Multi-Agent Systems in GOAL

Jannick Johnsen and Søren Jacobsen

Kongens Lyngby 2013
IMM-B.Sc.-2013-18

Technical University of Denmark
DTU Compute
Matematiktorvet, building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
compute@compute.dtu.dk
www.compute.dtu.dk B.Sc.-2013-18

Summary (English)

In this report we will document our efforts to develop the Xmas (cross platform Multi-Agent System) engine for designing MAS (Multi Agent System) environments and managing intelligent agents acting in them. As the engine is for designing environments, the agents are supposed to receive commands from and send percepts to a seperate agent programming language, which implements the artificial intelligence of the agents.

The primary goal of the project is to make the engine as general as possible so as to allow any desirable environment to be designed with it, while making it easy to extend individual components to suit the needs of specific types of MASs. The engine comes packaged with support for interfacing with EIS (Environment Interface Standard), and, by extension, the agent programming languages supported by it. A simple tile-based environment is also provided. The engine is designed with the model-view-controller (MVC) pattern, to allow clear seperation of components. To showcase and test our engine, we have created a reference implementation which uses the GOAL agent programming language to control agents.

We believe that the Xmas engine have achieved a high degree of generality, although this comes at the expense of features and functionality useful to many MASs, which have instead been delegated to extensions. The engine is best suited for designing, setting up and executing larger systems, as there is a lot of overhead involved. The engine as well as the example extensions runs on the major operating systems, including Linux, Windows and Mac OS.

Summary (Danish)

I denne rapport vil vi dokumentere vores bestræbelser på at udvikle Xmas (cross platform Multi-Agent System) maskinen, der er designet til at udvikle MAS (Multi-Agent System) miljøer og holde styr på intelligente agenter, der agerer heri. Da maskinen bruges til at lave miljøer, er det meningen at agenterne sender sanselige indtryk til og modtager kommandoer fra et separat agent-programmeringssprog, der implementerer agenternes kunstige intelligens.

Det primære mål med projektet er at gøre maskinen så generel som muligt, således at ethvert ønskeligt miljø kan udvikles med den, og samtidig gøre det nemt at udvide individuelle komponenter, så de passer til specifikke MAS-typer. Maskinen kommer med indbygget understøttelse for sammenkobling med EIS (Environment Interface Standard), og dermed også for sammenkokbling med de af EIS understøttede agent-programmeringssprog. Et simpelt flise-baseret miljø følger også med maskinen. Maskinen er designet ud fra model-view-controller udviklingsmønsteret, hvilket muliggør en klar opdeling af de forskellige komponenter. For at teste og fremvise vores maskine har vi udviklet en referenceimplementation der bruger agent-programmeringssproget GOAL til at styre dens agenter.

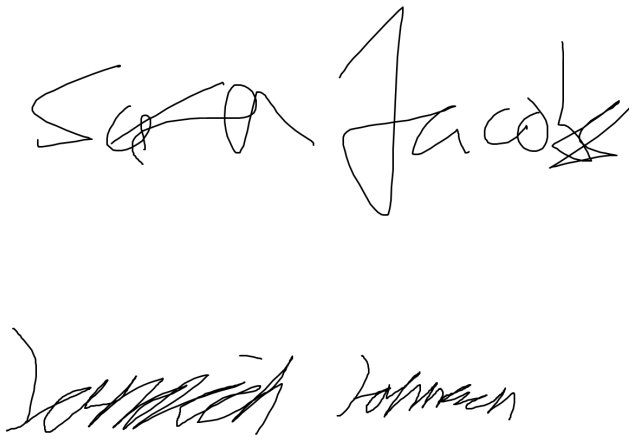
Vi mener at Xmas i høj grad er generel, hvilket har betydet at flere funktionaliteter er blevet rykket fra Xmas-kernen ud til udvidelsespakker. Xmas er bedst egnet til større systemer, da der skal en del kode til at designe, opsætte og eksekvere sytemer med den. Både Xmas og udvidelsespakkerne kører på de dominerende operativsystemer, Linux, Windows og Mac OS inkluderet.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an B.Sc. in Informatics.

Although multi-agent systems is a field with a lot of research, there are not many tools aimed at designing or setting up environments for these systems. This problem is compounded by the fact that there are many capable agent programming languages (APLs). In this project we will explain and showcase the design of our Xmas Engine designed specifically to communicate with different APLs in the same environment. Furthermore the engine is also designed to be used for any type of environment. Along with the engine we provide a reference implemention (with agent programs written in GOAL) showing the abilities of the Xmas Engine.

Lyngby, 01-July-2013



Jannick Johnsen and Søren Jacobsen

Acknowledgements

For the guidance given to us in shaping our project, and for the continuous assistance throughout, we would like to thank our supervisor Jørgen Villadsen. We would also like to thank Koen V. Hindriks for providing us the source code of the environments that comes with GOAL, so that we could quickly understand how to develop our own environment.

Contents

Chapter 1

Introduction

Background

MASs (multi-agent systems) is an important research topic in the field of AI (artificial intelligence) as they consist of several intelligent agents interacting and cooperating to reach a specific goal. This lends itself well to distributed computing, following the recent (since the turn of the millenium) trend of designing processors with more cores rather than better ones. Multi-Agent Systems have also been used to simulate naturally occurring systems, where several autonomous agents interact.

There have been a lot of research on the topic of MASs, as well as related ones such as distributed artificial intelligence (DAI). There are several APLs (Agent Programming Languages) tailored to MAS development; they are called MASPLs (Multi-Agent System Programming Languages).

Motivation

While there exists many multi-agent programming languages, there are few tools for constructing environments for the agents to behave in, which also allows for

easy graphical representation.

There are many complications when developing multi agent systems, our goal with this project was to lessen one of these by designing an engine with the specific purpose to develop multi agent environments. What these environments can be is left to the developer, however almost everything in the engine we propose is modular and interchangeable, ensuring that all types of multi agent environments are possible.

What the types of projects can be is manyfold but here are some possible examples:

Agent comparison software There are many different languages in which it is possible to write agent programs; some are specifically designed for it, others are powerful enough to accommodate the possibility of agent programming. Our engine is designed with support for multiple languages at once which makes this engine a suitable candidate for designing a comparator program.

For instance, if two groups wanted to test their agent programs against each other, this engine would make it possible for them to easily design a world in which this test could occur.

Agent testing/Simulation software Testing agent software can be complicated. Being able to rapidly create an environment and visualize it can be important to a MAS project, as it ensure basic mistakes are ironed out before larger scale implementation.

Agent teaching tools Teaching agent languages can be tough without proper exercises; however, the time spent on designing these exercises can prove too exhausting for the teacher to develop. Using our engine the teacher can rapidly design the world he had in mind for his exercise instead of designing every integral part of the multi agent system himself. This is because our engine provides all the basic features of a multi agent system, so that the time can be spent more productively on designing how a given exercise should play out, showcasing the problem the students are supposed to deal with.

Computer games In practice most computer games are just multi agent programs where one of the agents is controlled by the player. Our engine should make it fairly easy by setting up framework for creating rules inside a given

world and ensure that the agents of the world follow said rules, thus defining a game which the engine would be capable of running.

Goals

The engine which we propose must reach a list of goals in order for us to deem the project succesful:

Generality: By this, we mean that components developed with our engine should be *reusable* in the sense that they should be able to be used in other projects. For examle, the logic for connecting to a specific APL should be implamentable in other projects. Furthermore, the engine itself should be multi- applicable; it should be able to be used to construct any kind of scenario, and interface with different APLs.

Easy to use: It should be relatively easy to construct a complicated scenario, and even easier to construct a simple one.

Cross platform compatibilty: The engine should be executable on as many platforms as possible, at least the three major operating systems (Linux, Mac and Windows).

Overview of the Report

In this report we will document our efforts to design an engine as described above. The report is structured as follows:

Theory outlining the design patterns used in the engine along with explanation of MAS.

Reference Implementation providing an introduction to how a final product of the engine might look, along with agent programs connected to the engine.

System Features describing the features of the engine and how it can be used.

Implementation containing an examination of the actual implementation of the engine, along with our considerations of the choices taken.

Testing describes how the engine was tested.

Results and Comparisons evaluates whether the goals we had set for the engine was reached, and compares it to similar works.

Chapter 2

Theory

2.1 Multi-Agent Systems

2.1.1 Agents and Environments in Artificial Intelligence

In artificial intelligence, an agent is something that can perform actions in and (partially or fully) perceive the state of the environment it is situated in. As an example, consider an agent tasked with finding the shortest route between two nodes in a weighted, undirected graph, with the limitation that it can only see the nodes that have an edge to the one it is standing in, an example of the graph can be seen on fig. 2.1. We will now use this example to describe what [RNC+96] calls a *task environment*, consisting of definitions for the performance measure for the agent, the environment it acts in, the actions it can take, and its facilities for perception:

Environment The environment is a model of the world the agent acts in. In our example, it is described as a graph. The environment may also contain artifacts for agents to interact with, such as a packages to pick up or obstacles to navigate around.

Actions This denotes what actions the agent can take to change the state of

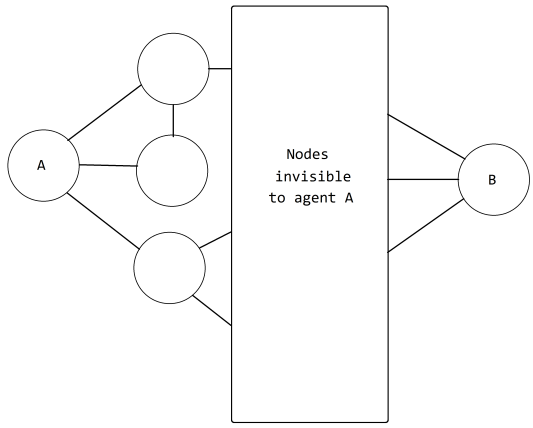


Figure 2.1: What an agent can see of a graph where it tries to move from node A to node B.

the environment or itself. In the example, the agent would have a *move* action, allowing it to move to an adjacent, connected node.

Percepts If the agent is to make intelligent decisions, it must be able to perceive the current state of the world – that is, itself and the environment. Such a fragment of information that the agent have sensed is called a percept. In the running example, the agent can perceive the nodes immediately connected to the one it is standing on, as well as the edges to those nodes.

Performance For an agent to be as efficient as possible, it is useful to have a performance measure describing how well the agent is executing the task at hand. In the provided example, the performance measure could be defined in terms of the number of actions taken per unique node visited, giving the agent an idea of the amount of redundancy in its pathfinding.

While the task environment can be used to succinctly describe the properties of the world, it says nothing of the logic that the agent applies to perform tasks in the world. This is left in the hands of an *agent program*, which is responsible for processing the percepts and choosing actions for an agent. In general, an agent program receives percepts and chooses an appropriate action based on the knowledge available to it in an aptly named *percept-action cycle*. This knowledge may just be the current percepts, or it may be all the percepts retreived so far. When choosing an action, it may take into consideration how different actions would affect the world, and how much closer performing the action would bring the agent to its goal. Agents with such capable agent programs are called *utility-based agents* in [RNC+96].

2.1.2 Multi-Agent Systems

While there is no strict, universal definition of what constitutes a multi-agent system, the following seems to represents the simplest consensus: *In a multi-agent system, several intelligent agents act and interact more or less autonomously in an environment.* The interacting of agents may be of any character, the important part is that each agent can be aware of the others and affect their execution directly or indirectly. Here, a direct effect means changing another agents state, eg. by moving it into another postition or decreasing its health. An indirect effect could be communicating with the agent, suggesting another execution path. For instance, a system in which agents compete for ressources and hinder other agents' progress is a multi-agent system, as is a system in which the agents work together towards a common goal.

While the former may be useful in simulating certain systems, the latter is more interesting in software design, as such an approach could conceivably lead to more decentralized problem solving. A mixture of the two approaches can also be used, as in the Multi-Agent Programming Contest[1], where teams of cooperating agents compete for points. In this case, the goal for each team is to develop the best strategy, where performance is measured by competitiveness.

In addition to the above, limiting the agents' knowledge of the state of the world is a desirable characteristic of a MAS; otherwise, there would be no need for the agents to communicate, and they could just as well be subroutines of a single agent [PL05]. It could still be modelled as a MAS, of course, but not a very interisting one. Additionally, the case could be made that the less agents know of the world, the less information they have to process when searching for an action to execute, thus reducing their computational load. On the other hand, less information can cause the agent to follow a suboptimal execution path, causing a trade-off between computation time and optimality.

One of the strong points of multi-agent systems is that it consists of several pieces of software running more or less autonomously. As mentioned above, this allows for developing decentralized systems, where agent programs runs on different threads or servers. In the context of distributed systems, less dependency on a central intelligence is of course preferred.

Furthermore, MASs are useful when simulating naturally occuring systems wherein several "agents" interact with each other An example of this could be a group of animals around a watering hole, where some are prey and some are predators.

[1] `multiagentcontest.org`

2.1.3 The GOAL Agent Programming Language

In this section we will outline the GOAL language based on [Hin09].

Several APLs (Agent Programming Languages) have been developed to suit the defining characters of agents and their interaction with environments as we have described them above. In this section, we will focus on the relatively new GOAL APL, which can be written using the prolog logic programming language.

In GOAL, code is segmented into sections describing:

- What it knows
- What it wants to achieve
- What it can do
- How it handles new information (percepts) from the environment
- The actual logic for choosing an appropriate action to execute

The first two points in the list will be explained below.

Mental State

GOAL provides the notion of *mental state* of an agent, which describes what the agent knows and what it aims to achieve. Specifically, it consists of the following three components:

Knowledge describes what the engine knows to be universally true. This information is completely static; it is something the agent is "born" with, and can not be changed. In other words, this describes the rules and constants of the system.

Beliefs are facts the agent deduces during its execution, using its knowledge. Beliefs can be updated at runtime by using the `insert(`φ`)` and `delete(`φ`)` commands, where φ is a belief. The `bel(`φ`)` operation can be used to ascertain whether the agent believes that φ holds.

Goals are what the agent strives to accomplish. These can be dynamically updated along the way to acommodate for a changing world. This is done with the `adopt(`φ`)` and `drop(`φ`)` commands, where φ is a goal. `goal(`φ`)`

checks whether φ is currently a goal of the agent. If a goal have been achieved – that is, if the agent's current beliefs and knowledge satifies a goal – it is automatically removed from the `goals` collection, as the agent would otherwise keep trying to accomplish it.

The information in the mental state is stored as logical statements in prolog. The operations mentioned above for querying and modifying the mental state thus takes a prolog statement as input.

Acting and Perceiving

When a GOAL program is running, it executes the following steps, in order:

1. Receive percepts
2. Update the goals and beliefs of the agent if needed
3. Choose an appropriate action to execute

This is repeated in a cycle.

The processing of new percepts mentioned in point #2 is handled in the `event module` of the agent program. Here, all new percepts in the current cycle can be inspected, and the mental state of the agent can be updated. If, for example, the agent perceives that it is in a different location than in the previous cycle, this module can be used to change its beliefs accordingly. If the world have changed drastically, the agent may also choose to drop goals that can no longer be achieved, and/or adopt goals that seems more fruitful to pursue. As such, the processing of percepts is the primary place to change the mental state of the agent.

The choosing of an action is where the agent decides – based on its mental state – which execution path it should take. The actions themselves are provided in an `action specification` section of the program, where each action denotes pre- and postconditions of the action. That is, what must hold for the action to be executed and what effect it will have on the environment. An action is not considered for execution if its precondition does not hold. If it does hold and the action is taken, the logical statements in the postcondition is inserted into the belief base.

When an action have been chosen in GOAL, it is only *set* to be executed, that is, GOAL requests that it be executed. It might be that the program managing

the agent in the environment sees fit to not execute it, or that the action fails (if the agent tries to move into a wall, for example). In that case, GOAL only knows about the failure of the action if it is somehow obvious from the next set of percepts it receives. In light of this, postconditions on actions should only be used when it absolutely certain that what the postcondition specifies is true, as it may otherwise insert flawed information into the agents beliefs.

2.1.4 The Environment Interface Standard

Several agent programming languages – including GOAL – is only concerned with the agent logic of a MAS. To provide a world in which these agents can function, they need to be connected to a program providing an environment.

EIS (Environment Interface Standard) [BHD11] is a Java framework, which can be used to design environments and connect them to agent programming languages. As such, it does not assume much about the implementation of the environments or the agents inhabiting it.

