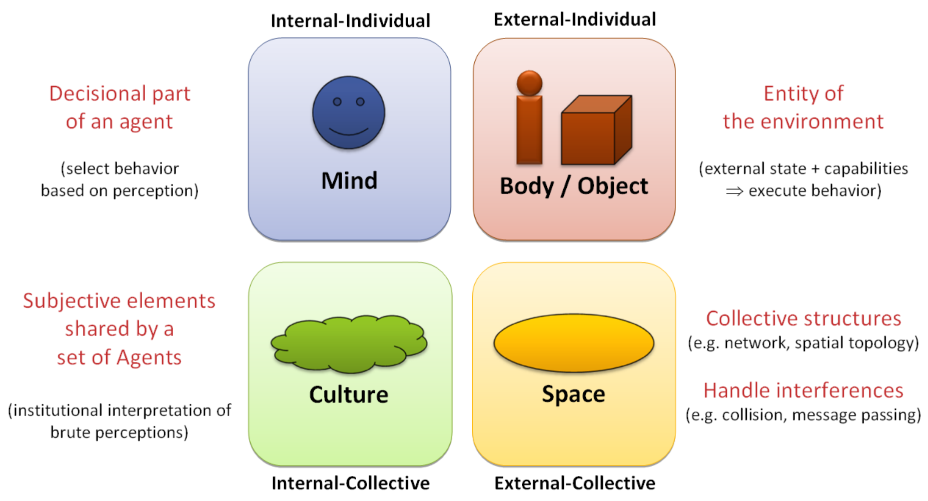
Internal-Individual
External-Individual
Decisional part of an agent
(select behavior based on perception)
Mind
Body / Object
Entity of the environment
(external state + capabilities ⇒ execute behavior)
Subjective elements shared by a set of Agents
(institutional interpretation of brute perceptions)
Culture
Space
Collective structures
(e.g. network, spatial topology)
Handle interferences
(e.g. collision, message passing)
Internal-Collective
External-Collective

Figure 7.1: Summary of the four quadrants of MASQ, based on a presentation from V. Dignum [1].

environment, in our simulation for instance the physics of a vehicle: apply the brakes and the vehicle gradually slows down. When a mind is connected to a body it can perform actions in the environment; the body is used as executor of these actions.

Space (Exterior-Collective) The environment in which bodies are located are described by *spaces*. Spaces can be physical but they can also represent non-physical concepts such as a being used to display the social relationships between different bodies. In our highway scenario we currently only use one space to model the physical environment.

Culture (Interior-Collective) An environment is mainly described by objects and spaces. However the interpretation of these spaces differs per group of individuals, these kinds of interpretations are referred to as culture. In MASQ *culture* is used to represent facts such as dependencies, norms and values. For instance in our simulation we could have a driver who thinks it is very normal to drive over the speed limit, while another group of drivers thinks it is very annoying.

We chose to use MASQ for the highway simulation so that we can give the environment more structure and to investigate what the added value of MASQ would be. MASQ is evaluated in Chapter 9.

7.2 Modeling intelligent behavior in traffic

Instead of defining behavioral rules or state machines to simulate the behavior of traffic, several mathematical models have been developed to mimic the human driver. Two of them are introduced in this section. Section 7.2.1 describes a mathematical model for dealing with the longitudinal control of a car and section 7.2.2 introduces a mathematical model for steering based on two salient points.

7.2.1 Intelligent/Human Driver Models for longitudinal control

If we take a step back and take a look at what is behind some of the simulations out there we find that most of them try to model following behavior using mathematical models. A summary of a set of popular models is given in [38]. These models all relate to one ability, namely following the car in front.

One of these models that has been built for intelligent vehicles is called the Intelligent Driver Model (IDM). It has been developed by Treiber[36]. Formula 7.1 defines the longitudinal acceleration of the vehicle as per the IDM.

$$\alpha_{idm} = a \cdot [1 - (\frac{v}{v_0})^4 - (\frac{x^*}{x})^2] \tag{7.1}$$

$$x^* = x_0 + vT + \frac{v\Delta v}{2\sqrt{ab}} \tag{7.2}$$

In these formulas v is the velocity of the vehicle, α is the acceleration, x is the distance to the vehicle in front and Δv is the velocity difference between the two cars. Formula 7.2 defines x^* which is the desired distance to the predecessor.

The model contains five parameters which have a sensible definition when it comes to car following.

- a is the maximum possible acceleration of the vehicle
- b is the maximum comfortable deceleration of the vehicle. This parameter has to be positive otherwise a negative root occurs in the formula. If the situation dictates a quick stop the model can suggest a deceleration of the vehicle which is higher then b.
- v_0 is the desired velocity of the driver.
- x_0 is the minimum distance, in meters, to the vehicle in front. If the car in front stops x_0 is the distance between the two cars.
- T is the desired time headway in seconds. It defines the time it takes for the vehicle to cover the distance to the vehicle in front at its current velocity. Human drivers are suggested to have a headway of at least two seconds[22]. See also Equation 5.1.

Six years later the Human Driver Model (HDM) was developed, again by Treiber, as a meta model on top of IDM[37]. The reasoning behind this was that the IDM, although good for intelligent vehicles, didn't take into account human factors such as temporal and spatial anticipation errors and reaction time. Estimation errors are modeled using a Wiener process[8], which produces pseudo-random estimations. The reaction time is modeled by taking values not at the current time, but by taking values at a certain reaction time T_0 ago. This is done for the value of x the distance between the two vehicles, the velocity of the vehicle v and the difference in velocity between the two vehicles Δv. If the value at exactly T_0 is not available a linear interpolation is used to estimate it.

7.2.2 Two-Point steering model for lateral control

In 2004 Salvucci et al. introduced a two-point control model for lateral movement, or steering[31]. It was later used to model driver behavior in a cognitive architecture[30], which makes it a candidate for lateral control in an agent-based system. The steering model is based on two salient points, i.e. points that tend to get a lot of attention when it comes to human driving.

The first point is called the *near point*, it represents how close the vehicle is to the center of the lane. It is defined as the point in the center of the lane in front of the center of the car at a set distance of $10m$.

The second point is called the *far point*, it represents the curvature of the upcoming road and helps the driver to anticipate the actions required to stay close to the center of the road. The definition of the far point depends on three different cases:

1. On a straight empty road it is the point in the center of the road up to 2 seconds THW distance from the car.
2. If there is a car nearby that we are following then the center of that car is used.
3. Otherwise, the tangent point of an upcoming curve (the point on the inside of the lane).

The near and far points provide overlapping information that allows for adjustment to stay in the lane center using the near point and for compensation at a near-future position using the far point. The model uses formula 7.3 to determine the adjustment needed to the steering angle $\Delta\varphi$.

$$\Delta\varphi = k_{far}\Delta\theta_{far} + k_{near}\Delta\theta_{near} + k_I\theta_{near}\Delta t \tag{7.3}$$

θ_{near} and θ_{far} are the angles to the two points respectively. Their difference from the last iteration is denoted by the Δ symbol. Δt is the elapsed time since the last iteration. The formula aims at imposing three constraints. First a stable far point ($\Delta\theta_{far} = 0$), secondly a stable near point $\Delta\theta_{near} = 0$ and finally the near point at the center of the lane $\theta_{near} = 0$.

k_{far}, k_{near} and k_I are constants that determine the weight of each component, these can be dynamically adjusted[30]. Salvucci's research also showed that for lane changing behavior that mimics that of a human the constants $k_{far} = 20$, $k_{near} = 9$ and $k_I = 6$ can be used.

7.3 System Architecture

The goal of our agent is to drive a car in a highway setting. Since both safety and reaction time are important our architecture should definitely have a reactive component that keeps the safety conditions in check. However since driving involves getting from point A to point B it is also important that some higher form of planning is involved. The most obvious candidate to support these functionalities comes in the form of a hybrid architecture discussed in Section 2.1.4.2. The role of the reactive component is played by multiple finite state machines, while the deliberative component is implemented by GOAL.

The agent is layered in the following manner from top to bottom:

Cooperation The cooperation layer is responsible for actions that require communication with other cars. These are the actions surrounding platooning. This layer can be implemented in GOAL but is not part of this thesis.

Decision The Decision layer is responsible for the day-to-day decision making process. These include decisions about actions such as overtaking or merging. This layer is implemented in GOAL.

Motion The motion layer is responsible for simple actions such as the throttle and the lane change. The actions are separated in a latitudinal and a longitudinal part. These two parts are implemented in Java in the form of a lateral and longitudinal controller that follows the models from Section 7.2.

Safety The safety layer is responsible for vehicle safety. It activates when the safety of the driver is in danger. This layer is implemented in Java together with the motion layer. This layer however can not be directly controlled by the agent but is influenced by their profiles.

More information about the implementation of these layers can be found in Chapter 8.

Chapter 8

Implementation

This chapter details the implementation of the control architecture for the automated vehicle in a highway situation. The implementation consists out of two controllers, one for the longitudinal and one for lateral actions. These controllers are implemented to enable a vehicle to successfully navigate the highway, however when to initiate switching lanes or when to overtake is decided on a higher level by the agent, and not by these controllers. Overtaking and merging are modeled as higher layer processes which uses the power of the longitudinal and lateral controller. These controllers are based upon work of Habib[21]. First Section 8.1 introduces the different objects that are simulated. The longitudinal controller is described in Section 8.2 and the lateral controller is described in Section 8.3. Section 8.4 then explains the different parameters of the driver profile. Section 8.5 details the implementation of sensing in the simulation. Finally Section 8.6 introduces the interface that EIS exposes such that agents can control the vehicles.

8.1 Physics objects and the road

This section discusses the physical objects and the road surface that has been implemented in the simulation. Since we are not interested in height differences the simulation takes place in a two-dimensional plane.

Vehicles Vehicles are implemented as simple objects with a two-dimensional speed, and have a rectangular bounding box for detection of crashes. For sake of simplicity crashes immediately halt the vehicles involved. Controlling the vehicle is done by setting the acceleration for the lateral and longitudinal movement. The acceleration is automatically applied during every timestep in the simulation.

Road The road is not implemented as a physical object but as a virtual surface that can determine on which of its lanes a vehicle currently is. The vehicles in turn use this information to determine whether they are driving properly in between

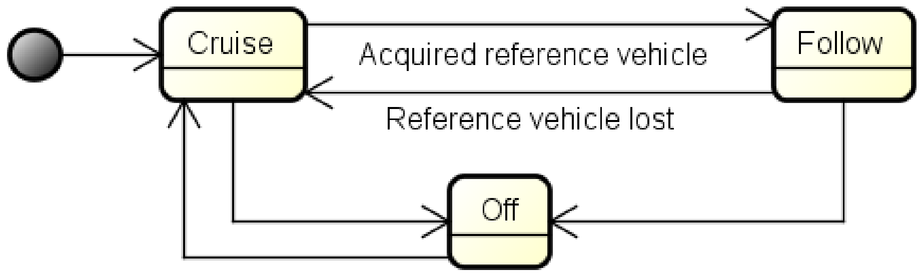

Figure 8.1: State machine for longitudinal control

8.2 Longitudinal Controller

The longitudinal controller is responsible for controlling the speed of the vehicle according to the model in Section 7.2.1. It can be modeled as a state machine with three states, see Figure 8.1. An important role is played by the so-called reference vehicle. This is the vehicle directly in front of us which we use as a reference point to figure out whether to accelerate or decelerate.

The controller has three different states, namely:

Cruise In this state there is no reference vehicle defined and the vehicle accelerates to its desired cruising speed. It changes to the FOLLOW state if a reference vehicle has been acquired. It changes to the OFF state if a higher level is taking control over longitudinal actions.

Follow Being in the follow state makes the vehicle adjust the speed of the vehicle to reach the desired distance to the reference vehicle. The state changes to CRUISE if the reference vehicle is lost or to OFF if a higher level is taking control over longitudinal actions.

Off The OFF state means that the longitudinal controller doess not influence the speed of the vehicle. A system on the higher level is directly controlling the speed.

Since one of the goals of the simulation is to model different drivers, certain parameters influence the decisions made by this controller. These parameters are the cruising speed, following distance, safe time headway and acceleration profile. The latter influences how a driver uses the acceleration paddles to change the speed of the vehicle.

8.3 Lateral Controller

The lateral controller is responsible for controlling the steering wheel of the vehicle according to the steering model in Section 7.2.2. It is responsible for successfully and safely making the vehicle switch lanes. It consists out of a state machine with three states, see Figure 8.2.

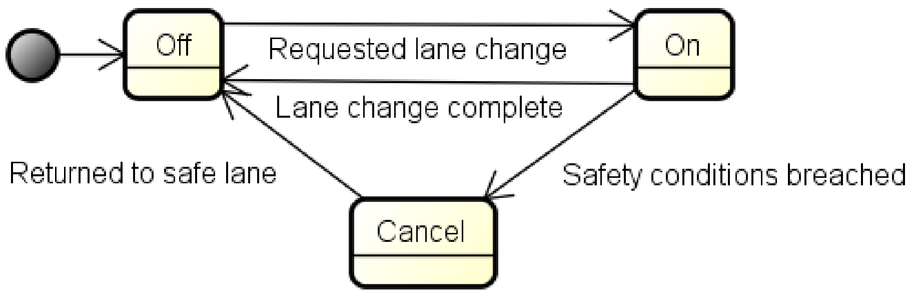

Figure 8.2: State machine for lateral control

Off In this state the lateral controller tries to stay on the middle of the lane. When it receives a request for a lane change the state changes to ON.

On The lateral controller steers the vehicle in the direction of the lane that is being changed to. If a safety condition fails, such as a vehicle in front suddenly starts to break, then the controller changes to the *cancel* state. If the lane change is successfully completed the controller changes back to the OFF state.

Cancel In this state the controller returns the vehicle to a safe lane, which is very likely the lane it came from. Once it is back on a lane the controller switches to the OFF state. This means that if the driver would still like to switch lanes this has to be indicated again.

Important when performing lateral actions is the so called gap acceptance. The gap acceptance of a lane determines if the driver believes a lane change is safely possible. For modeling gap acceptance we use the minimum distance vehicle distance from the driver profile. A second parameter indicates the kind of steering profile the driver uses, i.e. aggressive versus calm steering.

8.4 Driver Profile

Because we gave direct control over the throttle and steering to the environment we need a different way of introducing differences between drivers. To this end we have implemented a driver profile. The driver profile is fed to the controllers, which use it to make their adjustments.