It provides `entities` which can function as the bodies of agent programs, and means for receiveing commands and returning percepts to the connected agents.When using EIS, the environment designer can handle actions sent by the agent programs with the `performEntityAction` method to define exactly how they affect the world the designer have constructed. Percepts can be requested explicitly through the `getAllPerceptsFromEntity` method (this is how GOAL gets its percepts from EIS) or as notifications, for APLs that support it.

The main point of EIS is to provide a standard (hence the name) for developing MASs, such that environments designed with this standard can easily be interfaced with different APLs. As such, EIS comes pre-loaded with bindings for several common APLs such as GOAL and Jason. As part of this standard, the IILang (Interface Immediate Language) abstract syntax tree have been developed. It can be used to easily and unambiguously define actions and percepts consisting of identifiers, numerals, representations of functions over identifiers and numerals, and lists of identifiers, numerals and functions. These IILang objects can be created as native Java objects and easily parsed to – in the case of GOAL – prolog statements, and vice versa.

In conclusion, it is important to note that GOAL and EIS are supposed to be two components of a multi-agent system. GOAL is not supposed to maintain an environment, and EIS is not very well suited for implementing agent logic.

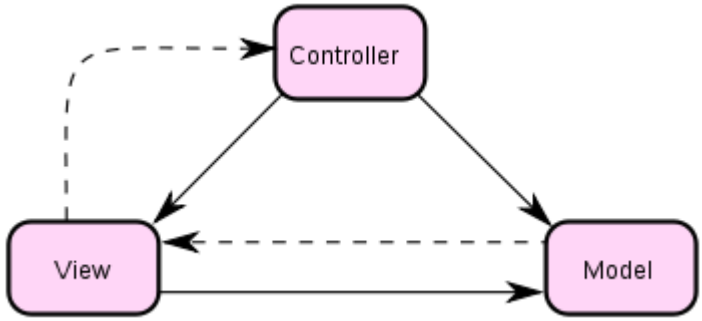


Figure 2.2: This image shows how the three components are connected to each other; the full arrows indicate that a component has complete knowledge of the component it is pointing to. A dashed arrow indicates that the component the dashed arrow is pointing to is listening to the component the arrow is originating from. (Image taken from `www.htmlgoodies.com/img/2010/11/mvc.png`)

2.2 Model View Controller

To ensure that code is correctly decoupled from one another, strict design patterns are necessary, as these allow developing complex projects without losing sight of the entire project. If no pattern is followed, code can easily become so entangled that later development might prove impossible. This section will cover all the rules and ideas behind the MVC (Model-View-Controller) design pattern. [Wikc] provides a short introduction to MVC, although complete comprehension of the design pattern requires experience using it. Hence, this section will be based on our prior experiences with MVC.

The MVC pattern principle is that programs that can be interacted with by a user can be split into three different components. These components are as follows:

- Model – Core of the program
- View – Visualization of the program
- Controller – Manager of state changes to the core of the program

2.2.1 Model

The model is the core of the program. It is why the program functions as it does, it contains all the data, and it is here all business logic is located. The model should have no knowledge of either the view or the controller; by not knowing either the program is ensured to not be tainted by their influence.

While it may not know of the view or the controller, it is paramount that the model is built to optimally transfer information concerning its current state. That means that providing a way for other components to listen on the model is very welcome. This gives the model a way to publish its state when it has been altered. What this does for the model is, that in case the model state has been altered, it will have a way to provide the information of the state.

If such features are not built into the model, it would require the component changing the state to inform of the state changes, in case of a MVC design. The component changing state is the controller and as such the controller would have both the duty of changing the state of the model and maintaining the view. This is generally a case of a badly designed model and can be completely avoided if the model simply has the ability to inform its listeners(such as a view) of any changes.

2.2.2 View

The view is a way to visualize what is currently occurring inside the model by visualizing it to the user. A view may take many shapes depending on the model. If, for instance, the model is a program processing data on a server, then the view could take the form of a logging console. Or if the model was a computer game then the view would be the graphic representation of the game. Additionally, a model can have several views, each displaying information in a different manner. For example, the computer game mentioned above could also feature a view printing debugging information to a console while the game was running. Generally, a view should only have knowledge of the model and not the controller. The idea is that if the view can see all model data then interaction with the controller should not be necessary.

When designing a view there are some common pitfalls that can be avoided with careful design. First off, the view is what it is named: a view. This means that it should never do any state changes to the model. If getting hold of data means that the model must change its state to accommodate this, then the model is poorly made and should be changed. However, a view is allowed to change its

own state without involving either the controller or model. To fully understand what is meant by this, consider the following example:

Assume you have a menu bar as depicted in fig. 2.3.

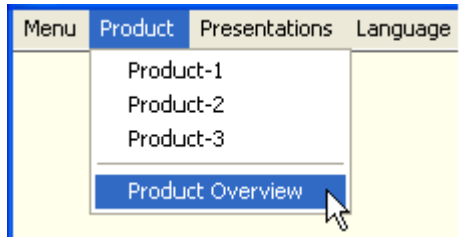


Figure 2.3: A standard menu bar.

To open a menu, the user need to drag the mouse and click on one menu he or she wishes to open. Many would see this as a task of the controller. This is not the case, however, since the changes done are only performed on the view's own state and not the model of the program.

2.2.3 Controller

The controller is the link between the model and the user. By convention, all changes the user wishes to perform on the model should be done through the controller. Like the view, it can take many shapes, like an object that transforms input from the mouse into changes to the model, or an object controlling how a network data stream effects on the model.

In a well-designed program the controller should never have to interact with the view. However, this can be practically impossible on larger projects unless they are carefully planned, and as such the controller by convention is allowed to know of both the view and the model.

A common mistake when designing the controller is to mistake the unit which the controller gets input from as the actual controller. In many cases, the keyboard is the device from which input is transformed into state changes to the model. However, that does not mean that the controller should be the only unit interacting with the keyboard. Going back to the example used to understand the view, the reason why the controller should not deal with opening a menu bar on the view is because the controller is not responsible of the state of the view. The controller is only responsible for the state of the model; the

only case in which it is allowed for the controller to interfere with the view is in the case that the model was unsuccessful in properly informing about its state change caused by the controller. In this case it is okay for the controller to call the view and ask it to adjust itself.

The reasoning for why the controller is normally mistaken to be responsible for handling changes to the state of the view is because it is mistakenly thought of as a controller for the entire program and not the model, a view may contain its own controller which should not be mistaken from the other controller. To fully understand this, imagine that the view in itself also contains a MVC inside itself (see fig. 2.4).

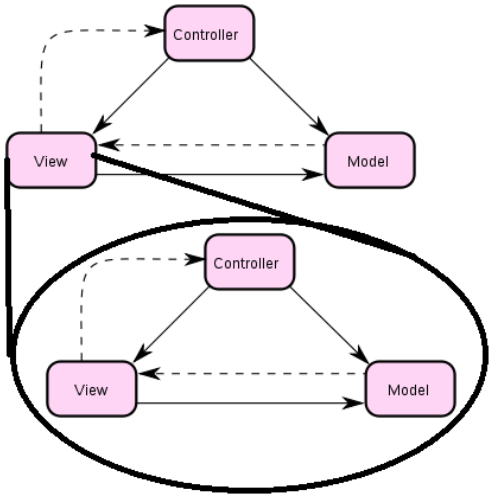


Figure 2.4: A view with a MVC inside of it.

The model of a view (such as a menu bar) would contain data about the names of the menus and it would be responsible for ordering and accessing information as to what each menu contains. Its view would be that of a drawing board responsible for properly drawing the menu bars. Luckily, most views are simple, so one does not need to make an entire MVC design, but for graphical user interfaces used in most operating system it is very important to understand that a view can be an entire MVC setup in itself. This is why most operating system comes with libraries to easily design GUI.

2.3 Factory Design Pattern

Removing flexibility from a program is normally considered a poor design decision. One way to do that however would be by tying object creation to the business logic of the program. This section will show why this combining of object creation and business logic actually removes flexibility in the program. Furthermore it will show how to solve this problem using the Factory Design Pattern. We refer to [Wika] for an explanation of the subject.

Why object creation logic should be removed from business logic

In all OOP[2] languages you have the ability to create new objects to be used in the program. When instantiating an object in a function, you inadvertently tie a specific object type to that function. This means that the function can never be used for other purposes than to work with that type of object. Thus if an identical function was needed but for another version of that object you would have to copy the function redundantly, greatly increasing the chance of errors in the code as the same code was written twice.

To give an example of this, imagine you have a Printer that prints text on sheet of paper, the pseudo code for such a printer would look like this:

```
Class Printer
    Method PrintPaper takes Message returns Paper
        Paper = new A4Sheet()
        Paper.PrintText(Message)
        Return Paper
    endMethod
endClass
```

As we can see in this example, `A4Sheet` is an implementation of the object type `Paper`, however as we have mixed object creation with business logic, we are forced to specify the exact type of paper that our printer produces. This means that if we wanted to make the printer able to print multiple types of paper, we would have to make new functions copying the functionality of `PrintPaper` we would have to make `PrintPaperA3`, `PrintPaperA5`, etc. As is evident, this is very redundant and increases code complexity. Not only that, but if years later

[2] Object Oriented Programming

someone invented a new type of paper, the printer would have to be completely changed, since the class was locked to specific paper types on the business logic level.

Solving the problem of mixing object creation with business logic

As we have shown, there are many problems involved when business logic contains object creation logic. In order to solve this problem it is necessary to separate the two. This can be done in multiple ways. Consider our previous printer example. In this case, instead of having the method `PrintPaper` take only a message, it could also take paper as part of its arguments. This way we would avoid having to create the Paper as part of printing it. However, this would only move the problem from the `PrintPaper` method to the business logic that is calling it. It also changes the overall functionality of the method which was to produce paper with messages on them and instead makes it so it doesn't produce the paper and only prints on paper given to it. Using the factory design pattern we can avoid both problems while at the same time separate the object creation away from the `PrintPaper` method. We do this by giving the `Printer` class a factory which we will name `PaperFactory`.

The `PaperFactory` only has one task and that is to create Paper objects. That means we have moved the object creation code away from the `PrintPaper` method; all the `PrintPaper` method has to do is simply request a new piece of paper from the `PaperFactory`.

The pseudo code for the `PaperFactory` could look something like this:

```
Class PaperFactory
    Method CreatePaper takes nothing returns Paper
        Return new A4()
    endMethod
endClass
```

And the pseudo code for the new `PrintPaper` method of the `Printer` class would be this:

```
Method PrintPaper takes paperFactory, message returns paper
    Paper = paperFactory.CreatePaper()
```

```
    Paper.Print(message)
    Return Paper
endMethod
```

Thus as we can see the responsibility of creating paper has been removed from the `Printer` class and instead moved to the `PaperFactory` class.

2.3.1 Abstract Factory Design Pattern

The abstract factory design pattern is closely tied to Factory design pattern in that an abstract Factory like an abstract class only defines specification of the implementing class, and doesn't contain any logic at all.

An Abstract Factory is made by making an Abstract class of the factory. Going back to Printer example, imagine that the `PaperFactory` instead was an abstract class, as displayed below:

```
Abstract class PaperFactory
    Method CreatePaper takes nothing returns Paper
endClass
```

Now the Printer class only has to know about a Factory capable of producing Paper, but it will have no information on what kind of paper is produced, an implementation of the `PaperFactory` could now be made for each type of paper that one wishes to produce.

For instance an implementation of the `PaperFactory` for creating A4 papers could be designed like this:

```
Class A4PaperFactory implements PaperFactory
    Method CreatePaper takes nothing returns paper
        Return new A4Paper()
    endMethod
endClass
```

As we can see, the `A4PaperFactory` class – which is an implementation of the `PaperFactory` class – is capable of producing a special type of paper. As such if this factory was used by the `Printer` class we saw in an earlier example,

the `Printer` would be capable of producing A4 papers with messages on them. However were we to want another type of paper, the code for the printer would not need to be changed or copied since we simply implement a new version of the `PaperFactory` class and give that to the `Paper` class.

Summary

A lot of problems arise in code when mixing business logic with object creation logic, this section has explained why the problems occur and what the reasons behind them are. Furthermore it has shown how to solve these problems by using the factory design pattern.

To summarize some of the problems when mixing business logic with object creation logic:

- Increases the complexity in business logic, because of added creation logic
- Removes flexibility of the business logic, forcing it to work only with special classes
- Increase the need for redundancy of business logic to accommodate for new object types
- Makes changing legacy code difficult, as the code is tied to specific object types

2.4 Test Driven Development

Designing code using a test first approach helps direct the design of the code in a way that makes it more flexible. This section will cover how to achive this, along with helpful advice on how to handle certain aspects of the TDD[3] process. As such it will also explain the ways unit tests should be used and what problems that might occur when attempting to write unit tests.

[3]Test Driven Development

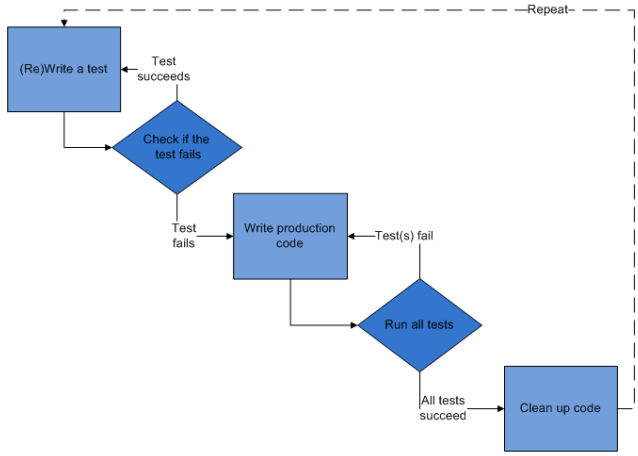


Figure 2.5: The cycle of writing tests used to develop production code (from [Wikd], `http://upload.wikimedia.org/wikipedia/en/9/9c/Test-driven_development.`

The Idea Behind Test Driven Development

The idea of TDD is to write tests of how the program is supposed to function before actually writing the program itself. These tests can be referred to as executable specification, because they specify how single units of the program are meant to be used.

To get an idea of how a TDD process works, look at fig. 2.5. As is shown in the figure, the idea is that you begin the development of a unit in the program by writing a test. The production code is then developed with the goal of having the test succeed. Once all tests succeed, you clean up the code and start the process over, with developing new features and accompanying tests. After multiple iterations, you have ensured that not only does your code have all the features you want, but that those features work as you would expect. Furthermore, if new features were requested at a later time, it would be quite easy to simply add new tests and begin development of the new features, while using the old tests to ensure the old features were not ruined.

By writing the tests first, the developer can easily determine what the final units should look like. If he did not apply a test first approach he would have to do a lot more preplanning, as he would have to state the specifications of the program in some other way. While a TDD approach will reduce the amount of preplanning required, it will not completely remove the need for it. It will still

be required to plan such things as the domain model and the components of the program.

To give an idea of what the executable specifications obtained through TDD will look like, we provide the following example:

Assume you were to make a Calculator. Since this is a simple calculator, it can only do addition, subtraction, multiplication and division. To specify this calculator, one must create a test for each of its features:

- A test showing addition of two numbers
- A test showing subtraction of two numbers
- A test showing multiplication of two numbers
- A test showing division of two numbers

This specification will ensure that the final calculator can perform all these actions or it will not work, thus our tests are enforcing specified features of the calculator.

However, the great thing about using TDD is that you can go deeper and specify what the exact outcome should be. Assume that you wanted to ensure that when the calculator divides by zero, an error is thrown. To do this, all that is required is to simply add a new test:

- A test showing that when the calculator divides by zero, an error is thrown.

As is evident, the more tests written for a certain aspect of the program, the more specified that aspect is. Thus by doing test driven development, you have essentially done two things at once. First, you have created a way to test if the features are still functional; this provides a way to test them if their functionality is changed at a later date. Second, by making tests you are specifying what the output of the program should be, thus if others were to try and use your units in their code, it would be easy for them to understand the provided functionality by simply inspecting the unit tests you provide.

2.4.1 How to write unit tests

When following the TDD approach, it is important to properly understand how to design unit tests, as many problems can arise when writing them. Most of

these problems can be traced to a few common programming mistakes. A good introduction to using test-driven development is Misko Hevery's lecture on the subject, see [Tala].