The driver profile contains the following variables:

Target Velocity The velocity the vehicle attempts to accelerate to when the road is clear.

Minimum Vehicle Distance Minimum distance that will be kept to the vehicle in front.

Target Time Headway This is the targeted outcome of Equation 5.1, it measures the time it takes for the vehicle to hit the vehicle in front if it would be standing still.

Maximum follow distance The maximum distance to the vehicle in front. If the vehicle goes beyond this distance the lateral controller transitions away from the FOLLOW state.

Steering Speed The speed/aggressiveness of steering is determined by the value of this parameter. The value of this parameter is multiplied by k_{far} from the steering model in Section 7.2.2. A value > 1 means faster steering, while a value that is < 1 is slower than the average human as defined by Salvucci in [30].

8.5 Sensing

Sensing is implemented by modeling sensors that are separated from the rest of the vehicle. These sensors push their latest readings to all the components that have expressed interest in the information.

The sensor data consists out of three types of information, all of which are measured without any sensor deviation. In future work, sensor deviation can be added for more realism.

Vehicle information The information that is available about the vehicle is the current speed, both lateral and longitudinal as well as the position of the vehicle in the world. The vehicle is also aware about which lanes it is currently occupying.

Road topology The agent receives information about the topology of the road, i.e. how many lanes there are and what the speed limit is.

Vehicle neighborhood information The data about the vehicles nearby that is given to the agent is information about which vehicle is in front and which vehicle is to the back of the agent on each of the lanes. They'll know how fast these vehicles are going and what their relative position is to the agent itself.

8.6 EIS

In this section the percepts and actions that are made available through EIS are discussed. Section 8.6.1 lists the percepts and section 8.6.2 lists the actions.

8.6.1 Percept Definitions

Percepts are defined as the input the agent receives at any given moment in time[29]. Intelligent agents use these percepts to decide which action they are going to take. These percepts are a one-to-one translation of the sensor data that the agents receive.

Percept: vehicle(*id*)

id The identifier of the vehicle.

Description The identifier in this percept belongs to a vehicle. This identifier can be used to query for specific information in other percepts, such as velocity and distance.

Percept: position(*id*, *x*, *y*)

id The identifier of the vehicle.

x The x coordinate of the position as a double.

y The y coordinate of the position as a double.

Description This percept describes the position of the rear of the vehicle. In practice this might be done with road side tracking equipment or GPS.

Percept: velocity(*id*, *v*)

id The identifier of the vehicle.

v The longitudinal velocity of the vehicle in m/s as a double.

Description This percept contains the longitudinal velocity of the vehicle in m/s.

Percept: lateralVelocity(*id*, *v*)

id The identifier of the vehicle.

v The lateral velocity of the vehicle in m/s as a double.

Description This percept contains the lateral velocity of the vehicle in m/s.

Percept: lane(*id*, *lane*)

id The identifier of the vehicle.

lane The identifier of the lane.

Description This percept indicates the lane on which the vehicle is driving. This is included because the details of the road's topology are hidden to the agent, e.g. the agent does not know how wide a lane is. This percept may occur more than once for a single vehicle if it is switching between lanes.

Percept: distance(*id*, *d*)

id The identifier of the vehicle.

d Distance to the vehicle in meters.

Description This percept indicates the distance to the other vehicle when they are projected on a line (i.e. all lanes collapse into one). This is the head to tail distance between the two vehicles.

Percept: neighbors(*lane*, *front*, *rear*)

lane The identifier of the lane.

front A list with ids of vehicles that are in front, starting with the one that is closest.

rear A list with ids of vehicles that are in the rear, starting with one that is closest.

Description This percept contains which vehicles are to the front and rear of your vehicle for a specific lane.

Percept: numberOfLanes(n)

n The identifier of the lane.

Description This percept contains the number of lanes on the road.

Percept: targetVelocity(*v*)

v The target velocity of the vehicle in m/s.

Description The speed at which the vehicle would like to drive on the open road.

Percept: minimumVehicleDistance(*d*)

d The minimum distance to keep to other vehicles in meters.

Description The minimum distance the vehicle wants to keep from other vehicles in front of it.

Percept: targetTimeHeadway(*t*)

t The new time head-way in seconds.

Description The time head-way (see Equation 5.1) that the vehicle tries to keep.

Percept: maximumFollowDistance(*d*)

d The maximum distance in meters.

Description The the maximum distance at which the vehicle follows the vehicle in front.

Percept: steeringSpeed(x)

x The steering speed relative to the average human. > 1 is faster and < 1 is slower than the average human.

Description The steering speed or agression of steering relative to that of the average human. This is used by the lateral controller for steering.

8.6.2 Action definitions

Actions define the way agents can manipulate the state of the environment. Intelligent agents base the actions they choose on the percepts they receive. All actions defined for agents that control vehicles influence either the profile of the driver or the lateral/longitudinal controllers. The speed and steering of the vehicle are not directly influenced so that agents are protected from most dangers of driving.

Action: changeToLane(*lane*)

lane The identifier of the lane to change to.

Description Attempts to change the vehicle's lane. The action fails if the lane is not deemed safe enough according to the agent's own profile parameters. If the lane is safe enough the lateral controller changes the sidewards velocity of the vehicle so that it can change lane.

Action: setTargetVelocity(*v*)

v The target velocity of the vehicle in m/s.

Description Allows the agent to change the speed of the target speed of the vehicle. The vehicle attempts to maintain this speed as long as the situation is safe according to the profile parameters of the agent.

Action: setMinimumVehicleDistance(*d*)

d The minimum distance to keep to other vehicles in meters.

Description Changes the minimum distance the agent wants to keep from other vehicles in front of it.

Action: setTargetTimeHeadway(*t*)

t The new time headway in seconds.

Description Changes the time head-way (see Equation 5.1) to the time the agent wants to obey to.

Action: setMaximumFollowDistance(*d*)

d The maximum distance in meters.

Description Changes the maximum distance at which the agent follows the vehicle in front. If the vehicle in front goes further away then the agent's vehicle accelerates to the target velocity.

Action: setSteeringSpeed(x)

x The steering speed relative to the average human. > 1 is faster and < 1 is slower than the average human.

Description Sets the steering speed relative to that of the average human.

Chapter 9

Evaluation

In this chapter we evaluate the highway simulation, keeping in mind that the goal of this thesis is to provide guidelines for interfaces between simulation environments and multi-agent frameworks. Section 9.1 evaluates MASQ to see if it adds value for the highway simulation. In Section 9.2 we evaluate the interface exposed to GOAL agents through EIS.

9.1 MASQ

MASQ is developed to help create a deeper and more complete understanding of the interaction processes in multi-agent systems [34]. In its current state this understanding only seems to add value at the conceptual level. When it comes to putting MASQ into practice its benefits seem to disappear.

First, the use of the MASQ framework for the highway simulation did not add new functionality to the simulation. The only MASQ feature used is the unique identifier it assigns to every object, which is very useful for representing these objects in EIS. However this can also be easily achieved without MASQ.

Second, the abstraction provided by the MASQ quadrants proved to be redundant when combined with Repast. For instance, the Space quadrant is already represented in Repast, and wrapping the Repast code in a MASQ Space does not add essential functionality.