As mentioned in [Tala], there is nothing that can be said about writing unit tests that would improve the tests, there is no trick to writing them. However there is a lot that can be said about designing code, a correctly designed piece of code can make the process of making a unit test easy, while a badly designed piece of code can make creating a unit test very difficult, if not impossible. To understand what these bad design choices are, we will go through each of them.

Mixing Object creation logic with business logic

To properly design a test for a given class in the code, you must be able to instantiate that object. If the object you wish to test instatiates all its dependencies on construction (in contrast to taking instantiated dependencies as arguments), you are forced to test all these dependencies along with the class.

To give an example of this problem, assume you have a `WebDocument` class, as shown below:

```
Class WebDocument
    Field: Document
    Constructor takes URL
        client = new TCPClient()
        Document = client.Download(URL)
    Endconstructor
Endclass
```

In this example the `WebDocument` creates its own TCP client which it uses to download a document from an URL. If we were to test this class we would be forced to setup a TCP connection every single time. This not only causes the test to be slow it also introduces uncertainty, as the TCP connection could fail.

This problem can be solved by designing the class so that it requires the classes to be provided instead of instantiating them itself. This method is called dependency injection. Consider the following adaptation of the `WebDocument` class, which uses dependency injection:

```
Class WebDocument
```

```
    Field: Document
    Constructor takes Client, URL
        Document = Client.Download(URL)
    EndConstructor
EndClass
```

By making this change, the test creator can choose which object is given as the client. For instance he could use a mock[4] client providing a document of his choosing in the `Download` method.

This basically comes down to giving choice to the unit test writer, without this the unit tester could be required to instantiate almost the entire program in order to just test a single unit. By using dependency injection we effectively remove this issue.

Global State in the Code

Whenever you have global state in your units it becomes very difficult to design tests, as pointed out in [Talb]. This is because actions done in one test will inadvertently affect the result of another test. Thus by eliminating all sources of global state you ensure that the code which you are testing always works in the same manner.

If the developer is not careful, it is often easy to accidentally write code with global state in it, since it can the global state can be quite subtle. By definition, global state occurs every time a piece of code knows about something that is has no reference to, thus it has reference to something that is globally accessible.

To illustrate this, consider the following simple test:

```
Output1 = new A().Calculate()
Output2 = new B().Calculate()
Assert(Output1 != Output2)
```

Since a computer is deterministic, the result of asserting that the two outputs are not equal should always be the same. If the assertion is sometimes true and sometimes false, then we have a global state. This means that global state in code is what makes the code non-deterministic. By its very nature, code that

[4] An object that mimics the behavior of the real object

is non-deterministic is untestable, since a test requires knowing the outcome in advance so it can be asserted whether the result is the same.

Here we provide two examples of commonly accepted code design that produces global state:

Singletons are objects that are only instantiated once. Their instantiation is located on a global variable, since the variable on which the singleton is located is global. That means all objects that use the singleton has their state bound to that of the singleton.

Random numbers, Time and date, etc. are all cases of objects that hides global state inside them. Thus if you use them as part of your code without providing a way for a test to inject them as with other dependencies, you run the risk of the program being untestable. The problem with these objects are they usually hide the fact that they use global state, and as such can easily sneak their way into the code if one is not careful.

Breaking Law of Demeter

One thing that makes testing difficult is if an object does not ask for what it needs, but for the objet that can locate what it needs. The act of asking only what is needed is called Law of Demeter or principle of least knowledge. The idea is that a unit only needs to know about its immediate friends; units it doesn't directly work with should be irrelevant to it. Breaking the law of Demeter is not only considered bad code design, but also makes writing unit test harder.

When writing code, it is not always immediately obvious when law of Demeter is violated. In the real world, however, breaking the law of Demeter often results in absurd situations, and are thus more easily visible.

As an example (adapted from [Wikb]), imagine that you are in a shop and the cashier asks for 10€. What would you do?

1. Give him a 10€bill
2. Give him the wallet and let him find the money

3. Give him the location of a hidden treasure which he should locate and return the difference to you.

As we can see option 2 and 3 clearly violate law of Demeter because instead of giving what is actually required we give something that provides what is actually required.

In the example of the `WebDocument` we ourselves violated law of Demeter so let us show how we could change the code to remedy this. The code that breaks law of Demeter:

```
Class WebDocument
    Field: Document
    Constructor takes Client, URL
        Document = Client.Download(URL)
    EndConstructor
EndClass
```

The modified, more correct code:

```
Class WebDocument
    Field: Document
    Constructor takes ADocument
        Document = ADocument
    EndConstructor
EndClass
```

As we can see, instead of making `WebDocument` go locate the document on some server, we simply make the document a dependency of the `WebDocument` class, thus testing of the `WebDocument` will not even require a mock server anymore. As such designing the test just became a lot easier.

Summary

While a TDD approach will increase the workload of the project as it will require the developer to write a lot of tests, it adds a lot of value in return. The most useful feature of TDD is perhaps that it provides a natural specification of the programs individual units, which could be hard to properly formulate in words. It also enforces good code design practices by making it hard to write unit tests for badly designed code.

Advantages

- Provides test cases for all units, making it easier to see what breaks when units are introduced or changed
- Reduces the amount of errors in the final product and as such reduces time spent debugging
- Enforces proper code design
- Provides specification of the code, making it easy for others to understand
- Makes the writing process of a class easier since you start by stating what you want from a class, instead of how it works.

Disadvantages

- Requires unit testing frameworks to do it properly
- Has a learning curve for those no familiar with TDD
- Increases the develop time as all code produced must also have a unit test to prove it works as expected

Chapter 3

Reference Implementation

In this section we will introduce our reference implementation (henceforth sometimes called the *package grabber scenario*), which we have developed to showcase and test our engine. It will also serve as an example of using the engine as well as the extensions we have provided. In section 5.6, we will detail the actual implementation of the scenario in the engine, the agent programs, as well as the integration of the two. A less complex example, which describes the simple vacuum world ([RNC+96], p. 35), can be found in appendix C.

The reference implementation is intended to cover as many of the engine features and extensions as possible, while not focusing on creating any particularly revolutionary artificial intelligence. As such, we have set up a world that is relativivelly simple with respect to the action and perception repetoire of the agents, and with an environment representation limited in complexity. Thus, the interesting part of the reference implementation is not the scenario itself, but rather the actual implementation. That being said, the scenario is as follows:

The agents in the scenario are tasked with exploring a maze in a discrete two dimensional grid of tiles in order to locate `package`s and bring them to a special tile called the `dropzone` (see figure 3.1). A tile in this scenario can contain several objects, unless they are explicitly forbidden to occupy the same square. An agent, for example, can not move into a tile that already contains another agent, but it can move into a square containing a package or a dropzone. Importantly,

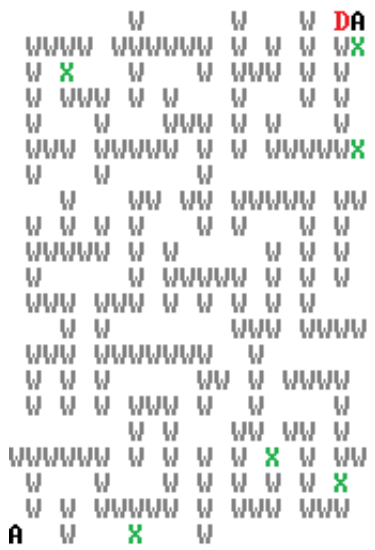


Figure 3.1: An initial configuration of the package grabber scenario. `D` (red) marks the dropzone, `X`s (green) are packages, `A`s (black) are agents and `W`s (grey) are walls

a tile containing a `wall` cannot contain anything else, thus the tiles between walls constitutes the navigable pathway of the maze.

Actions

The three actions *moving*, *grabbing* and *dropping* are enough for the agents to fulfill their task, and as such they describe the complete action specification:

`move(`*`Direction`*`)` moves the agent one tile in the specified *`Direction`*. *`Direction`* is limited to the four cardinal directions, so an agent can only move to an immediately adjacent square. Note that every tile not containing a wall is reachable from every other such tile in the maze when following this movement rule.

`grab` removes the package in the same tile as the agent (if any) from the world, and marks the agent as carrying a package.

`drop` adds a package to the world in the same tile as the agent (if it is carrying a package) and marks the agent as not carrying a package. If a package is dropped at a dropzone, it is removed from the world.

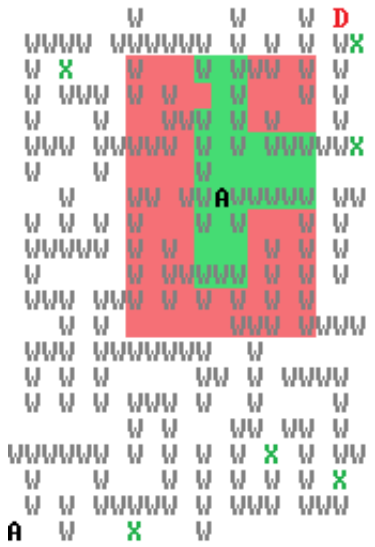


Figure 3.2: Here we see the visible tiles for the agent in the middle of the colored section. Tiles colored in green are visible to the agent while tiles in red are are blocked by walls. All other tiles are outside the agents visibility range, and thus invisible. As can be seen, agents are good at peeking around corners.

Note that this action specification does not mention any means of communicating or otherwise cooperating, which is an otherwise important feature of any multi-agent system. We have not implemented such functionality in the XMAS engine, although it would be a high priority task if we were to develop it further. Normally, GOAL provides built-in communication devices, which we cannot use in the reference implementation, for reasons explained in the implementation section. In general, although several grabber agents can inhabit the scenario, they will not actively work together or compete, although they will try to accomplish the same goals.

Percepts

Now that agents can act in the environment, they must also be able to sense their surroundings in order to make informed choices about their course of action. For this purpose, the following three percepts can be obtained from the world:

Vision: A rather obvious perception device in this scenario is *vision*, allowing

an agent to learn the contents of tiles around it. Each agent can can see a distance of five tiles in any direction, assuming there is no walls or other agents blocking its vision. See fig. 3.2 for an illustration.

Package Possession: Specifies whether the agent is currently holding a package. An agent can only hold one package at a time. This truth value could be mangaged by the agent program, but having it as a percept is easier on the AP. Additionally, an effect could forcibly remove a package from an agent, and we would thus need a percept for that (note that such an effect does not currently exist in the scenario). With our current approach, we provide a snapshot of the state of the perceivable world through percepts and let the AP make of it what it can.

Position: The absolute position of the agent in the grid. In the agent program, we initially used positions relative to the starting position of the agent to build a map of maze. However, using this method we would lose track if the agent was forcibly pushed into another tile (again, such an effect does not exist in this scenario).

Additionally, agents receive their current movement speed (the time it takes to move from one tile to an adjacent one) as a percept, although we do not use it in this scenario.

Agent Program

We have implemented the agent logic for the grabber agent in the GOAL agent programming language. The GOAL instance is connected to an EIS environment, which communicates with the XMAS engine by sending actions and receiving percepts.

It will try to find all packages in the maze and bring them to the dropzone. Note that since packages might be hidden in places the agent has not yet looked, the requirement for scenario completion is not just that no more packages can be *seen* on the map, but also that there are no more tiles to explore.

In order to find all the hidden packages, the agent must explore the maze by moving to tiles it has not stood on before in order to gain vision of other tiles and pathways. The agent logic itself is pretty straight forward; it can be summarized as below:

```
if I hold a package
```

```
        and know the location of the dropzone
    then go to the dropzone and drop it
if I have no package
        and know the location of one
    then go grab the package
otherwise,
    explore the maze further
```

In the pseudo code above, decisions such as "`explore the maze further`" are *goals* the agent sets for itself. When it needs to get to a specific location, it finds a path to that location using the `A*` algorithm, and then follows it each turn until it reaches a goal or finds something better to do.

Chapter 4

System Features

4.1 Overview

Before we begin explaining all the features of the engine, we would like to point out that if it is necessary for the reader to see an actual implementation using all the feature, there is one available in appendix C (Vacuum World).

The goal of the engine is to allow for simulation of a world where agents within are allowed to act. As such, it is important that it can accurately model a state-machine. To model a state-machine, one must have the ability to contain a state and perform actions to change the current state.

A complete UML Domain model diagram is provided in appendix A.

4.1.1 State

In our domain model, we have state stored in three object types:

- World

- Entities
- Modules

World The world is the place all entities are meant to inhabit as either agents of the world or simply objects for other entities to interact with. The world is not defined by the engine. As shown in appendix A.1, it is an abstract class, meaning it is the developer using our engine that defines the world. As such the world can be any type of world needed, it could be a 3-d world, a 2-d world, a world based on tiles or hexagons, or simply be nodes with an undefined number of edges connecting each other.

Entities The world is empty without anything inside it, as such we have the entities which are meant to model the objects one would have the world contain. For example, in our reference implementation, we have a world with packages scattered about a maze. It is then the task of the agents to collect these packages; the entities here are not only the walls of the maze and the packages, but also the agents since they inhabit the world as well. The agents are different from entities in that they all have a name. This name is unique and is meant to be a way of distinguishing the agents from one another.

Modules The modules can be viewed as either the constraints or as the abilities of all entities. For example, if you wanted to constrain entities from moving into each other, you would create a *movement blocking module*, which would contain information on whether or not a given entities is allowed to pass through it. However, if you wanted to give an agent the ability to move, a *speed module* would be required. Whether a certain module is a constraint or an ability is up to the individual module.

4.1.2 Actions

A world is static and unexciting if one is not allowed to perform any changes to it, for this we provide what we have chosen to name actions. There are two different types of actions: environment actions and entity actions. The core difference between them is that entity actions are meant as actions performed by a single entity, such as moving the entity or having the entity pick up another object. Environment actions are actions that affect the entire world. In our domain model, we have chosen to add two actions that are built into the engine, the first is an entity action that gets all the percepts for a given entity called

`GetallPerceptsAction` and the other is an environment action that can shut down the engine called `CloseEngineAction`.

4.1.3 Events and Triggers

The engine relies heavily upon events, this means that all actions performed within the engine is meant to trigger events in response. This can be used to either activate new actions within the engine, or be meant to transfer data to the views listening.

In order to listen to the events, a trigger need to be created with all the events it listens to registered to it. Furthermore, a trigger needs a condition and an action. The condition is a predicate that determines whether the trigger is fired, and the action is the function that is excuted when the trigger is fired.

4.2 Virtual World

The object used to keep track of all entities in the environment is called the *world*. This object is also used to model the structure of the environment; eg. whether it is tile based, graph based or something else entirely. The only restriction imposed on the structure of the environment is that all entities have an associated *position* in it, fitting the data structure describing the environment. This is a pretty loose requirement, considering that it can effectively be ignored.

To give an example of what these positions should look like, imagine a tile based environment the world could consist of a two dimensional array containing lists of entities, with each field representing a tile, and positions represented as (x, y) coordinates. In a graph based environment, the world would contain some structural representation of a graph, and the positions could be references to nodes, or representations of the graph as seen from different nodes.

4.2.1 Entities, Agents and Entity Modules

Entities are the objects inhabiting the world. They are very basic objects, equipped with no definitions of how they are represented in the world or how they can be interacted with, save for allowing other objects to subscribe to events fired by the entity. All this is instead handled by *entity modules*, which

each entity contains a set of. These modules can be queried and called by other objects. An entity could, for example, have a *speed* module – as is the case in the tile extension – specifying how long it takes to move from one position to another.

When modules are asked to identify themselves, they do so by means of a *module type*. Two modules are identical – from the viewpoint of an entity – if their module types are identical. As such, only one occurrence of any module type can exist in a set of modules. and a module of type t on an entity e can unambiguously be reffered to as $e.M_t$, where M is e's module set.

It is perfectly legal (and sometimes recommended) for a module to identify itself by another type. This means that if a module m_1 of type t is registered to an entity e, which already has a module m_0 of type t attached (such that $e.M_t = m_0$), m_1 replaces m_0 in the set, and $e.M_t = m_1$. Additionally, when the new module is registered to the entity, it checks to see if any modules with the same type is already attached. If that is the case, it stores a reference to the original, and re-attaches it when it is itself detached. This allows for using filter modules, which can use the functionality of the module they have replaced to produce a modified output.

As an example, consider an entity e with a *speed* type module s_0. Assume that s_0 has a method *Speed*, such that $Speed(s_0)$ returns the speed of e. If it is for some reason desired to change the movement speed of entity e by 50%, it is recommended practice to register a new module s_1 to the entity, which identifies itself as a *speed* type module, and likewise has a method *Speed*. As s_1 is registered to the e, it stores a reference to s_0, and replaces it so that $e.M_{speed} = s_1$. Now s_1's *Speed* method can be defined such that it returns half the value s_0 would, so that $e.M_{speed}.Speed = s_1.Speed = \frac{s_{0.Speed}}{2}$. If at some point this effect is no longer desired, s_1 can be deregistered from e, which causes s_0 to be reattached and $e.M_{speed} = s_0$ once again.

Note that this "chaining" of modules can be applied indefinetely, allowing several modules to affect a single property of the entity. However, the module methods are called as a stack, which means that the method of the module inserted last is the first to be called. This imposes somewhat of a limit, and may not work as desired in all cases. Consider a stack of n modules, where module m_1 has been pushed first, followed by m_2 and so on, so that m_n is at the top of the stack. Then a module m_i can not directly alter the way modules $m_{i+1} \dots m_n$ changes its output. This may or may not be desirable, depending on the situation. It is not possible, for example, to apply an effect causing an agent's speed to be set to x, no matter what happens, using this method.

Another issue the designer should be aware of is that it is possible to create a

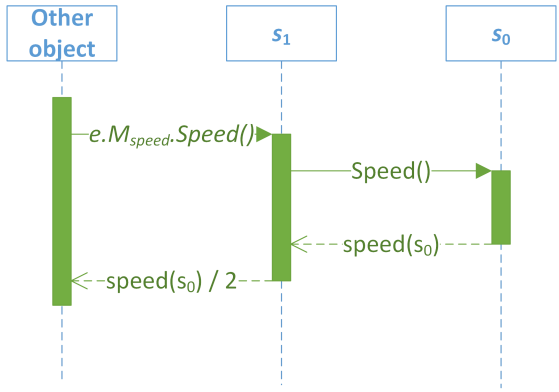


Figure 4.1: A *speed* module s_1 has replaced another (s_0) of the same type on entity e. Some object requests the speed of entity e by querying its *speed* module s_1. s_1 then queries the *speed* module it replaced, and returns half its value.

module m_1 of type t, intented to replace another module m_0 (as in the above example), and not implementing the same public methods in m_1 as in m_0. Doing this will cause a runtime exception.

An *agent* is a special entity which have a unique name and can collect percepts. When an agent is asked to return all of its percepts, it queries each module for any available percepts, and returns those as a collection. The agents are designed to be controlled by an APL, they are done so through an `AgentController`, see sec. 4.6 on how to work with agent controllers.

4.3 Events and Triggers

In this part the concepts and ideas of both events and triggers will be explained. We will go through their intention and how to use them with the engine. Furthermore, a couple of examples will be provided to give the general idea of what they can be used for.

4.3.1 Concept

In the natural world, all actions have a reaction, these reactions could be thought of as events meant to trigger when such actions are performed. Thus an engine for modeling a virtual environment must provide as many features as those of the real world environment.

Events To be clear, an event in the context of this engine is the occurrence of something, for instance an event could be that "an Agent has moved", or "an Agent has picked up an item". Furthermore, an event also has the duty of providing necessary information for the listener, giving the listener a correct idea of the meaning behind an event. In the case of the event which signified the movement of an agent, it is necessary to provide the listener [1] with information of which direction the agent moved, where its starting position is and how far it has moved. Since a listener might be operating in a different thread, the listener is completely dependent on this information, as it might no longer be retrievable at the time the event is being analyzed. For instance, if an agent moved and then was killed and removed from the world, its position would no longer be stored in the world. As such, the listener would have no way to determine where the move had ended if this information was not provided in the event.

Triggers Triggers in our engine are the means to which listeners gain access to events. A trigger in our engine is the combination of three different parts.

- Events
- Condition
- Action

The events are what the trigger is listening for. These can be any type of event, and a trigger can be registered to any number of events. But only one event is required to "trigger" a Trigger. For instance, if a Trigger is listening on both the events "10 seconds passed" and "Agent has moved", then the Trigger will be "triggered" when either of these events occur. However it will be triggered each and every time such event has occurred and is not limited to just one occurrence.

[1] Listeners refer to the object which is listening to the occurrence of an event, with the intent of reacting to it

The Condition is a built in predicate for the trigger to check if it is willing to respond to the event. If the condition is satisfied, the trigger's action is fired. A condition should only be used in cases that is not covered by another event. For instance, say you have the event "An agent has moved". Let us call the agent that moved a_m and the agent whose movement you are interested in a_i. The condition on the trigger would then be:

$$\textbf{is } a_m = a_i \textbf{ ?}$$

As we can see the condition narrows the range of events that are responded to at the cost of added calculations. In this case it would be much better to subscribe to the event "Agent a_i has moved". This is purely an example as events can not be tied to specific entity instantiations as events are defined at compile time.

The Action of a trigger is the part that performs the work, it is a method which is executed once an event has been raised and the condition is satisfied. For instance if a trigger is meant to write a message when a specific event has occurred, then this is where the action of writing such a message should be placed.

4.3.2 Entities and `EventManager`

For triggers to become part of the engine it is required that the trigger is registered to the engine, however it is of crucial importance what one registers the trigger to. A trigger can be registered to either a specific entity or the `EventManager`. A Trigger registered to the `EventManager` will be triggered each time an `Event` that it is listening to is fired. However a `Trigger` registered to a specific entity will only be informed of events raised on the specific entity instead of when the event is raised for every single entity.

An example of this would be: assume you have a Trigger $\texttt{T}_1$ with the event "An agent has moved", and $\texttt{T}_1$ is registered to Agent `A`. Additionally, you have a Trigger $\texttt{T}_2$ with same event as $\texttt{T}_1$, but this trigger is registered to the EventManager. To give a complete picture, also assume there is an Agent `B` which has no Triggers registered to it.

This provides us with two scenarios:

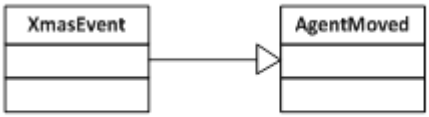
Agent `A` has moved: In this case, both $\texttt{T}_1$ and $\texttt{T}_2$ is triggered, since $\texttt{T}_1$ listens on Agent `A` and $\texttt{T}_2$ listens on any agent moving.

Agent `B` has moved: In this case, only T_2 is triggered, for the reasons stated above.

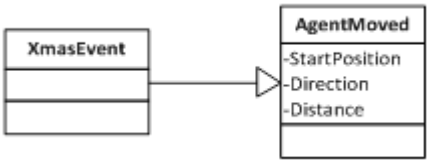
4.3.3 Example of making and using an `Event`

Let's assume one was to make an event which was fired each time an agent had moved, let us name this Event: `AgentMoved`.

First, make a class extending the `XmasEvent` class as shown below:



Then, add all the necessary data fields on the newly created event class.



To utilize the newly created event, it must be raised when appropriate. In this case, the appropriate place would be to raise it during a move action.

In this action, after the movement of the agent had been performed, the method `RaiseEvent` would need to be called on the entity that is being moved.

Summary

Events are what provides the engine flexibility and allows making reactions to others actions. Events are designed for ease of use and are meant to be used as much as possible. Triggers are used as a way to interface with events and they are the only way to connect an object to the event it wishes to listen to.

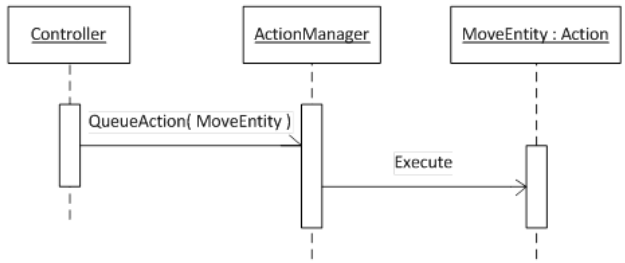


Figure 4.2: Sequence diagram illustrating how a `MoveEntityAction` is processed. The idea is that a controller, such as keyboard input or an agent language, queues an action such as moving an entity on the action manager. The action manager will then execute the action as soon as it is ready.

4.4 Actions

`XmasActions` in the UML Domain model diagram in appendix A refers to the object type that performs actions inside the engine. The reasoning behind these actions being its own class is to ensure only one action at a time is being performed. This is because there are many separate threads operating on the model code at once and as such, there must be a way to activate only one action at a time.

For the task of executing the actions we have the `ActionManager`. Its job is to take in one action at a time and place them in a queue (see fig. 4.2).

4.4.1 Action Types

The engine is equipped with two different action types. One of them is the `EnvironmentAction`, which are actions that perform changes on the entire environment. Examples of such actions are closing this engine or adding/removing entities from the world.

The other action type is the `EntityAction`. This action type is meant as an action that a single entity performs; ideally the actions should be as atomic as possible. In our reference implementation we have given some ideas how these

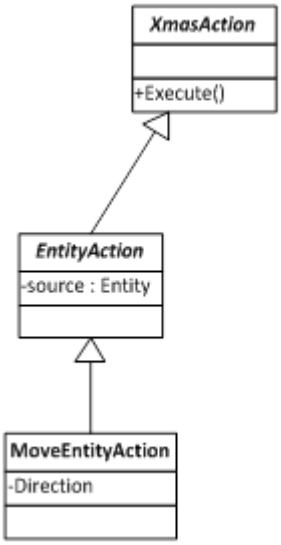


Figure 4.3: Illustrating the inheritance of the newly created `EntityAction` "MoveEntityAction"

actions work, such as `grab`, which is an action that grabs a package from the tile the executing agent is standing on.

4.4.2 Example – Move Entity Action

Here we will show how an entity action is constructed by inheriting the `EntityAction` class. As shown in fig. 4.3, we have created a `MoveEntityAction` class with one field containing the direction of the move.

To have the action actually perform something, it is required that an abstract method `Execute` is implemented. This execute method is the method that is executed by the action manager. The implementation of the execute action could then look something like the following pseudo code:

```
method Execute returns nothing
    NewPosition = GetPositionOf(World, This.getSource()) + Direction
    Wait(MOVE_TIME)
    SetPositionOf(World,This.getSource(), NewPosition)
EndMethod
```

As one can see, the idea is that you find the position of the source of the `EntityAction` and use that to generate the new position, which is its old position incremented by its direction vector. The wait is there to give the move a speed, as it would otherwise be an instant movement.

4.4.3 Summary

Using actions is fairly simple and serves to shield the user from the tedious and error prone workings that takes place behind the scenes. It is meant to ensure thread safety and allow multiple threads working with the engine at once. These were the exact reasons we chose this design, as we ourselves had to deal with the problem of interference from multiple concurrent threads. Furthermore, it will also reduce code redundancy as generic actions can be reused by other actions. The problem with this design is that it in a sense remakes what is already implemented in a programming language. After all, running procedural code is what programming languages are meant to do. However, in return it provides a lot of utility and makes it possible to make tools for simplifying the process of making actions. It also gives the ability to differentiate between different action types and even create new action types if one wishes so.

4.5 Converting Actions and Percepts

Both actions and percepts are very abstract objects, and the XMAS engine can not know how they are represented in different APLs. This means that the system designer needs a way to translate actions from foreign types – such as GOAL actions – into `XmasAction`s, and `XmasPercept`s into percepts of foreign types. We have provided the base necessities for implementing this functionality with *converters*. A converter is, as the name implies, a class that takes objects of a foreign type and map them to internal types or vice versa.

To use our converter, the designer simply extends the `XmasConversionTool` class, then proceeds to define what foreign type he wishes to convert from or to an internal `XmasType`. After this, the user defines each individual object of the foreign type by extending the `XmasConverter` class. Once this task is complete, the converter objects are added to the Tool object, and conversion between foreign and internal types are now possible.

This feature is used as part of our EIS extension for converting IILang data into XMAS percepts/actions.

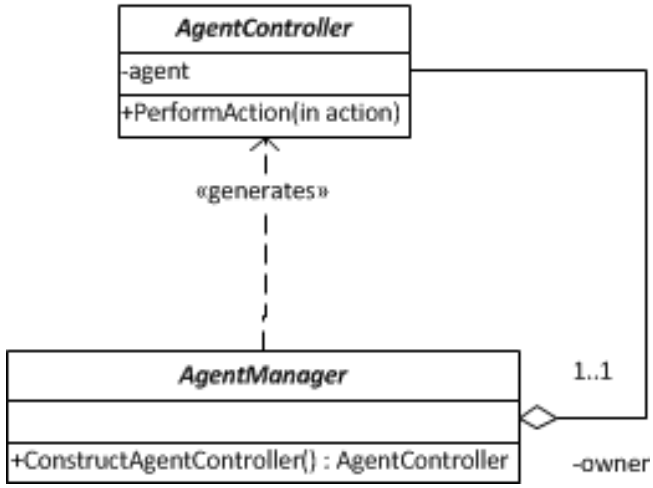


Figure 4.4: Domain model for Agent Controller

4.6 Agent Controllers

The purpose of the agent controllers is to be able to control agents in the engine from outside. This section will cover our setup of an agent controller.

To get an overview of the classes used for the `AgentController`, look at fig. 4.4.

4.6.1 Concept

The engine is designed to support the ability to be adapted for all APL types, this means that the engine itself does not support all APLs but instead provides a framework for quickly designing interfaces between the engine and any APL. There are two classes that one must use in order to properly design the interface:

The `AgentManager` has the duty of speaking directly with the agent language it attempts to interface with. Its job is to spawn an `AgentController` for each agent the AP wishes to take control of. The `AgentManager` is in that sense much akin to an Abstract Factory (see sec. 2.3) , which – according to the design pattern – is an abstract class with a method generating a certain type of object, without restricting exactly which object is generated, as long as it is of the specified type. The idea is

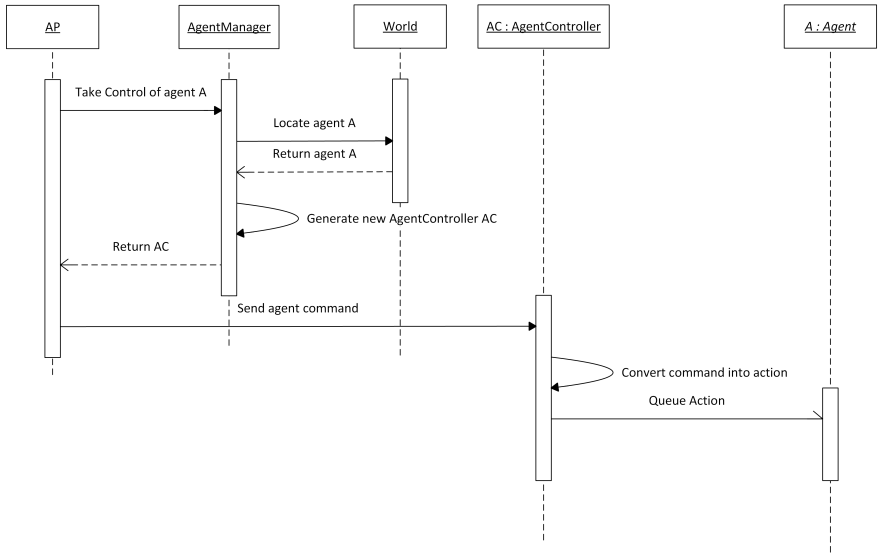


Figure 4.5: This sequence diagram shows the process of an AP taking control of agent through the `AgentManager`, and commanding it through the `AgentController`

that if you have a `GoalAgentManager` then the controller it constructs would be `GoalAgentController`. The methods required by both the `AgentController` and the `AgentManager` are abstract. Thus, we ensure at compile time that the engine framework is properly used.

The `AgentController` is the link between a single agent and the AP. Its job is to take all commands sent to it by the connected AP, transform them into actions understood by the engine, and apply them to the agent that it controls.

To simplify the `AgentController` design, it provides the method `PerformAction`, which makes it easy to execute actions on the agent it controls. When the `PerformAction` is called, the `AgentController` queues the action given through the method and puts the `AgentController`'s thread to sleep. Once the action has been executed by the engine, the `AgentController` is woken up and returns from the `performAction` method. All percepts received by the `AgentController` during this time is stored on the `AgentController` and can be easily accessed by the actual implemenation of the `AgentController`.

The process of an AP taking control of an agent is illustrated in fig. 4.5. The AP calls the `AgentManager` to locate the agent it wishes to assume control of.