Third, while the separation between Body and Mind in MASQ is conceptually interesting, most modelers already make this separation when entities are to be controlled based on some logic, be it in an APL or another language. In that case a modeler already applies a structure to publish the capabilities of an entity, mimicking what in MASQ terminology is a Body and letting the logic play the role of the Mind.

If we compare MASQ to EIS2Java we see that, while MASQ defines a *Body* as having capabilities that the *Mind* can execute, their current implementation does not expose capabilities to the mind. In that sense MASQ is different from EIS2Java where the capabilities are explicitly defined and made available to the agent. In theory MASQ could use the same approach as EIS2Java and use the annotation approach to automatically expose the capabilities of a body. A mind would then only be allowed to execute capabilities that are annotated. MASQ would be doing the same as EIS2Java although their goals are different. MASQ tries to structure the environment to create more insight into the interaction processes of multi-agent systems, while EIS2Java aims to publish the capabilities of controllable entities for use in multi-agent systems.

To conclude, MASQ has a plausible theoretical foundation, but when using Repast its current implementation does not offer more than some common terminology during design. This may be resolved in a future version by implementing the Culture quadrant and tying the Body and Mind closer together by means of interfaces that really enable the programmer to give Bodies explicit capabilities.

The advantage of our prototype is that we can now take other environments that were written using MASQ and couple them to GOAL, such as crisis management and warehouse logistics [1, 13].

9.2 Environment Interface

Making abstractions to allow an agent to drive a car is not easy. There are different layers of abstraction one could propose. To take two extremes, one could have agents calculate the speed of the vehicle by giving them access to how many times the wheel rotates and what its diameter is, on the other end they could just get the speed directly from the sensors.

Is the wheel rotations method relevant to the agent when it comes to driving a vehicle? We think the answer to this question is no, for the same reason why dials on vehicles give us an approximation of the speed rather then the number of wheel rotations: namely to reduce the amount of cognitive reasoning required to drive. Just like humans, agents cannot perform an infinite number of tasks as the same time. When designing an interface it is therefore important to first determine what the agent aims to achieve, and then select a simple and effective representation the information it needs to realize its goal. This helps to reduce the reasoning power the agent requires. It appears the current interface does provide an appropriate representation: they are able to drive the vehicle and complete the tasks stated in Section 5.2.

Using the same line of reasoning we decided not to give the agent direct control over the speed of the car. Instead we allow the agent to suggest a speed, which the vehicle then accelerates to, *if* there is enough room in front of the vehicle to do so safely. This also goes for steering: the agent simply suggests that it wants to switch to another lane and the steering is automatically handled by the controller in the vehicle.

From a software engineering point of view we note an important aspect when it comes to connecting environments to an APL. Since APLs usually do not address objects by pointers like OOP languages, it is important that they are assigned a unique ID. In our case this was handled by MASQ.

In a scalability test where agents drove from one end of the highway to the other without overtaking, we were able to simulate 150 vehicles when controlled by GOAL and 200 when they are purely controlled in Java. This can likely be improved upon by modeling sensors in a more efficient way. The reason why GOAL is less efficient actually seems to reside with EIS. Every so often the *getAllPercepts* method gets called for every entity. This leads to a large amount of generated percepts, each which has to be handled by GOAL. More research is required to confirm this is the case and to figure out where optimizations can be made.

Part IV

Blocks World For Teams Simulation

Chapter 10

Blocks World for Teams Environment

The Blocks World for Teams (BW4T) environment was introduced by Johnson et al.[19] as a testbed for planning problems in a multi-agent environment. The idea behind the environment is based on a classical AI problem called Blocks World, in which the world consists of blocks resting on a table and it is the agent's task to rearrange them in stacks in a particular order.

The multi-agent version of Blocks World works slightly different. The agents are placed in a world consisting of rooms and hallways connecting these rooms. The rooms contain colored. The goal is for the agents to bring these blocks to a drop zone in a specific order, based on the colors of the blocks. The agents can not see what is outside the room they are currently in, nor can they see other agents, forcing them to communicate about their observations and actions for optimal team performance.

We can classify the BW4T environment according to the five dimensions in Section 2.1.3. The environment can be considered partially observable since robots are not able to observe other robots nor are they able to observe anything outside of the room they are currently in. The environment is deterministic and only depends on the actions taken by the agents that are present. This makes prediction possible in this environment provided the agent has enough information. The environment is sequential, since actions can not be divided into episodes although such an episodic approach can be taken to complete the mission, i.e. find and transport blocks one at a time. The environment is dynamic since the state of the environment can be changed while the agent is thinking about its next action. Finally the environment is dynamic in nature since speed, position and location are all measured on a continuous scale representing a continuous world.

Chapter 11

Specification

This chapter specifies the different types of objects that needed to be implemented to simulate the BW4T environment in Repast.

Blocks Blocks are objects that can be in two states. Either they are held by a robot or they are placed somewhere in the world. Apart from this state the other property that a block has is color. The color is important since blocks need to be delivered in a certain colored order.

Rooms Rooms are rectangular locations in the world that can be surrounded by a wall with small doorways as an opening. The rooms can have blocks placed inside of them on initialization of the environment.

Dropzone The dropzone is basically a special room in the world where no blocks are placed initially. The dropzone has an ordered list of colors of blocks which need to placed in the zone to complete the mission.

Robots Robots are objects that can move around the world. Therefore they always have a position and a velocity. They also have the ability to observe the world around them and hold/drop a block.

These four different objects are the only objects that exist in the BW4T environment. They work together to create challenging tasks that requires the robots to work together to solve them efficiently.

Chapter 12

Implementation

This chapter discusses the implementation details of the BW4T environment and details the interface which agents use to interact with the environment. Section 12.1 describes the modeling of the environment. Section 12.2 details the EIS interface developed for the agents.

12.1 Modeling the environment

The environment is implemented in two dimensional space since we are not interested in height as a third dimension.

BoundedMoveableObject The objects in the Java implementation of the BW4T environment are all based on the *BoundedMoveableObject* class. This class gives each object a unique integer for identification. It also offers basic functionality such as adding/removing the object from the world, putting objects in another position, changing the size of objects and doing intersection checks.

Robot The robot is modeled as an entity shaped like a square. It has a current location and a location it wants to move to. The robot is capable of picking up blocks that are a certain distance from it, much like a robot would have an arm. The robot also has a human-readable name for visualization purposes. The movement of the robot is done by moving at a set speed towards the location it wants to move to, slow-down or speed-up has not been added but would require little effort to implement if the need arises.

Room The rooms are modeled as rectangular areas. A robot is considered inside a room if it is entirely inside. Rooms are capable of having blocks placed inside of them however the room has no knowledge of this. In the implementation each room also has a color for visualization purposes.

The *DropZone* is a special kind of room which keeps track of the sequence of colors the blocks dropped in the zone need to adhere to to complete the mission. A drop zone is capable of removing blocks from the world once a block has been dropped in it. Every block is removed, even if it doesn't have the right color. In such cases however, the color sequence does not advance.

Code Snippet 12.1 Example of a map for the BW4T environment.

```
100 100
R 45 45 10 10 B B R W
D 5 5 10 10 B R
E Bender 20 25
J WallE 20 20
```

Blocks Blocks are modeled as rectangular objects, they have a color not only for visualization purposes but also important for the sequence in which they need to be delivered to a dropzone. Blocks know if they are being held by a Robot.