The agent is located through a string (its name) which is unique to it and ensures that only one agent is taken. When the `AgentManager` finds the correct agent, it will immediately generate a new `AgentController`. The AP will not gain access to the agent but instead it will gain access to the `AgentController`. Now that the AP possesses the `AgentController`, it will have the ability to send the `AgentController` commands. These commands might not be understood by the engine if the APL is foreign enough to the engine's own language and as such it is the duty of the `AgentController` to convert these commands into actual actions which the engine can understand.

4.6.2 How to use agent controllers

To use the built-in controller features of the engine, the designer must provide his own implementations of `AgentController` and `AgentManager`. The designer must do this for every different APL he wishes to use in the engine.

The classes `AgentManger` and `AgentController` both provides functionality as part of their own classes but also requires some methods that must be implemented for the classes to function.

For the `AgentManager` the designer must provide a method that produces `AgentControllers` of their implemention, along with locating an agent though the `AgentManager` does provide some assistance in that regard, in the form of a agent locating method defined as: `TakeControlOf`, which takes the name of an agent as a string, and returns the corresponding `Agent`. Furthermore, it should be noted that the `AgentManager` is also automatically designed to create threads for the `AgentControllers` it constructs.

For the `AgentController`, the designer must provide all logic defining how a controller handles a given agent, this means getting the percepts from the agent, analyzing the percepts and deciding an action (Could be outsourced to another APL such as GOAL) and then queue said action to the agent. The AgentController has the method `PerformAction`. This method can be very useful as it queues an action automatically to its agent, then blocks the thread until the action has been performed. Furthermore, this action will also trigger a C# event called `PerceptsRecieved` in the case that the agent controller actually recieve any percepts during the action. For instance, this event could be used for where the `AgentController`, decides the next action for the agent to perform, as done in the Vacuum World Example in appendix C.

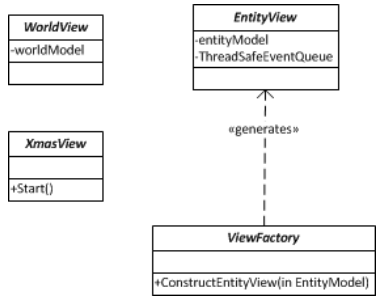


Figure 4.6: UML diagram of the view

Summary

The agent controller is designed to be very lightweight, since we do not want to impose any restrictions that might limit an APL which we know nothing about. As such, the `AgentController` is more akin to a convention or a design pattern for how interfacing with agents should occur. It provides the skeleton of how a link might be designed but does not impose any restriction on how the link should be set up.

4.7 View

The engine is designed to assist the user in all parts of the process when making an environment, this also extends to the visualization of said environment. However, since our goal is to have as few restrictions on the model as possible, our knowledge of that view's representation is very limited.

4.7.1 Concept

The view API which the engine provides consists of four abstract classes that are meant to be implemented by the user. The four classes can be seen in fig. 4.6. We will go through each class and explain how they are meant to be implemented.

`XmasView` The `XmasView` class is very simple; it only provides a single method that is required to be implemented. When the engine starts the view up it

generates a thread for the view and the `Start` method is the first method to be executed inside that thread. The start method could contain an endless loop that on a time interval updates the view. Another task of the implemented view is also to update its `ThreadSafeEventManager`. The `ThreadSafeEventManager` ensures that events sent from the model thread of the engine is not immediately executed, but instead lie dormant in the `ThreadSafeEventManager` until the view thread is ready to execute them. How many events that one wishes to execute is up to the user. We also provide the appropriate methods for the user to specify exactly how long he wishes to wait for the next event, or if it should timeout. The `ThreadSafeEventManager` is very important to the view as without it, designing view code becomes complicated as one need to constantly ensure that no concurrency bugs has been applied to the system.

`WorldView` The `WorldView` class is added because of the long term benefits, as of now it provides nothing for the designer. However if we found benefits to add to the class, making it ahead of time, even if it is initially empty, can have many benefits as the project expands.

`EntityView` Much like the `WorldView` class, the `EntityView` is also very minimal. However, it enforces certain things that the user of the engine should take care of. First, it automatically makes a `ThreadSafeEventQueue` from the entity and attaches it to the `ThreadSafeEventManager` which should be provided by the `XmasView`. The idea is that all events the `XmasView` wishes to listen on should be subscribed to by registering its triggers on the `ThreadSafeEventQueue`. This will ensure that when the view updates the `ThreadSafeEventManager` all events pertaining to the specific entity is also updated on the `EntityView`'s triggers, but done so on the view thread instead of the model thread, separating the two threads completely.

`ViewFactory` The `ViewFactory` is meant to include all objects with low life cycle used by the view, it is also designed specifically to construct new `EntityViews` during runtime of the engine. In order to know which `EntityView` belongs to which `XmasEntity`, one is required to register all types of `XmasEntities` and link it to its counterpart `EntityView`. For instance, assume you have a class inheriting `XmasEntity` called `Wall`, and the `Wall`'s representation called `WallView`, then you need to manually register inside the `ViewFactory` that `Wall` is represented by `WallView`.

Summary

The view framework provides four classes each with their own advantages; they each represent a part of the model of engine. They are designed to assist the

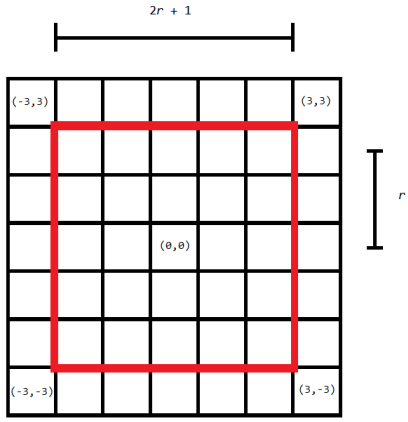


Figure 4.7: Illustrating a 7×7 tile map. The tiles inside the red zone are being queried by specifying the tile at $(0, 0)$ and the range $r = 2$

user in keeping his code threadsafe so that as few problems as possible arise.

4.8 Engine Extensions

Since most of the engine is very abstract in functionality, we have made three extensions, which makes it easy to implement a tile based environment, communicating with EIS supported APLs and log events.

4.8.1 Tile Extension

This extension represents the world as a two-dimensional array of tiles using what we will call a tile map(as can be seen in fig. 4.7). Tiles in this sense are squares that fit exactly one entity. We have implemented it so that the tile in the center has the position $(0, 0)$. This means that all positions are given relative to the origo tile at $(0, 0)$. As a consequence of this, a tile map must have

odd dimensions, as it would otherwise not have a center tile. If the user tries to access a tile that is out of bounds (for example the tile at position $(0, n+1)$ in a $n \times n$ tile map), a tile containing a special entity signaling that the tile is not part of the world is returned. This ensures that querying the tile map for a tile at any position will never fail and always return a valid value. As well as accessing tiles at arbitrary positions, the tile map can be queried with a position and a range r, in which case a two dimensional array of size $(2r+1) \times (2r+1)$ is returned, containing the tiles in that square (see fig. 4.7). This can be used when the querying of large pieces of the map at a time is needed. When determinining an agent's vision (described below), we use this functionality to collect all tiles in the agent's visible range and then filter out the obscured tiles.

In the tile extension, we have provided several modules that can be equipped to agents to make them better suited for inhabiting a tile based environment. For example, the *movement blocking*- and *vision blocking* modules apply to all entities with a physical presence in the environment; given another entity, they specify whether the entity they are attached to blocks the aforementioned entity's movement or vision, respectively.

Also provided in the Tile Extension is a `MoveUnit` action which is designed to move agents along a vector inside the Tile world. This action requires a *speed* module, defining how long it takes for an agent to move one tile.

Vision

The tile extension also provides means for seeing tiles around an entity via the `Vision` module. All entities that are able to sense their surroundings also have a `VisionRange` module which – as the name implies – defines how far (in tiles) the entity can see.

When the vision module is asked to return its percepts, it asks the world to build it a `Vision` object, which it returns. The `Vision` object uses an algorithm described in sec. 5.5.1 to assemble a set of mappings from positions (relative to the entity) to tile references.

4.8.2 EIS Extension

The EIS extension provides means for communicating with an EIS instance over a socket, as well as serializing and deserializing percepts and actions encoded in

an XML representation of EIS' IILang format. The extension features a custom agent controller and manager, which have been developed to work with EIS.

As EIS is implemented in Java and our engine is written in C#, information can not easily be passed between the two in a native manner. Instead, we have opted to have them communicate over sockets. As EIS already supports formatting IILang objects to XML, we chose this to encode information passed over the sockets. We have implemented our own IILang object tree in C#, which implements XML serialization and deserialization, as well as a Java class used to parse XML to IILang objects.

Furthermore, we have implemented special package streaming objects in both C# and Java, which sends the size of a payload before the actual data when streaming XML over sockets. This allows us to detect when an XML message has been completely received. This means that designers wanting to use EIS with our engine should implement our accompanying Java libraries, as well as the EIS engine extension.

In order for an EIS instance to connect to an agent controller, it must connect to a socket which is known by both the EIS instance and the engine at runtime. The agent manager listens to this socket and accepts the connection. As the EIS instance connects, it receives a new socket which can be used to communicate with the agent controller. It now sends an XML message with the name of the agent on this socket, and the agent manager constructs an agent controller tied to this name and socket. The controller and the EIS instance now have their own private socket connection to communicate on, and the agent manager proceeds to listen for other APL instances requesting an agent controller.

Execution Protocol

The EIS instance can now proceed to send actions to be executed by the agent controller. When such an action is received, the controller enqueues it, and sleeps till it is finished, at which point it resumes listening for actions on the socket. The request to return all percepts is just an action with the name `getAllPercepts`, which causes the controller to gather all available percepts from the attached agent and sent them to the EIS instance via the socket. Note that by default, this is the only way perepts are sent; percepts are returned in response to no other actions. Since the agent controller effectively blocks on actions, the EIS instance can not have the controller queue other actions or return percepts when it is executing an action that takes time, such as the tile extension's move action (this is a restriction imposed by the agent controller base class as described in section 4.6, as it is the default behaviour of the `performAction`

method).

Problems With the Chosen Execution Protocol The execution sequence deescribed above is very simple, but has some downsides. Consider, for example, that two agents wish to communicate with each other through the engine via a `talk` action. It could be that agent a_1 wanted to ask agent a_2 whether a certain tile was a desirable place to go. In that case, a_1's APL would have the action queued in the engine, which would execute it and place the question in a_2's mailbox. If, however, a_2 had just started a lenghty action, such as moving, its APL would not get notified that it had been asked a question until the move was complete, and the controller could respond to the `getAllPercepts` action. This introduces quite some delay in performing such actions, which are rather important in a multi-agent system.

To remedy this, the system could be designed such that the controller instead blocked on the call to return all percepts until the agent had some new percepts available. In the communication example described above, a_2 would immediately perceive that a_1 had asked it a question, which would cause its controller to send all a_2's percepts (including the question) to the waiting EIS instance. Assuming that the corresponding AP prioritizes answering the question, a_1 would have its answer in the shortest possible amount of time. In general, allowing agents to perform multiple actions at the same time makes the world more responsible in a number of ways. As another example, agents in a tile based world (or any world that allows vision) could subscribe to events on tiles they could see, and be able to respond when eg. an enemy moved into one of them. This would allow them to communicate the offenders position to nearby agents, or simply give the AP a chance to preemptively figure out what the best possible action would be to execute next.

This method does have some problems. How, for example, does a_2's AP know that it should prioritize answering the question, and not, say, command the agent to begin a new move action? Since a_2 is already in the middle of a move, it would most likely break the rules of the environment. To remedy this, the agents need to return the action(s) they are currently executing as percepts, and the AP would have to consider these when choosing actions. For larger environments and agent programs, this would complicate the agent logic and percept pool.

4.8.3 Logger Extension

One of the extensions that we provide with the engine is a simple logger, this logger is implemented as a view on the engine and it is meant to be used by others to add a logger for their environment.

To use the logger all that is required is that the user extends the class `LoggerView`. The logger will then provide the extending class with a `ThreadSafeEventManager`, which will have its events automatically executed. As such, the only thing required by the user is to create a `ThreadSafeEventQueue` and register the triggers with the events the user wishes to log.

The `LoggerView` is constructed with a `Logger` object, on which the user can call the method `LogStringWithTimeStamp`, providing a string to be logged as well as the importance of the event, that is, what debug level it is on. The `Logger` class is instantiated with a maximum debug level, and will not log messages from events with higher debug level. In this way, the user can specify if he only wants critical errors, critical errors and warnings, or all information to be logged.

The user must also provide the logger with a `StreamWriter` object, this object can take many different forms however for logging we recommend using it to wrap a file stream.

For an example of how the logger is used, see appendix C. This uses the logger as a view, to track the movements and action of the vacuum cleaner in vacuum world.

Chapter 5

Implementation

5.1 Architecture

The section will cover how all the components of the engine interacts with one another, it will detail how flow of information is transferred through the engine and into the components connected to it. The component diagram of the Xmas Engine can be found in appendix B.

The components of the engine are

- Model
- World Creation
- View
- Controller

5.1.1 Model Component

Requires: `XmasWorld`, `XmasAction` and `Trigger`

Provides: `Percept`

The model component is responsible for handling internal interactions of the engine. These interactions are based on which `XmasAction` it is given.

The model component has three requirements, these requirements are necessary for the model component to properly execute the environment requested by the user.

The first requirement of the model component is the `XmasWorld`, the model component uses the `XmasWorld` by giving it to `ActionManager`, the `ActionManager` then gives the `XmasWorld` to all `XmasActions` as they are about to be executed.

The second requirement is the `XmasActions`, all `XmasActions` queued on the model component are executed by the `ActionManager`. An `XmasAction` are not executed immediately however, as they wait until all prior `XmasActions` executions has been completed. Once queued to the `ActionManager`, they are provided with all necessary dependencies such as the `XmasWorld` and the `EventManager`.

`XmasAction` is designed to allow other threads the ability to interact with the engine. The reason is that we did not wish for multiple threads to change the state of the model component at once is that would force the designer using the engine to deal with multi threading problems. To guarentee that this is never necessary we provide the abiltity to inject code into the `ModelComponent` thread, which is transferred in the form of an `XmasAction`.

The last requirement of the model component is `Trigger`, the model component takes any number of triggers and inserts them in the `EventManager`. When an `XmasAction` raise's an `XmasEvent` on the `EventManager`, the `Triggers` that are registered to that `XmasEvent` are all triggered.

The only thing that the model component provides is the `Percept`, each `Percept` is something that an agent can sense. An `AgentController` connected to the model component can receive these `Percepts` which it is meant to use for analyzing the agent's next move.

The model component is made of many classes however the three `XmasModel`, `EventManager` and `ActionManager` are what provide the core features of the model component and as such is the only ones shown in the diagram. When going into details of the exact design of the engine it will be evident that the class `XmasEntity` also provides some of the features of both `EventManager` and `ActionManager`, however it only does this to make the feel of using the engine

more natural. For instance when moving an entity we thought that it would make sense that the code for this was `Entity.QueueAction(new Move())`, instead of `ActionManager.QueueAction(new Move(EntityToBeMoved))`. In actuality the code does the same thing since in the first case: All the `Entity` does is to call the `ActionManager` in the same way we just showed and then use itself in place of the `EntityToBeMoved`. This is the reason why the `Entity` is not shown in the model component as it has no relevance when understanding the component itself.

5.1.2 World Creation Component

Provides: `XmasWorld`

The world creation component is responsible for making a world for the engine's entities to inhabit. The world is created when the engine starts to execute, as such its internal class `WorldBuilder` only contains a blue print for which entities it should construct and not the actual entities themselves. It does this by storing a function for each entity, those functions contains the information on how each of the entities should be constructed.

The user of the engine is meant to implement his own `WorldBuilder` class, that implementation should contain knowledge on how the world he wishes to construct is created. That means if for instance wants to use a Tile based world then his implementation of `WorldBuilder` should construct a tile based world.

5.1.3 View Component

Provides: `Trigger`

The view component is meant as the component that visualizes the model of the engine to the user, it does not enforce how the visualization is done or in which way the visualization occurs. It only provides the tools necessary to perform this task.

The view is meant to register `Triggers` on the model component, these `Triggers` contains `XmasEvents` when an `XmasEvent` is raised, the `Triggers` with those `XmasEvents` are triggered. The idea is that when a `Trigger` is triggered that means the current state of the model component has changed, the view uses these `Triggers` to be informed about such changes, and are thus able to change

its own state in responds correctly making it able to visualize the new model state.

5.1.4 Controller Component

Requires: `Percept`

Provides: `XmasAction`

The controller component's responsibility is to command `Agents` to perform actions inside the world. The controller component does this by making the `AgentController` send `XmasAction` objects to a specific `Agent` in the model component. Where upon that `Agent` will perform said `XmasAction`, once the model component has executed all prior actions.

The controller component also has ability to receive `Percept` objects back from the engine, these `Percept` contain data about what the `Agent` it is controlling has sensed. These `Percepts` are meant to be analyzed by the controller component to determine what its next `XmasAction` should be.

The controller component is made up of abstract classes which the user of the engine must first implement; these implementations could be setup to act as an interface between a single APL and our engine. This means that for each APL one must make a new implementation of the controller component. To reduce the burden of the user we will in our extensions provide the ability to interface with EIS supported APLs.

Furthermore the controller component is not only designed to make interfacing with different APLs easier, it is also meant to be used when making an interface between the user and the model component. For instance if one wished to control an agent with the keyboard, then an Keyboard implementation of the AgentController and AgentManager should be made, where it would be possible to bind the queuing of move actions to specific buttons on the keyboard, this done as part of our Reference implemenation that can be seen in its source code.

Summary

The architecture of the engine shows the connectivity between each of the components. The Model component which job it is to ensure proper interactions

occur inside the world. The world which is constructed by the World Creation Component, meant to be designed along with the world itself.

The interactions of the model component are provides by the controller component which task it is to command the agents inside the engine, and make it so they are given intelligence. And lastly the view component which only task is to visualize the state of the engine.

5.2 Model

5.2.1 World

To be able to unambiguously reference an entity, they are assigned an `id` (represented as a number) in the engine. This is relevant when, for example, a backend APL such as GOAL executes an action involving other entities than the agent it is controlling. In this case, the entity's position can be amiguous, since several agents may very well occupy the same spot in the world.

To hold references to all entities in the environment, the `XmasWorld` class contains a set of mappings (a C# `Dictionary`) from `id`s to entities. Since all agents have a name, it also contains mappings from names to agents.

When an entity is added to the world, the variable holding the last used `id` is increased by one, and the entity is associated with this number. This ensures unambiguity, since no number can be used twice. However, it does impose a limit on the number of entities that can be added to the world. We have represented the `id` as a 64 bit unsigned integer (C# `ulong` type), so it supports adding more than 1.8×10^{19} entities. Even in an environment that is meant to run indefinetely, and where entities are added and removed often (such as a server based website indexing tool), this limit is still very hard to reach.

The process of assigning `id`s to entities is handled in the `AddEntity` method, which takes as arguments the entity to be added and an `EntitySpawnInformation` object, containing the desired position of the entity in the world, and any other relevant information, such as initial state. However, this only occurs after the user-implemented method `OnAddEntity` is called with the entity and spawn information as arguments, and has returned success. This method is overridable by the designer, and can be used to ensure that entities are added properly to the custom world, or not at all. For example, if the world has the restriction that no two entitites can start in the same position, `OnAddEntity` can be imple-

mented so as to return failure when an the entity in question would be spawned in an occupied position. Alternatively, it may correct the error, for example by placing the entity in an adjacent, unoccupied square and return success. In any case, the `AddEntity` method propagates the return value from `OnAddEntity` to its caller when it returns.

The `RemoveEntity` method dereferences the entity by removing itself and its `id` from the previously mentioned set of mappings. Similarly to the `AddEntity` method, it calls the user-supplied `OnRemoveEntity` method, and returns its return value.

5.2.2 Entities and Entity Modules

When designing the `Entity` class, we wanted to detach the properties and behaviour of entities from the class itself, and segmentize them into smaller, succinct objects. In essence, we wanted to be able to construct an entity that could, for example, move and speak by assembling it from a movement object and a speaking object. In object oriented programming languages such as C#, problems like this are typically accomplished by means of inheritance. It would indeed make sense to let a `MovingAndSpeakingAgent` class inherit from the `MovingAgent` and `SpeakingAgent` classes, which would then provide the desired behaviour. Unfortunately, C# does not support multiple inheritance; a class can not directly inherit from more than one class, although it can inherit multiple interfaces. Instead of using inheritance, we designed the module system described in section 4.2.1.

5.2.3 Events and Triggers

This section will cover the inner workings of how events and triggers are connected, as well as detail why we designed it the way we did. We will also cover the exact procedure when an event is raised, to defuse any confusion there might be as to what happens inside the engine.

Explanation

By themselves, events are not particularly complicated since they are essentially just data structures that are transferred to all its listeners upon triggering. As such we shall do a close examination of how exactly the `EventManager` works.

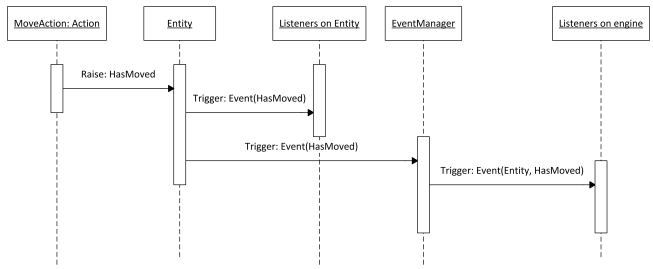


Figure 5.1: A sequence diagram of an event being raised on an entity

The `EventManager` is tethered through the engine to all entities that are inside. Whenever an entity has an event raised on it, it is copied to the event manager which also raises the event. The intent is to minimize the number of events needed to cover a single case. To be clear on how exactly this transpires, we have drawn a sequence diagram shown in fig. 5.1.

As can be seen in the figure, an action – in this case a move action – raises an `EntityMovedEvent` on a given entity. The entity then calls all triggers registered to it, where the trigger also contain the event being raised. After this, the entity informs the `EventManager` that an event has been raised on it, which causes the `EventManager` to also call all its registered Triggers with the given Event. Once all relevant triggers have been informed of the Event being raised, the procedure is complete and the `EventManager` returns to its dormant state.

In the case of events that are not linkable to a specific entity such as an "Engine Close Event", the event is raised purely on the `EventManager` itself. Otherwise, the process is exactly the same as above, except that no particular entity is involved.

Considerations

As there were many considerations that went through our design process, we will take each component of this area and break down why exactly why we designed it as we did.

Problems of C# events and why we chose to design our own events The language which our engine is written in is C#, one of the good things about C# is that events is built into the language. As such it may come as a surprise that we have chosen to re-implement events ourselves. However, while the name might be the same, the intent between C# events and our events is so different that it is impossible to compare the two. The intent behind C# events is to keep maintenance on single objects, so that changes to a given object can affect its linked objects without having to be designed specifically to do so. This allows for really decoupled projects and is what makes object maintenance in C# easy. However our events are not meant for such low-level tasks. Instead they are meant to allow reactions to occur in response to other actions. Furthermore, C# actions are bound to a specific class, and can only be fired inside methods of an instantiation of the specific class. The events we have designed are meant to be raised by all types of class that wish to signal such an event has occurred.

To give an idea of what sort of problems that would arise from using C# events, one need only look at how global events would have to be implemented. Since Events using C# are linked to a specific class, this would essentially mean that the EventManager class would need to be setup for every single event the engine is capable of running. What this basically has accomplished is to couple a single class into the entire workings of an engine, this makes the engine difficult to extend and modify at a later time since the design would be practically hardcoded into it.

Improvements of events As of now, our events are not tied to being `EntityEvent`s or `EnvironmentEvent`s like actions are, however this might have been a wrong move on our part. The problem is that the user of the engine might be unclear as to which is what, currently the difference lies in the name convention used for events. For instance, it is clear from the name that the `EntityMovedEvent` can be tied to a specific entity. In the case of the `AddedEntityToEngineEvent`, however, there is some ambiguity, as the event is clearly speaking about a single entity, but as the entity is only just added it would have been impossible for any trigger to be registered to it. If one was to make improvements to the event design this would be one change that was worth looking into.

Triggers The trigger design came about as a necessity for providing a way for the user to easily design reactions to a given event. The trigger design is very minimalistic except for the fact that it has a condition. We designed it with the condition because we wanted it to be obvious how unwanted events should be handled. Furthermore it also helps to separate the code containing the condition and the code containing the action itself, allowing for more readable code.

Another way the triggers could have been designed would be if the user simply registered lambda functions (anonymous function), this would help reduce the amount of classes a user should know and understand. However we preferred to encapsulate this into what we call the Trigger, since we wanted to have the ability to expand the capability of the trigger at a later time.

In short, triggers are a simple design that gives the engine user a lot of flexibility.

Summary

Events and triggers might be a hassle to setup and design, but in return they provide the engine with a lot of flexibility. Without Events the engine would suffer greatly and all actions would be required to be bogged down with a lot of extra logic. This would not only remove the modularity of the engine but also make using the engine more error-prone.

5.2.4 Actions

As we already went through what actions can be used for, this section will instead focus on the idea behind actions, and how we implemented them. It will furthermore cover the entire life span of an action object.

Explanation

An action – or `XmasAction,` as it is called in our engine – is a class which provides an API for performing state changes inside the engine, while also ensuring that only one action at a time is being executed.

As can be seen in fig. 5.2 it starts with the `XmasModel` running an endless loop that tells the `ActionManager` to execute all newly queued actions. The `ActionManager` then takes all the actions from a thread safe list and places them in a local list. After this, each action is executed individually, putting the action that is being executed in a running state, this state will not change before the actions `Completed` method is called. Once an action has been properly executed, it will be changed to a completed state and will be properly disposed of. When the last action has been executed by the `ActionManager`, the call to `ExecuteActions` returns and `XmasModel` will put the thread in a waiting state. The `XmasModel` will remain in a waiting state until a new action has been

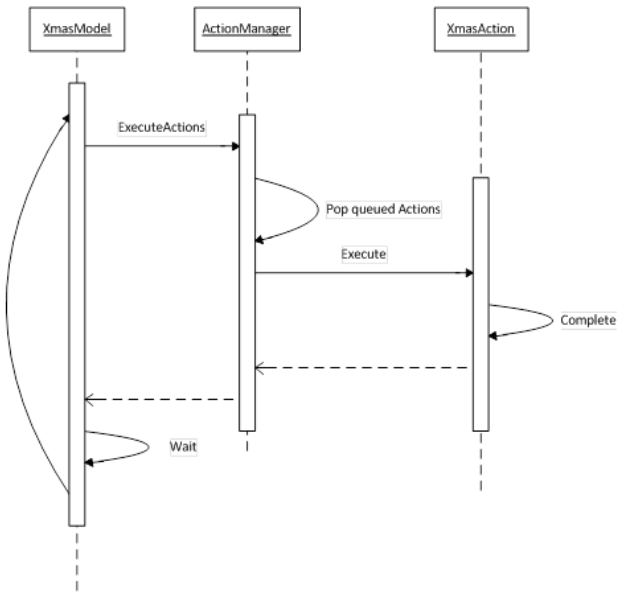


Figure 5.2: A sequence diagram describing the execution of an action.

placed on the queue; this prevents it from busy waiting when no actions are to be executed.

Considerations

The way that action completion is designed might seem tedious in that it has to call the special method `Completed` on each action. However it is quite necessary as the completion of the execute method call does not guarantee that a method is completed, for instance in the case of non-instantanious actions, as explained in the example below.

Consider the action of moving from one place to another. In this case the move action would need to set a delay before the actual move, to give the idea that the move action had a speed. As we can't halt other actions during this time it is paramount that the `Execute` method is released so that other actions can be executed during this period.

This is also how the move action is designed in our reference implementation, the algorithm is as follows

1. The move action is put on the queue
2. The move action sets up a timer on a different thread and finishes its execution
3. The timer is fired after a given time, and places a new action on the queue
4. The new action performs the actual move, and calls the `Completed` method of its parent Action (the `MoveAction`)

As one can see, the problem in this design is the redundancy created by having to call the method `Completed` on every designed action execution. This might not seem like a problem but it is problematic in a few ways. First and foremost it adds complexity in usage of the engine, a person with no knowledge of using the engine would not intuitively deduce the correct way to make and use actions. Thus it creates a second problem: there is no way to determine if an action is correctly constructed during compile time. This means bugs will naturally accumulate during extended use, even if a user has experience and foreknowledge forgetting even for a single action can be crucial. This is because running actions use resources and if never completed the resources of the actions are never released. For instance let us assume the `MoveAction Completed` method is never called, the result of this is that it is stored in the `ActionManager`

as `Running`. Now let us assume that this move action is continuously being executed by hundreds if not thousands of agents. As each action is never released the memory stored for each action is never released and an unintentional memory leak is thus created.

Another way we could have chosen to implement the action completion process, is the usage of child action. Imagine if an action could generate new actions that were linked with it, thus the completion of an action would be tied to the fact that all its child actions had been executed and not the arbitrary call of a `Complete` method. This could undoubtedly provide new problems to overcome and as such we have not fully followed this path, however given more time to study the consequences of this design would reveal whether or not this is a better design.

Summary

A lot of the considerations when designing the action all comes down to the reliance on user to clean up the Action, which is generally not good from a design perspective; it is always preferable that used data is cleaned up automatically when it is out of scope. However it is not all bad as this design does guarantee a flexible usage of the actions; it provides more control to the user which might give the user abilities to do certain things which would otherwise be denied within the engine. This is also why this design method was chosen as our philosophy in the engine design was to minimize limitations as much as possible while still providing the features we thought necessary to fulfill the engine's goal.

5.3 Agent Controller

The agent controller is designed specifically to be able to accommodate all types of APL. This means that a lot of special care had to be taken in order for us to impose as few restrictions as possible. This section will focus on the different designs we went through and why we eventually landed on the design we have now.

Explanation

The `AgentManager` is designed to run separately from the engine's model thread, which means it has the ability to take all the time needed to properly connect

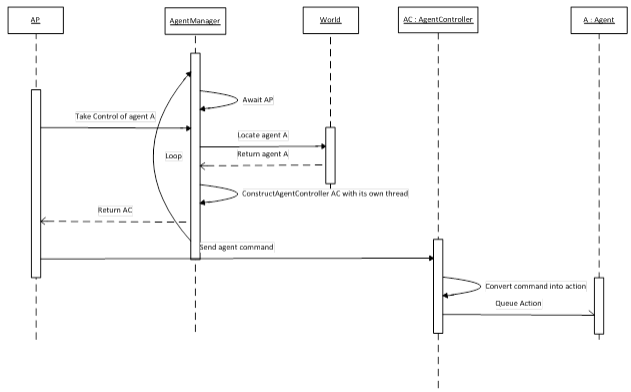


Figure 5.3: This image details exactly how an `AgentManager` Processes incoming requests from an outside AP

to an outside AP, same goes for the `AgentController`. This means that when an `AgentManager` generates a new `AgentController` to be used by the AP, it also generates a new thread which the `AgentController` is executed on. In the System Features section we covered how `AgentController`s are used. Here we will elaborate on the exact process. In fig. 5.3, a sequence diagram is shown that looks familiar to the one shown in the system features. However there are a few key differences. First, this sequence diagram shows the complete life cycle of an `AgentManager`, since the `AgentManager` is running on its own thread it does not care about blocking until work needs to be done and the only kind of work it is responsible for is ensure that `AgentController`s are generated for APs in need of them. Second, it also details that `AgentController`s are in fact generated by the `AgentManager` with its own thread.

Considerations

The `AgentManager` went through many design iterations in order to arrive at its present state. Originally, the `AgentManager` was called `AgentServer`. The reason was that for another language to interface with the language of the engine – C# – there must be a universal way of connecting the two languages. A way in which practically no languages was prohibited from interacting, and as we thought such a way could only be achieved through a TCP connecting since the protocol for TPC connections is very old and as such usable in most languages by far. While it is true that probably almost all languages do require a TCP connecting in order for them to work with our engine, it is not true for

languages that the engine understand, considering that all the .NET platform languages works together very well. For example, you could use the functional programming language F#, which also runs on the .NET platform. As such, if we imposed that all `AgentManager`s are `AgentServer`s, it would be required to setup a server just for using a language which the engine already understands. There is also that our goal for the engine was to be general as possible and since an `AgentServer` is more restrictive than an `AgentManger`, then `AgentManager` is the design we went with.

Summary

`AgentManager` and `AgentController` is designed as a framework for making an interface between an APL and the engine. They are intentionally made very lightweight so that they do not prohibit any special requirements of any given APL.