MapLoader The MapLoader is an important class that exposes a static method to load an environment from a text file. The an example of the text file that defines a map is give in Snippet 12.1.

The first line in the map file specifies the width and the height of the map. Every next line indicates an object in the world, the first character of that line indicates which type of object we are dealing with.

The R indicates a room at a certain (x, y) coordinate followed by the width and height and finally $[0, \infty\rangle$ colors of the blocks that are put in this room upon initialization. The colors are coded as single characters and only a limited number is supported at this time.

The D indicates a dropzone, only one of which may be present in a map file. It has the same specification as a room with one difference, the block color indicators indicate the sequence of blocks of the dropzone.

The E stands for EIS entity and spawns a Robot that can be controlled through EIS. The E is followed by the human readable name (without spaces) and the starting location of the robot in (x, y) format.

The J stands for a Robot that is controlled by Java itself, a programmer can built an implementation for it. The syntax is the same to that of a robot that can be controlled through EIS.

12.2 EIS

This section details the percepts and actions that are made available through EIS to control a Robot entity in the BW4T environment.

12.2.1 Percepts

This section details the percepts a robot can receive during the simulation.

Percept: at(*id*, *x*, *y*)

id The identifier of the object that is at the location.

x The x coordinate of the location.

y The y coordinate of the location.

Description Percept that tells the location of the object with the given id. This can be used to find out where to navigate to but also to find out where the robot itself is.

Percept: block(*id*)

id The identifier of the block.

Description Percept identifying that a certain id belongs to a block.

Percept: room(*id*)

id The identifier of the room.

Description Percept identifying that a certain id belongs to a room.

Percept: dropZone(*id*)

id The identifier of the drop zone.

Description Percept that couples an identifier to a drop zone.

Percept: robot(*id*)

id The identifier of a robot.

Description Percept identifying that a certain id belongs to a robot.

Percept: color(*id*, *c*)

id The identifier of the block.

c The character that represents the color of the block.

Description This percept gives information about which color a certain block has.

Percept: sequence(*seq*)

seq List of color identifiers.

Description This percept contains the color order in which blocks need to be delivered to the dropzone. Once a block with the right color has been delivered, the sequence is updated.

Percept: holding(id)

id The identifier of the block that is being held by the robot.

Description The percept gives information about which block the robot is currently holding.

12.2.2 Actions

This section introduces the actions a robot can perform. Although they are rather basic, they grant the robot the power it needs to perform the tasks needed.

Action: goTo(*x*, *y*)

x The x coordinate of the position to move to.

y The y coordinate of the position to move to.

Description Instructs the robot to move to the given coordinates.

Action pickUp(*id*)

id The identifier of the block to pick up.

Description The robot tries to pickup the block. The action fails if the block is not in range of the robot's arm.

Action putDown

Description The robot drops the block it is holding at the current position. If the current position is in the dropzone the block is removed from the environment and the sequence may progress, if and only if the block is of the right color.

Chapter 13

Evaluation

The goal in the BW4T domain is to facilitate agent-agent and human-agent teamwork. This means that it is important that the interface enables agents to cooperatively engage in problem solving. To facilitate this focus on teamwork we decided to apply abstractions to aspects of the environment where the agent does not need to be in full control. One of these aspects is navigation. We believe that control over navigation in the BW4T environment is best left to be implemented in Repast for two reasons. First, because APLs are less efficient at solving path planning problems. Second, because changes in the setup of the environment, such as adding walls, would severely influence how navigation works. By hiding the navigation behind an abstract action we can ensure that the agents code stays relatively simple.

Our thinking about the navigation abstraction resulted in three questions which should be asked when faced with the problem of designing the agent interface.

First, what are the decisions an agent needs to make? This question is important because agents can be written in APLs, which are developed to facilitate creating agents that are strong in logic-based complex decision making and reasoning. It is therefore important that the agent has enough information and power to make the decisions that are important for the interaction under research. In case of the BW4T environment the main decision an agent is faced with is which block to pickup or which room to visit. We built the interface around this premise.

Second, is there any functionality for which the strengths of the environment simulation language can be leveraged? Environment languages are usually better suited for 3D visualizations and navigational issues since they are more efficient in math-like calculations. Keeping this kind of functionality in the environment can increase the efficiency of the simulation and the clarity of the agent code.

Third, is there any functionality in the environment that might be subject to change? Functionality that internally might change over time is best abstracted away from for the agent to ensure that the agents do not need to be rewritten when the environment is modified. This is why we decided to make an abstraction for navigation, such that the path planning could easily be replaced without any changes to the agents.

This is the second environment we developed using EIS2Java and the connection between EIS and Repast. During development we found that we generated a lot of percepts with the same name, such as the block percept for each block. We therefore decided to change the *AsPercept* annotation of EIS2Java to include the option to have a method return multiple percepts instead of a single one.

This environment was also built from the ground up and helped in constructing the manual in Appendix C, to explain how environments in Repast can be connected to GOAL.

Finally, although it may seem obvious, it bears repeating that as a rule of thumb, one should aim to give the agent enough information and power to act upon the environment in such a way that is in line with the goals of the simulation, without cognitively overloading the agent. Start simple and make more complex information available only when required.

Part V

Conclusions, Recommendations and Future work

Chapter 14

Conclusions

In this chapter we answer the research question defined in Chapter 1: *"Which design guidelines provide effective rules for designing interfaces that connect simulation environments to agent programming languages?"* Our guidelines are given in Section 14.1. Section 14.2 highlights the implementation of the connection between EIS and Repast.

14.1 Interface Design Guidelines

Looking back at the evaluation of the highway environment in Chapter 9 and the BW4T environment in Chapter 13 we can construct the following three guidelines for designing environment interfaces.

Introduce unique names for objects Unique names for objects are important because most APLs do not have a concept of a memory pointer to address objects. The unique names will become the identifiers for the objects and can be used in actions and percepts. A modeler should keep this in mind when writing the environment.

Keep computationally expensive functionality in the environment Computationally expensive functionality, such as navigation, should remain encased in the environment. The reasons for this are two-fold.

First, APLs are less efficient in such tasks than OOP languages, since they are usually built to perform theorem proving and other forms of logical inference [33]. Other types of languages such as those that are OOP are more efficient at implementing navigation algorithms such as A^* and Dijkstra.

Second, keeping expensive functionality outside the agents allows them to focus on higher level tasks that are in need of a logical decision structure, something which APLs are good at.

Design the interface according to the simulation goals Environments for multi-agent systems are usually built to have agents complete certain tasks. In creating the interface it is therefore important that agents are able to focus their reasoning cycles on accomplishing these tasks successfully. A good way of doing this is to make abstractions for tasks that are of less importance. In the BW4T environment we applied this to navigation, where the agent is only able to state that it wants to go somewhere, how exactly this is achieved is abstracted away and done by the environment.

Looking at the guidelines we can conclude the choice for GOAL as APL is actually arbitrary. By using a generic layer, like EIS, we could have easily swapped out GOAL with 2APL, Jadex or Jason. It is however the case that some of these guidelines may become obsolete when an APL based on Java is used. Unique names for objects may not be necessary if the APL supports pointers. One could also argue that computationally expensive functionality can then be put in the agent. However this might not be the best solution since every agent would usually want to make use of the same implementation. It might therefore be better to just have it exist in the the simulation framework rather than in the agent.

14.2 Connection to Repast

In Chapter 4 we developed the tools and system to connect GOAL to Repast. In doing so we have used advanced features of Java such as reflection and annotations to create EIS2Java. EIS2Java makes it easier to define and expose controllable entities through EIS, reducing the work needed to make an environment EIS compatible. It also lowers the maintenance costs, since percepts and actions are defined as single method which makes it easier to test and change. Finally EIS2Java reduces the overhead of translating between Java objects and EIS's *Interface Immediate Language* by introducing a compact interface to separate translation from action/percept handling and to by making translations portable across projects.

The connection to Repast allows researchers to take any environment in Repast and expose it through EIS so that it may be influenced by a multi-agent systems. None of the current functionality of Repast was compromised to make this connection and can still be used by the environments.

Chapter 15

Future Work

This chapter discusses several possible improvements to the products developed during this thesis and also those that we have worked with. Most of the suggestions in this chapter are focussed on making it easier to connect environments to agent platforms. Some suggestions for future thesis projects are also included.

EIS2Java

Some percepts in environments should be sent only once, or when their value changes. Currently there is no support for that in EIS2Java. Research needs to be done to determine the use-cases of these special types of percepts and whether it is possible to capture these use-cases in EIS2Java. If it is deemed necessary this can be achieved by implementing a third annotation in EIS2Java or by extending the *asPercept* annotation.

We also seek to include EIS2Java as part of the EIS distrubtion, and we will discuss this with the authors.

Connection between GOAL and Repast

To make the connection between GOAL and Repast to use, the connection should be implemented using RMI, or any other method that would allow Repast to run in a separate VM from GOAL. To achieve this the current code needs to be refactored and RMI bootstrap code to publish the EIS interface needs to be developed. This should however be a straightforward task once EIS has become easier to use (see the future work section on EIS).

Highway Environment

The Highway environment is capable of simulating one-way traffic on a straight stretch of highway, but entrance/exit ramps are unsupported at this time. They can be added, together with a more flexible and efficient sensor design that is easier to maintain than the current implementation. This also helps to improve the scalability of the simulation, although GOAL seems to be partially causing the overhead as well. To achieve a better sensor implementation it would need to make efficient use of caching data that is requested multiple times during the same tick and make sure that data that is not needed is not built.

BW4T Environment

To make the BW4T environment fully comply with that specified by Johnson et. al. the rooms need to have walls [19]. This prevents agents from entering a room from every direction, making it necessary that the the navigation is updated to solve the arbitrary routing of robots through the environment. The navigation of robots can be implemented by using A^* or another grid-based navigation approach.

Additionally, a user interface needs to be built on top of the current implementation, to make agent-human interaction possible. The exact approach required to make this possible together with Repast is unknown to us.

Environment Programming with EIS

In this thesis EIS has been found effective in allowing environments to be coupled to agent platforms. However, there are a few issues which need to be solved to make it easier and more pleasant for a programmer to work with EIS. These issues have to do with the definition of the EIS interface in Java.

First, and foremost the current interface is trying to accomplish too many tasks at once. As has been noted in Section 4.1, the environment management code has been rolled into the interface that also deals with percepts, actions and agent/entity registration. In programmer's terms one could call the current interface a "kitchen sink"; the place where all the magic happens. This large array of responsibilities of the single interface makes it difficult to create a proper separation between the tasks that belong purely to the environment, like entity registration, and tasks that GOAL should handle, e.g. agent coupling. The overloaded responsibility of the interface causes problems when the environment is run remotely, and is the main reason why Repast could not be run over RMI. We propose that much of the current functionality be removed from the definition of the EIS interface and a more slim version of EIS form the basis for different flavors. These can be created using programming principles such as plugins, hooks and facades. For example, Repast can implement the core functionality of EIS while GOAL wraps this functionality and extends it to support the event handlers it needs.

Second, the default implementation of EIS, called *EIDefaultImpl*, uses concurrent hash maps for some of its data. However, the documentation of the interface does not mention concurrent access at all. Since agent in a multi-agent systems are very likely to be multi-threaded, this matter should be investigated and clarified in the documentation to ensure that concurrency issues do not occur.

Finally, EIS could use a general code clean up. Over time EIS grown in size and function and lessons have been learned about the strengths and weaknesses of EIS. Therefore, it should be possible to create a better version of EIS. Preferably starting from scratch code wise, the theoretical foundation seems solid. Also, the Javadoc needs to be cleaned up and made useful for programmers in several places, especially when it comes to the assumptions methods may make when they are implemented. Other aspects are important too, such as the reduction of code duplication and removing the use of old Java classes, such as *Vector*. Finally, the definition of exceptions that EIS can throw need to be cleaned up and make use of proper inheritance. Enumerations rather than hard coded integers can be used to indicate the type of an exception.

Suggestions for followup projects

Since this thesis is made in cooperation with the university it is likely that there are more students that are looking for similar projects. We use this opportunity to make a few project suggestions based on the work we have done and where we think challenges still remain.

Clear to us is that the BW4T environment has potential and it would be nice to see it further developed into an actual research project that focuses on robot teamwork and perhaps even human-agent interaction.

Secondly, working extensively with GOAL has made us realize that they both could use some usability attention. One of the interesting ideas that should be tried with GOAL is to integrate it with a proper IDE such as Eclipse instead of GOAL maintaining their own implementation.

Bibliography

[1] H. Aldewereld, J. Tranier, F. Dignum, and V. Dignum. Agent-based crisis management. *Collaborative Agents-Research and Development*, pages 31–43, 2011.

[2] Brahms Authors. Agent iSolutions. `http://www.agentisolutions.com/`.

[3] T.M. Behrens, J. Dix, and K.V. Hindriks. Towards an environment interface standard for agentoriented programming. Technical report, Citeseer, 2009.

[4] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Information and Software Technology*, 50(1-2):10–21, 2008.

[5] F. Bellifemine, A. Poggi, and G. Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, pages 97–108. Citeseer, 1999.

[6] M. Bratman. *Intention, plans, and practical reason*, volume 250. Harvard University Press Cambridge, MA, 1987.

[7] V. Dignum, J. Tranier, and F. Dignum. Simulation of intermediation using rich cognitive agents. *Simulation Modelling Practice and Theory*, 18(10):1526–1536, 2010.

[8] R. Durrett. *Stochastic calculus: a practical introduction.* CRC, 1996.

[9] I.A. Ferguson and University of Cambridge. Computer Laboratory. *Touring Machines: An architecture for dynamic, rational, mobile agents.* Citeseer, 1992.

[10] ACL Fipa. Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00061/SC00061G.html (30.6. 2004).*

[11] M.P. Georgeff and A.L. Lansky. Reactive reasoning and planning. In *Proceedings of the sixth national conference on artificial intelligence (AAAI-87)*, pages 677–682. Seattle, WA, 1987.

[12] Arne Hankel. Files kosten transportbedrijven ruim 1 miljard euro. `http://www.elsevier.nl/web/Nieuws/Nederland/282996/Files-kosten-transportbedrijven-ruim-1-miljard-euro.htm`.

[13] M. Hiel, H. Aldewereld, and F. Dignum. Modeling warehouse logistics using agent organizations. *Collaborative Agents-Research and Development*, pages 14–30, 2011.