5.4 View

As the view our engine provides is only a framework for making an actual view, it limits what can be said about its implementation. What this section will focus on is why we chose to design the view in this manner and how we provide ways to ensure that the view can be executed on a different thread while not being affected by its problems.

Design

The view design for the engine was never meant to be an actual view, this would limit the potential of what could be done so we are rather content with not providing more than the skeleton for making a proper view. The idea is that the actual implementation of a view should be part of some extension to make a view that displays graphics or a view that shows a console, it should never be a core part of the engine. The core engine should only provide what all views need, this means that if just a single view is restricted by our design then our design is flawed.

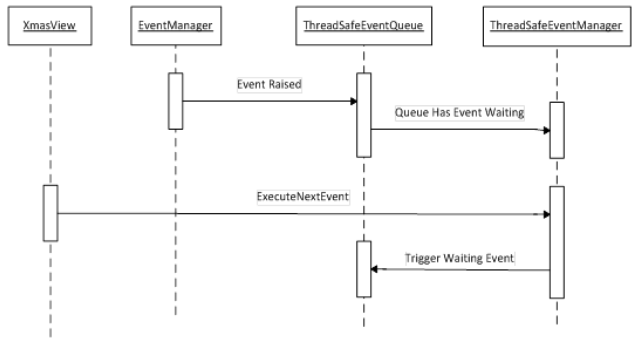


Figure 5.4: Sequence diagram show how events triggered on the model is stored and put on hold until the view thread is able to process them

Thread Safety

One thing all views have in common is the dangers of having code that is not thread safe, by having two threads run through the same address space at the same time, the risk of a race condition or deadlock is very high. This makes programming a view rather difficult. To combat this problem, we came up with the `ThreadSafeEventManager` and the `ThreadSafeEventQueue`. These classes both assist with ensuring that the model thread is never involved in the view thread's business. The way the `ThreadSafeEventManager` works is by storing all events triggered by the EventManager of the model, the events data are all kept safe and the order in which the events was triggered is also kept. The idea is that when the view thread is not performing any actions, such as when it is in sleep mode between a draw update, instead of sleeping it will call the `ThreadSafeEventManager` and tell it to begin executing. The process works by running the `ThreadSafeEventQueue` that had one of its events trigger and tell it to execute. When all `ThreadSafeEventQueue` are empty then that mean that there are no longer any events waiting to be executed on the view thread. Since views are only interested in seeing the changes to the world and not how the changes came about, then that means that the views only need access to the events and not the actions. To see a sequence diagram of this process look at fig. 5.4.

Summary

The view design is mostly focused on ensuring that the user of the engine should deal with as few threading problems as possible as such we have developed two classes `ThreadSafeEventQueue` and `ThreadSafeEventManager` these both make it possible for the view to trigger events when the thread is free from other duties, instead of relying on the model thread to also handle view event updates.

5.5 Engine Extensions

5.5.1 Tile Extension

The tiled environment is quite elementary, as it is basically a two-dimensional array with some extra arithmetic to change the coordinate system, as described in System Features. The actions and events are also very straight-forward, as they function as described in their own sections. The most interesting part of the tile extension is thus the way we handle vision, which we will describe here.

Vision

We say that the tile t_2 is visible from another tile t_1 if at least one corner of t_1 connects to at least three corners of t_2. If t_2 is vision blocking, only two corners of t_2 need be connected to. In figure 5.5 we have shown some examples of connecting corners. Next, we will explain what it means for two corners to connect.

We say that a corner c_1 on a tile t_1 connects to a corner c_2 of another tile t_2 if a straight line can be traced from c_1 to c_2 without intersecting with a tile that is blocking the line. In the tile extension, a tile is blocking the line if it contains an entity that is vision blocking with respect to the entity looking from t_1. This presents a problem with a line whose vector has an x or y component equal to 0. As such a line never intersects with any tiles (it only passes between them), it will always connect with the rules stated above, even if all tiles it passes are vision blocking. Thus, we say that a line which only extends in the x or y direction does not connect if it passes between two vision blocking tiles.

Note that in fig. 5.5, the line from the SE corner of `A` to the SW corner of $\texttt{T}_3$ does not connect, as it passes between two vision blocking tiles, as described

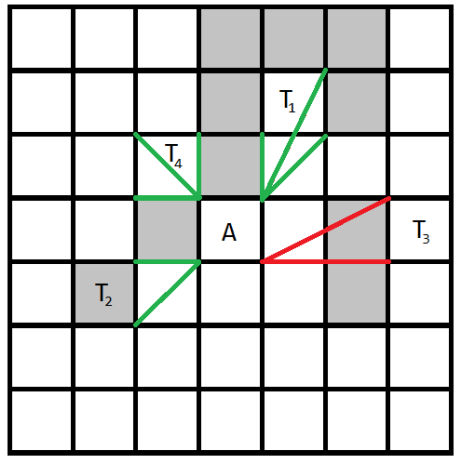


Figure 5.5: In this figure we see the visibility of selected tiles $\texttt{T}_1 \ldots \texttt{T}_4$with regards to the agent in position `A`. The light gray squares represent vision blocking tiles, and all other tiles are see-through. Lines have been drawn from the corners of the agent's tile to the corners of the selected tiles. Green lines signifies that the two corners connect, whereas red lines means they are not connected. Tiles $\texttt{T}_1$, $\texttt{T}_2$ and $\texttt{T}_4$ are visible to the agent, while $\texttt{T}_3$ is obscured.

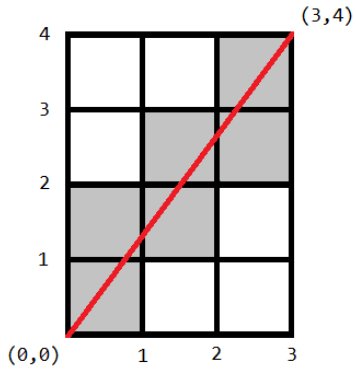


Figure 5.6: A line described by the vector $v = (3, 4)$ with slope $s = \frac{v_y}{v_x} = \frac{4}{3}$. Squares in light gray are tiles the line intersect with.

above. Also note that T_4 should, intuitively, be obscured to the agent; we should have a rule saying that when a line intersects with a corner, it should be blocked if both of the two tiles that are connected to the corner, and is not intersected by the line, are vision blocking.

In its most simple form, the vision algorithm iterates over all the tiles in the agent's visible range, and returns a collection containing just those satisfying the above condition, as shown in the pseudo code below:

```
Method Vision
        takes an Agent A,
        returns a collection of Tiles
    Tiles : Collection of Tiles
    for each Tile T in A's visible range:
        if T.isVisionBlocking(A):
            if any one corner of A connects with any two corners of T:
                add T to Tiles
        else:
            if any one corner of A connects with any three corners of T:
                add T to Tiles
    return T
```

The interesting part of the algorithm is this: how do we determine whether two corners connect? That is, how do we find all the tiles a line from one corner to another passes through?

To illustrate the problem, consider the line depicted in fig. 5.6. First, we find the slope of the line as $s = \frac{v_y}{v_x}$, where v is the vector describing the line. If we consider the line's equation ($y = s \cdot x$), as it is depicted in the figure, we can see that the line segment decribed by the vector between the points $(x_0, s \cdot x_0)$ and $(x_0 + 1, s \cdot (x_0 + 1))$ (where $x_0 \in \mathbb{N}$) crosses the tiles between $x = x_0$ and $x = x_0 + 1$ on the x-axis and $y = \lfloor s \cdot x_0 \rfloor$ and $y = \lceil s \cdot (x_0 + 1) \rceil$ on the y axis. This can be repeated for every x_0 in the range $0 \ldots v_x$ to obtain the complete collection of intersected tiles.

In the `Vision` class, the `walkAlongVector(Vector v)` method performs the above operations, with some modifications to handle vectors with negative components properly. It is called by the `ConnectCorners(Point, Point)` method, which calculates the vector `v`. `WalkAlongVector` uses the `yield return` keyword to return each tile at a time, so lines are only drawn until a blocking one is encountered.

5.5.2 EIS Extension

EIS support in engine is provided with a special `AgentController` and `AgentManager` class, along with a specially designed java EIS environment jar file. This section will go through how the implementation works and how we connect to the EIS environment.

The EIS environment in java and the agent controller on the engine is connected through a TCP connection. They communicate with each other with XML as a markup language for the data they transmit.

Fig. 5.7 shows the setup between EIS and the agent controller.

Although the EIS environment and the agent controller sends all data in form of XML data there is one difference and that is all XML nodes are packaged into packages of a certain size and the size is sent before the xml data, as can be seen in fig. 5.8.

This was done to ensure that the agent controller at all times knew how much data was to be transmitted, thus giving it the right to deny packages if they were over a certain size. In our current implementation however no package size is denied.

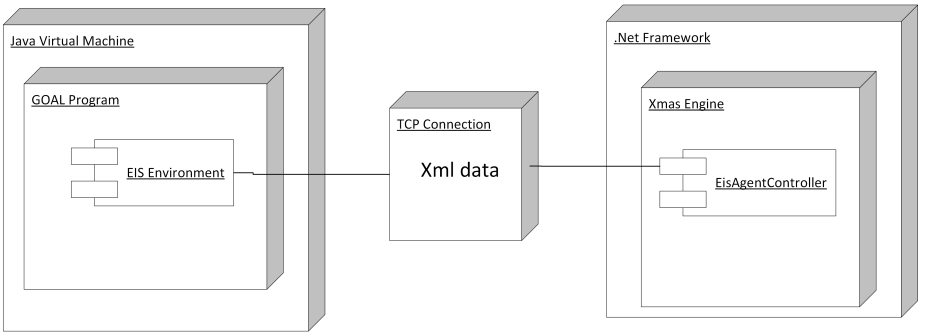


Figure 5.7: An illustration of the connection between the EIS environment and the agent controller.

Figure 5.8: Image of the data sent between the EIS environment and the agent controller.

Engine Side of EIS Support

In the project: `XmasEngineExtensions` we provide the following two classes:

`EISAgentController:` this class is responsible for converting xml data from the EIS environment into actions that can be queued to the agent it controls. And also for converting percepts the agent it controls receives into XML data that can be sent to the EIS environment.

`EISAgentServer` creates a TCP server. All EIS environments that wish to connect to it must make a TCP client call. Once a connection is established, the agent server will construct an `EISAgentController` object, that object will take over all further duties of comunication.

How the EISAgentServer works

The server manages the agent controllers and it also manages the connection creation between an EIS environment and an `EisAgentController`.

In Fig. 5.9 we show how an EIS environment connects to an agent server and how the agent server handles the connection. The connection works by the EIS environment making a TCP connection request. The agent server then responds by constructing the agent controller (and give it its own thread). Once the agent controller is constructed the agent server is no longer responsible for handling that connection and leaves it up to the agent controller to find out what the EIS environment wants. This is basically to connect to a given agent whom it knows by name, and start sending it actions.

How the EisAgentController works

The EIS agent controller's job is to ensure that all demands made by an EIS environment are met. This is done by receiving actions in XML data form and convert the data into Xmas Actions. These actions are then queued onto an agent.

In fig. 5.10, it is shown how xml data received is converted by the controller and then sent to the agent inside the engine. Percepts are only sent if they are updated. In this case the action was to retrieve percepts. If the action had instead been to move an agent, then no percepts would be sent by the controller.

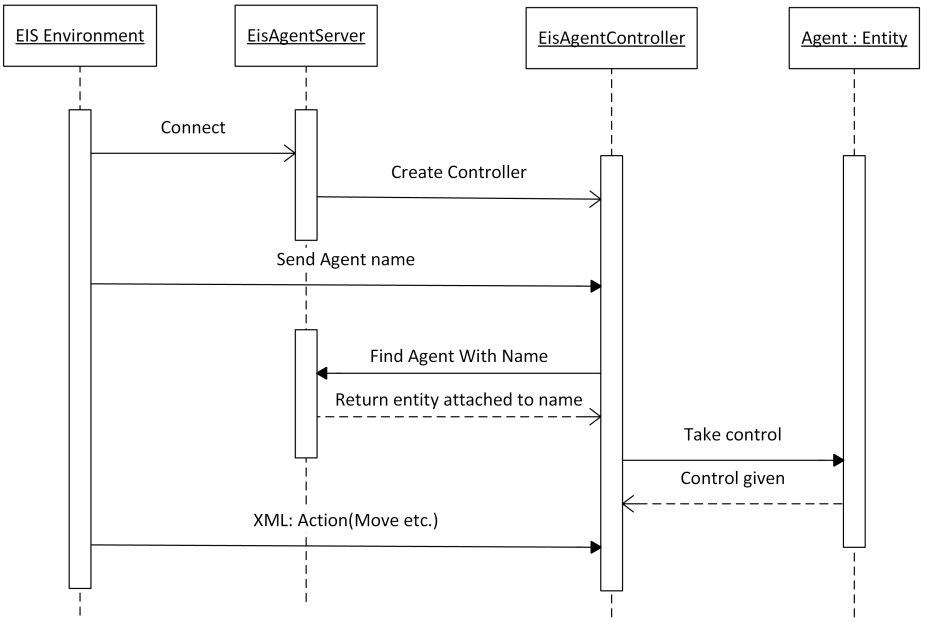


Figure 5.9: A sequence diagram of an EIS environment connecting to the engine through an EisAgentServer.

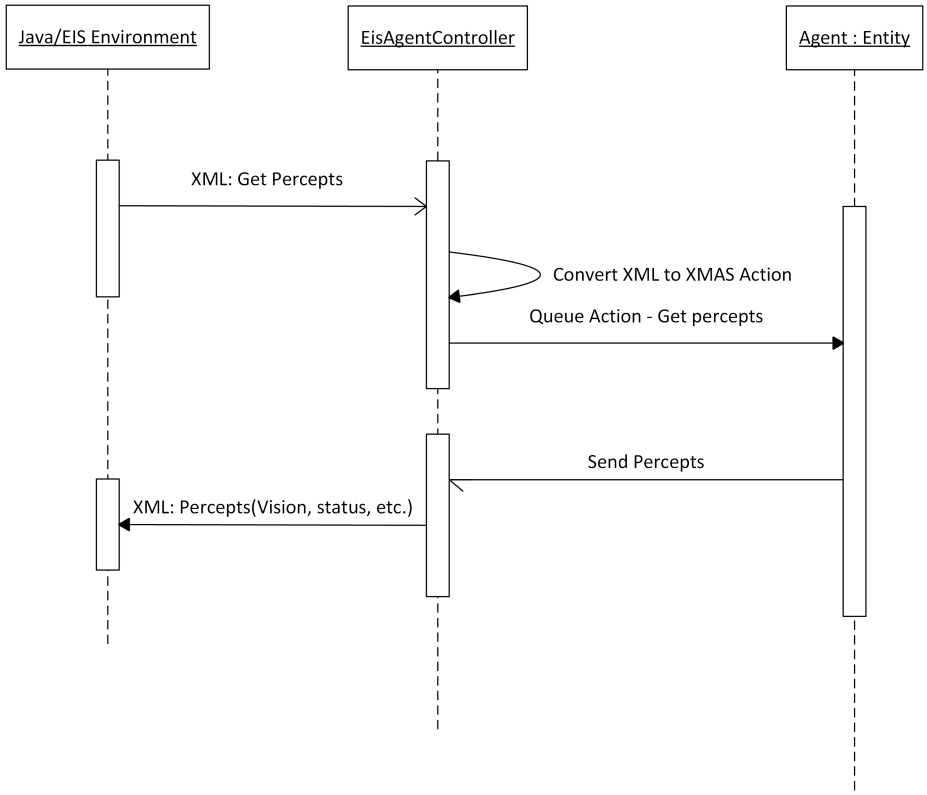


Figure 5.10: A sequence diagram showing how XML data from EIS environment are converted into Xmas data.