[14] K. Hindriks, F. de Boer, W. Van Der Hoek, and J.J. Meyer. Agent programming with declarative goals. *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 248–257, 2001.

[15] K. Hindriks, B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraayenbrink, W. Pasman, and L. de Rijk. Unreal goal bots. *Agents for Games and Simulations II*, pages 1–18, 2011.

[16] K.V. Hindriks. Programming rational agents in goal. `http://mmi.tudelft.nl/trac/goal/wiki`, 2011.

[17] K.V. Hindriks, F.S. De Boer, W. Van der Hoek, and J.J.C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[18] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[19] M. Johnson, C. Jonker, B. van Riemsdijk, P. Feltovich, and J. Bradshaw. Joint activity testbed: Blocks world for teams (bw4t). *Engineering Societies in the Agents World X*, pages 254–256, 2009.

[20] J. Markoff. Google cars drive themselves, in traffic. *The New York Times*, 10:A1, 2010.

[21] I.H. Muhammad. Multi Agent-Based Control Architecture in Intelligent Transporation System with Infrastructure-Based sensing . 2010.

[22] L. Neubert, L. Santen, A. Schadschneider, and M. Schreckenberg. Single-vehicle data of highway traffic: A statistical analysis. *Physical Review E*, 60(6):6480–6490, 1999.

[23] J. Ploeg, F.M. Hendriks, and N.J. Schouten. Towards nondestructive testing of pre-crash systems in a HIL setup. In *Intelligent Vehicles Symposium, 2008 IEEE*, pages 91–96. IEEE, 2008.

[24] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A bdi reasoning engine. *Multi-Agent Programming*, pages 149–174, 2005.

[25] S.F. Railsback, S.L. Lytinen, and S.K. Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609, 2006.

[26] A.S. Rao and M.P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[27] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in cartago. *Multi-Agent Programming:*, pages 259–288, 2009.

[28] G. Rimassa, D. Greenwood, and M.E. Kernland. The living systems technology suite: an autonomous middleware for autonomic computing. 2006.

[29] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach.* Prentice hall, 2009.

[30] D.D. Salvucci. Modeling driver behavior in a cognitive architecture. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 48(2):362, 2006.

[31] D.D. Salvucci and R. Gray. A two-point visual control model of steering. *Perception-London*, 33(10):1233–1248, 2004.

[32] M. Sierhuis, W.J. Clancey, and R.J.J. Hoof. Brahms an agent-oriented language for work practice simulation and multi-agent systems development. *Multi-Agent Programming:*, pages 73–117, 2009.

[33] L. Sterling. *The practice of Prolog.* The MIT Press, 1990.

[34] T. Stratulat, J. Ferber, and J. Tranier. Masq: towards an integral approach to interaction. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 813–820. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[35] L. Tesfatsion. Agent-based computational economics: A constructive approach to economic theory. *Handbook of computational economics*, 2:831–880, 2006.

[36] M. Treiber, A. Hennecke, and D. Helbing. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E*, 62(2):1805–1824, 2000.

[37] M. Treiber, A. Kesting, and D. Helbing. Delays, inaccuracies and anticipation in microscopic traffic models. *Physica A: Statistical Mechanics and its Applications*, 360(1):71–88, 2006.

[38] C. van Leeuwen. Driver Modeling and Lane Change Maneuver Prediction. *Order*, 501:3371.

[39] M. Wooldridge. *An introduction to MultiAgent Systems.* Wiley, 2009.

[40] M. Wooldridge and N. Jennings. Agent theories, architectures, and languages: a survey. *Intelligent agents*, pages 1–39, 1995.

[41] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152, 1995.

Appendix A

EIS2Java Implementation

This appendix shows the implementation of the methods that form the backbone of the EIS implementation that uses the EIS2Java annotations. These methods are called from methods that are specified in EIS. The moment an entity is added, the code in Snippet A.1 is executed to process the annotations. The *getAllPerceptsFromEntity* calls the code in Snippet A.2. When an action is performed the EIS method called *performEntityAction* is backed by the code in Snippet A.3.

Code Snippet A.1 Showing how annotations are processed.

```
/**
 * Processes and caches all annotations for the given
    class. If the class has
 * already been processed this method will do nothing.
 *
 * @param clazz The class to process.
 * @throws EntityException Thrown when the annotations
     are not used properly.
 */
private void processAnnotations(Class<?> clazz) throws
    EntityException {
  if (processedClasses.contains(clazz)) {
    // Already processed.
    return;
  }

  Map<String, Method> percepts = new HashMap<String,
     Method>();
  Map<String, Method> actions = new HashMap<String,
     Method>();
  for (Method method : clazz.getMethods()) {
    AsPercept asPercept = method.getAnnotation(
       AsPercept.class);
    if (asPercept != null) {
      String name = asPercept.name();
      percepts.put(name, method);
    }

    AsAction asAction = method.getAnnotation(AsAction.
       class);
    if (asAction != null) {
      String name = asAction.name() + "/" + method.
         getParameterTypes().length;
      actions.put(name, method);
    }
  }
  allPercepts.put(clazz, percepts);
  allActions.put(clazz, actions);
  processedClasses.add(clazz);
}
```

Code Snippet A.2 Simplified version of how percepts are generated.

```
/**
 * Creates new percepts by calling the given method on
     the entity.
 *
 * @param entity the entity to get the percept from.
 * @param perceptName the name of the percept.
 * @param method the method to invoke on the entity.
 */
private List<Percept> getPercepts(Object entity,
    String perceptName, Method method)
     throws PerceiveException {
  // Retrieve the object that is generated by the
      method.
  Object returnValue;
  try {
    returnValue = method.invoke(entity);
  } catch {...}

  // Figure out which Java objects to turn into
      percepts.
  List<Object> generatedJavaObjects;
  AsPercept annotation = method.getAnnotation(
      AsPercept.class);
  if (!annotation.multiplePercepts()) {
    generatedJavaObjects = new ArrayList<Object>(1);
    if (returnValue != null) {
      generatedJavaObjects.add(returnValue);
    }
  }

  // Generate percepts for each object that needs to
      be translated.
  List<Percept> percepts = new ArrayList<Percept>(
      generatedJavaObjects.size());
  for (Object javaObject : generatedJavaObjects) {
    Parameter[] parameters;
    try {
      parameters = Translator.getInstance().
          translate2Parameter(javaObject);
    } catch (TranslationException e) {
      throw new PerceiveException("Unable to translate
          percept " + perceptName, e);
    }
    percepts.add(new Percept(perceptName, parameters))
        ;
  }
  return percepts;
}
```

Code Snippet A.3 Simplified version of how actions are performed.

```java
/**
 * Performs the action method on the given method
    using the parameters. The
 * returned {@link Percept} will have the same name as
     the action.
 *
 * @param entity The entity to perform the method on.
 * @param method The method to invoke on the entity.
 * @param actionName The name of the action that is
    being performed.
 * @param parameters The parameters to the method (
    before translation).
 */
private Percept performAction(Object entity, Method
    method, String actionName,
    LinkedList<Parameter> parameters) throws
        ActException {
  Translator translator = Translator.getInstance();

  Class<?>[] parameterTypes = method.getParameterTypes
     ();

  Object[] arguments = new Object[parameters.size()];
  // Translate all parameters to method arguments.
  int i = 0;
  for (Parameter parameter : parameters) {
    try {
      arguments[i] = translator.translate2Java(
          parameter, parameterTypes[i]);
    } catch (TranslationException e) {
      throw new ActException(ActException.FAILURE, "
          Action " + actionName + " with parameters "
           + parameters + " failed to be translated", e
              );
    }
    i++;
  }

  Object returnValue;
  try {
    returnValue = method.invoke(entity, arguments);
  } catch {...}

  // Use the return value to generate a percerpt.
  ...
}
```

Appendix B

EIS2Java Manual

In this manual we go through the steps required to make an environment EIS compatible using EIS2Java.

B.1 Extend AbstractEnvironment

Create a class that extends the *AbstractEnvironment* that comes with EIS2Java. This is going to be the environment interface class that will be launched by GOAL to start your environment, we refer to this as the instance of EIS. Implement abstract methods according to the EIS documentation. Make sure that you are able to retrieve an instance of this class during runtime so that your environment may use it to register entities that can be controlled by agents, this can be achieved by passing it along to the constructor of your environment when EIS's *init()* method is called.

B.2 Create a class representing an entity

Every entity that an agent needs to control needs to be contained within a single class for EIS2Java to work. We will call this class *Entity* during the rest of this manual.

B.3 Define Percepts

Percepts can be defined by annotating methods in *Entity* with the *AsPercept* annotation. The object that the method returns will be translated to the Interface Immediate Language(IIL). Methods annotated with *AsPercept* must not take any arguments and have a non-void return type. The *AsPercept* annotation has one required attribute and one optional attribute.

name The percept that are generated by translating the object returned by the annotated method will have this name. This is a required attribute.

multiplePercepts If set to true then the annotated method must return a subclass of Collection, the collection will be iterated over and each entry in the collection will generate one percept. The default value is false.

Note that the return type must have an associated translator otherwise a *TranslationException* will be thrown when this percept is generated.

B.4 Define Actions

Actions can be defined by annotating methods in *Entity* with the *AsAction* annotation. Methods annotated may have an arbitrary amount of arguments and can have any return type (including void). If the annotated method has a non-void return type a percept with the same name as the action is generated upon completion containing the object returned by the annotated method. The *AsAction* annotation has one required attribute.

name The action must have a name so that it can be addressed by an agent through the EIS interface. This is a required attribute.

Note that the arguments must have a *Parameter2Java* translator and the return type must have an associated *Java2Parameter* translator otherwise a *TranslationException* will be thrown when the action is performed.

B.5 Register Entities

Entities that you want to have controlled by agents must be registered with the instance of *EIS*. An entity can be registered by calling the *registerEntity(name, entity)* method on the EIS instance. Make sure the name given to the entity is unique or otherwise an exception will be thrown.

B.6 Creating and using your own translators

If you want to use a type of object as a return value in any of the annotated method you must make sure that an appropriate *Java2Parameter* translator has been registered with the translator. This can be done by implementing the *Java2Parameter* interface that comes with EIS2Java and then calling *Translator.getInstance().registerJava2ParameterTranslator()* from anywhere before an entity is registered with the environment. Note that for objects that need to be translated to parameters the superclass tree will be traversed, meaning that you won't have to register a translator for each and every subclass, as long as you want them translated in the same way.

For the arguments of actions a *Parameter2Java* object needs to be registered with the translator in a similar fashion. These arguments require a translator registered with the exact same class as the argument.

B.7 All set

You should be all set now. Just follow the normal steps when combining an EIS environment with the APL of your choice.

Appendix C

How to use Repast with GOAL

This manual explains the process of how to combine your Repast environment with GOAL. Before starting we will assume that the user has completed the GOAL and Repast Simphony installation.

For GOAL and Repast to work together you must make sure you have followed the steps in Appendix B to make the entities EIS2Java compatible, this excludes implementing the *AbstractEnvironment* which can be the same for every Repast environment. This environment class that ships with this thesis is called *RepastEnvironment* and should be present inside a Jar in Repast's *lib* folder.

C.1 Enable GOAL to use Repast

Go into your GOAL installation folder and locate a folder called *modes*. Copy this folder to the root directory of your Repast project. This is done because GOAL has some problems with relative paths for files it requires after start up. If you don't copy this folder GOAL will still work, however there will be no syntax highlighting in the editor.

Copy *goal.bat* or *goal.sh* file to your Repast project. Open up the file and locate the command that starts the java VM. Add the following folders from the Repast Simphony Eclipse plugin folder to the classpath (-cp flag), *repast.simphony.runtime_2.0.0/bin* and *repast.simphony.runtime_2.0.0/lib/**. You should now be able to successfully start GOAL using this file.

C.2 Editing the MAS file

To enable GOAL to run a Repast the MAS file from GOAL, which specifies the location of the environment, needs to point to the location of the JAR containing the *RepastEnvironment*. You can use the initialization parameters in MAS to point to the location of the scenario folder, or you can use load function in the GUI of Repast to load a scenario. An example of the MAS file can be found in Snippet C.1.

Code Snippet C.1 GOAL MAS file for the default *RepastEnvironment.*

```
environment{
    % insert the RepastEnvironment reference on the
       next line.
    "lib/RepastEnvironment.jar" .
    % insert the initialization parameters and values
       on the next line.
    init [ scenario_location = "./HighwayTraffic.rs"]
       .
}
```

C.3 All set

You should now be able to control Repast/GOAL like you would in normal circumstances. Depending on how you structured the registering of entities you might need to press the play button in GOAL to make the agents do their job.

Appendix D

Example GOAL Agent

This appendix contains example code of a GOAL agent written for the BW4T environment. Note that this agent operates solely on his own, and that only the code for the reasoning cycle is shown.

Code Snippet D.1 GOAL agent for the BW4T environment.

```
main module{
  program{
    if bel(sequence([C|T])) then adopt(dropOff(C)).
    if goal(dropOff(C)), bel(sequence([X|T]), C \= X)
       then drop(dropOff(C)).

    % I'm holding a correct block, let's drop it off
    if goal(dropOff(C)), bel(robot(Me), holding(Block)
       , color(Block, C), dropZone(D), at(D, X, Y), at
       (Me, MyX, MyY), (X \= MyX ; Y \= MyY)) then
       goTo(X, Y).
    if goal(dropOff(C)), bel(robot(Me), holding(Block)
       , color(Block, C), dropZone(D), at(D, X, Y), at
       (Me, X, Y)) then putDown.

    % I don't know a block of the requested color, go
       to another room.
    if goal(dropOff(C)), bel(not(color(Block, C)),
       robot(Me), at(Me, MyX, MyY), room(R), at(R, X,
       Y), (X \= MyX ; Y \= MyY)) then goTo(X, Y).

    % I want to commit to picking up a block.
    if goal(dropOff(C)), bel(color(Block, C), at(Block
       , BlockX, BlockY), robot(Me)) then blockPicker(
       Me, Block).
  }
}

module blockPicker(Me,Block)[focus=new,exit=nogoals] {
  goals {
    holding(Block).
  }

  program {
     if bel(at(Block, X, Y), at(Me, MyX, MyY), (X \=
        MyX ; Y \= MyY)) then goTo(X, Y).
     if bel(at(Block, X, Y), at(Me, X, Y)) then pickUp
        (Block).
  }
}
```