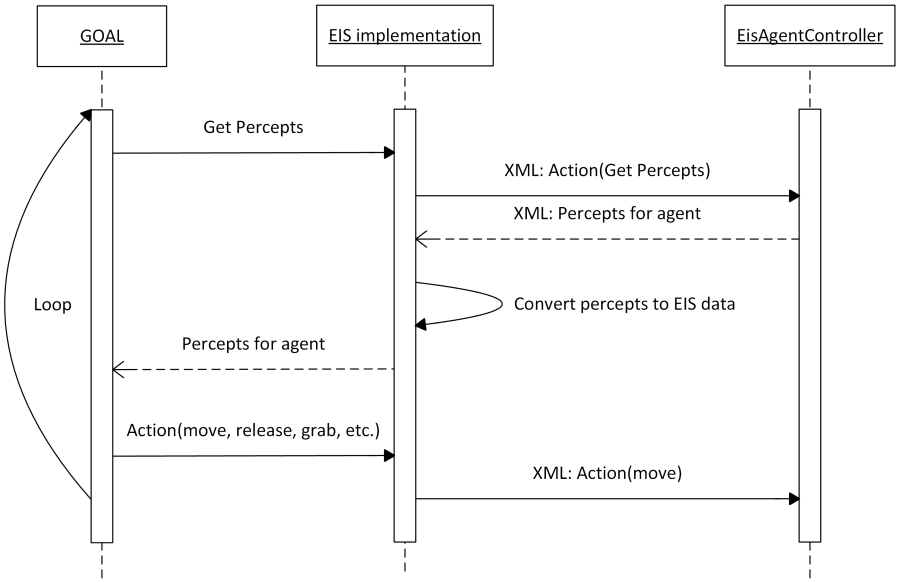


Figure 5.11: This diagram shows how the communication between goal and the EIS environment works.

EIS Environment

The EIS environment is designed to setup an interface between an APL and an environment. The way we use EIS is by making it a hollow link between our engine and the APL (such as GOAL). Thus the EIS environment implementation we make must be able to provide communication between the APL and our engine. The way we have done this is by making the environment convert all the XML data it receives from Xmas into IILang objects, which is an object tree implemented in EIS for recursively representing percepts and actions in native Java code as well as converting them to and from XML and prolog statements. It can then use the IILang objects in its internal GOAL interface.

An example of sequence for the EIS environment communicating with an APL (such as GOAL) can be seen on fig. 5.11. The basic idea is that the goal program sends commands directly to the EIS environment we have implemented and then we ensure that those commands are fulfilled by transmitting them over to the `EisAgentController` through a TCP connection.

Considerations

The design of interfacing with GOAL was originally what the engine design was mostly focused on; as such there have been lots of different approaches to this interfacing that we have gone through. One approach was to connect the EIS environment using J# which could be converted into C# byte code; this would be a lot faster than our current approach since XML data wastes a lot of space by encapsulating every bit of information. However J# is an old language and we wanted to ensure that we did not run into too many complications under development as such we chose our current approach since the real time transmission of data is not as important as the idea of it, for this project in particular.

Summary

EIS is an interface for designing environments in java that connects to EIS supported APLs, we use this environment to develop an environment that is simply a TCP connection between the APL(in our case goal) and our engine. The design provides the necessary features to the engine, but the design could have been more optimized by using a more compact way of sending data, since sending data as XML nodes takes up a lot of space since XML requires all data to be encapsulated by it.

5.6 Reference Implementation

The reference implementation relies heavily upon the extensions that are attached as part of the engine.

The extensions the reference implementation uses are:

Logger Extension this is used to log all actions that occur inside the engine and to log any errors that might also occur.

EIS Extension this extension is used to connect the reference implementation to our goal program for the agents inside the reference implementation.

Tile World Extension the reference implementation uses a Tile based world as such it directly uses the Tile World Extension that provides just this functionality.

The reference implementation as such only provides:

- Actions specific to the reference implementation(Grabbing/releasing packages)
- Entities specific to the reference implementation(Walls, Player, etc.)
- Percepts and modules specific to the reference implementation(Holding package percept)
- A view in console form
- A Goal program
- A way to control an agent with keyboard

5.6.1 The Console View

The console view is designed to draw the screen at a specific frame rate. When the console view does not draw it will instead update all view data it has stored.

To change view data in a view, an event must be fired from the model, however since the model is operating on a different thread than the view, the view must ensure no concurrency errors. This is done by using the `ThreadSafeEventMananger`, as explained in 5.4.

The console view works by drawing the screen, then – if it has time left before the next drawing is scheduled – it will execute a single event on the `ThreadSafeEventManager`. The view will continue this process until either there are no events left to be executed or the time is up and it is time for it to perform the next drawing of the screen. In fig. 5.12 an illustration of this process is shown.

This provides the reference implementation with a very quickly updated view as no time is wasted on the thread, since the view will continue to update even when it is not drawing. Furthermore, by updating the view data in a separate thread the engine core does not use its computation power on handling this, which makes the engine more efficient overall.

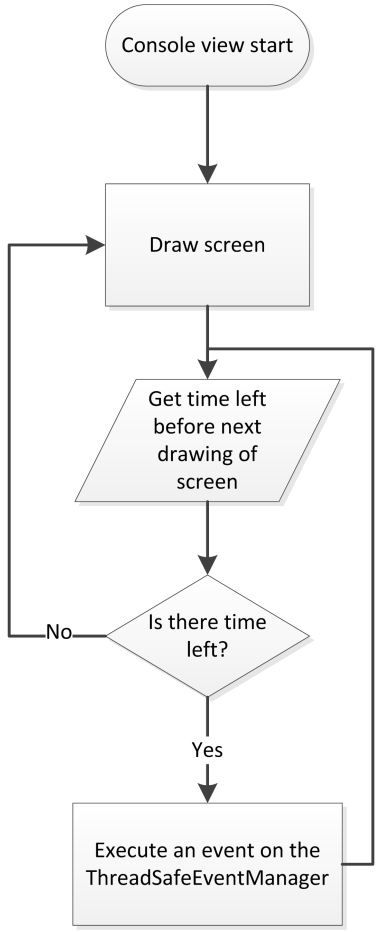


Figure 5.12: the sequence of the console view drawing process

5.6.2 GOAL Program Implementation

The GOAL program is designed to work directly with our reference implementation, as it is just a showcase of what such a program might look like. It will make assumptions about how the reference implementation works. For instance, it will assume that there are entities called walls that blocks movement.

To see the commented source code of our GOAL program, see appendix D.2.

5.6.2.1 Agent Decision

A full flow chart of the goal program decision chart can be found in appendix D.1.

As can be seen from the flow chart, the agent will try to find packages and bring them to a dropzone, if no such packages can be found or if no dropzone is found, the agent will start exploring the entire world.

The goal program operates with a few different notions;

Street The first notion is the notion of streets. A tile is a street if it contains no wall types such as normal walls or impassableWalls (map boundary walls). This means that the agent can move on this tile.

Route When an agent decides to move to a specific tile, it will perform an `A*` search to find the shortest path to that tile. The `A*` search (which is called `As` in the GOAL / prolog code found in appendix D.2) returns a route represented by a list of tiles the agent should follow to reach the desired tile. Whenever the agent have no tasks of higher priority (such as grabbing a package if it is standing on one), it will pop a tile from the route and move onto it.

Explored the agent's goal is to eventually have all tiles explored as this means that it can determine whether all packages have been picked up. The agent determines that a tile has been explored if it has seen all its adjacent tiles (fig. 5.13 shows an image of this). No matter how much the agent explores, whenever it uncovers a previously unexplored tile, it will see new tiles to explore, unless the uncovered tile is a wall. This will continue until a wall has been reached on all its paths. Uncovering a new tiles works much like putting a carrot in front of a mule; it will always try to catch up to the carrot, just as the agent will try to uncover unexplored tiles.

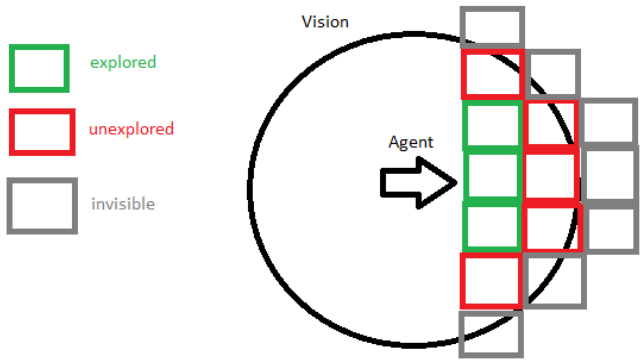


Figure 5.13: An image of an agent's vision and which it would determine to be explored

Summary

The reference implementation was designed as a reference for all the features of the engine, as such it made heavy use of the extensions that we implemented. This section only covered the view and the goal program in details. This is because most of the reference implementation consists of either declaring new agent/entity types or wiring all the extensions together. As such, there was almost no business logic involved which makes them rather uninteresting to explain in detail.

One part that the reference implementation does not cover which could have been interesting was the notion of linked modules as explained in 4.2.1. This could have been used in the reference implementation but we did not choose to do so.

Overall the design of the reference implementation is very solid and fulfills the goals we had for it, which were to be a showcase for our engine.

Chapter 6

Testing

When developing a large project, one of the most difficult aspects of the process can be proper testing. This section will go through what kinds of tests we performed to ensure that our engine works correctly.

6.1 Testing the Engine

Unit Tests: The location of the unit tests for the engine and all its depencies can be found in the source code in the `XmasEngine_Test` and `JSLibrary_Test` projects.

Since our engine by itself is not meant to be executed, but rather is meant to have some of its components implemented first, the only kinds of tests that can be performed on the engine are unit tests. The unit tests that are most important to the core functionality of the engine are the tests concerning the `EventManager` and the `ActionManager`, since these tests shows that the action, trigger and event functionality correctly works. However, as we created the engine using a TDD approach we should have tests for almost all classes with business logic contained in them.

One component proved impossible to properly unit test and that was the EIS agent controller, since this component required a connection to properly understand its errors. To perform this test we designed a simple component/functional test, this test is not meant to be executed as part of the unit tests. The test is designed to be executed with an debugger so that the programmer can easily follow if any errors occur during the run.

6.2 Testing the Reference Implementation

The testing of our reference implementation hinges on the fact that we assume the engine works correctly, thus it is the job of engine-tests to ensure correctness and not our reference implementation. As the reference implementation's complicated logic was located as part of the extensions, which were already tested as part of testing the engine, the only testing that the reference implementation needed was testing of the GOAL program which is controlling the agent.

The goal program was tested by taking out individual parts of it such as its path finding, and carefully tested and debugged in the SWI-prolog program. When all parts worked correctly we moved them to the actual goal program and then made a larger scale test of the goal program using the fully running engine. Once the agent correctly had located all packages and stopped as we wanted, we concluded that the reference implementation was working as it should.

Chapter 7

Results and Comparisons

In this section we will discuss the major considerations we faced during the project, as well as the choices we took in accomplishing our goals, and how we could have otherwise reached them.

7.1 Generality of the engine

One of the major goals of this project was to make the engine as general as possible. This includes the ability for the designer to implement any kind of environment, displayed any way he wants, and controlled by whatever APL he would like to use. In this section we will discuss how these three vital parts of our engine lives up to this goal.

For the engine to be general, it must have the ability to adapt to any needed situation. The only restriction is that these situations are based in multi-agent systems. Other than that, nearly any situation should be coverable by the engine. For instance if one wishes to use the engine to make a computer game, then the view must be able to support a graphical display and the world of the engine must have the ability to be changed to a 3d-world. But if instead one wishes to make an engine for searching documents for spelling errors, then

the world should be extendable to a text-document. Furthermore to be general also means that the engine should be used to work with any other APL, so regardless if the APL is GOAL, Jason or F# the engine should have the ability to be adapted into working with either of those languages.

In many cases, the shortest path to a general system is removing restrictions. Unfortunately, features are often restrictive in nature; the most general system of all is one that is completely featureless. Thus, it is often a trade-off between features and generality. In our engine design idea we try to overcome this by making all features as extensions or optional thus not restricting the core engine.

Model

For the model to be general, it should be capable of representing a world as any possible data structure. Additionally, the objects inhabiting it should be as general as possible, allowing them to be defined in a way that makes sense in the context of the world.

We have accomplished this by imposing as few restrictions as possible on these objects. For example, as described in section 4.2, a world in the engine has no data associated with it by default, leaving the modelling of it completely in the hands of the designer. The only restriction on the world is the idea that all entities should have a postition in it, although the position object is completely general itself.

Initially, we toyed with the idea of equipping the world with a graph as the default representation of the environment, since it is a very general data structure in the sense that it can be used to describe other data structures. The problem with this approach is that a graph may not be the best representation of any given world. In the case of a tile based world, for example, a two dimensional array is more feasible, since this is its natural representation. Ultimately, we chose to impose no restrictions on the data structure used, and instead rely on the use of extensions to model environments.

Interfacing with APLs

One of the major problems in designing an interface that works with different APLs is that: the order in which they, queue actions and queries percepts may be different from APL to APL. In effect, they do not share a common execution protocol. This means that we cannot provide a general method for

communicating with any AP. Instead, as with other parts of the engine, the intent is to allow for extensions capaple of interfacing with different APLs in any way they see fit.

It could be argued that using the notions of percepts and actions serves to limit the universality of the engine. These are, however, general concepts for interacting with an intelligent agent. They are basically the input and output of the agent; it perceives the state of the world, and produces an action based on this information. As such, they are essential to interacting with an agent, and incorporated in all agent programming languages we are aware of.

View

For the view to be considered general it is paramount that the design of the view is not being restricted in anyway, this is done by keeping anything in the view very minimalistic. By minimalistic we mean that the view only provides about four classes and they only provide a tiny portion of business logic. If we had narrowed down how exactly a view should be designed such as requiring a frame for which the view is projected on. This might have made implementations of the view easier as tools to draw on frames could be pre-implemented into the engine, but in turn restricted the view from being able to become other types of view. We did not want to do this since we think that restrictions should be non-existent .However this also poses a potential problem in that it is so minimalistic that we barely provide anything for the user, and leave the user to the task of making the view by themself.

Solving the Problems of Generality

As evident when discussing how to make the engine general and how to make it work with as many situations as possible, there is a problem that the workload for the user gets increased. This is because whenever we remove something from the engine in order to ensure that we impose no restrictions, we run the risk of removing something that made the life of the user easier, as they would not have to re-implement it themself. To combat this problem, we moved everything that added value to the engine but imposed a restriction on it to the Engine Extensions project. The idea would be that while the extensions was not part of the core engine, they would be part of what we delivered with the engine. We saw this as the best of both worlds. Not only do we ensure that the engine is not being restricted, but at the same time if the user did not mind some restrictions, then they would be able to find a suitable extension among the

ones we provide. As of now the only extensions we have are those needed for the reference implementation, but our long term plan would be to add more extensions if possible.

7.2 Model View Controller Design Pattern

The model view controller design pattern is one of the older design patterns within software design; its purpose is to ensure that the developer does not deal with multiple issues at once, and instead is able to focus on one task of the project at a time. We chose to base our engine on the MVC pattern because we also do not want the user of the engine to be tasked with multiple issues at once. Without the MVC pattern, the user could be confused about how for instance they should design a controller for an APL, and perhaps they would mistakenly design it tailored to specific actions. If the user did this, they would have to write new actions to perform the same task for each new APL they encountered, which would increase code redundancy. As the developers of this engine we wanted to ensure those kinds of mistakes do not happen. The way we enforce this is by forcing the MVC pattern. By forcing the MVC pattern we force the user to think about how they should construct the implementation of the engine's abstract classes. However since it is only a pattern, the user can still make bad design decisions as we impose no restrictions.

7.3 Choice of Technologies

The XMAS Engine

We have chosen to implement the engine in C#, which runs on the .NET platform. While Java has a strong presence in multi-agent system development – as it is used by established APLs such as Jason, as well as the EIS standard – we have a subjective preference for C#. In general, C# is a newer and more modern programming language with better facilities for writing comprehensive and maintainable code, and provides some features usually only found in functional programming languages. Additionally, .NET code can be executed across platforms, thanks to the Mono project[1], although the newest version of .NET (v4.5) has not been ported at the time of this writing. Although developing our reference implementation would have been simpler had we used Java, opting

[1] `http://www.mono-project.com`

out of this in favor of C# gave us the opportunity to test how well our engine interfaced with programs not written in the same language.

Reference Implementation

As our reference implementation was developed to showcase and test our engine, we aimed to implement it using the most commonly used agent programming language. Since EIS can be interfaced with many different APLs, this seemed like the obvious choice. If the engine could be shown to work with EIS, any APL supported by EIS would work by extension. Initially, we considered writing a J# (.NET bindings for the Java Language) module, which would work natively with both our engine written in C#, and the EIS implementation written in Java. However, we felt that this would remove the difficulties of communicating with an entirely different platform.This difficulty is important to overcome, as it readies the engine for future problems of a similar kind.

7.4 Comparison to other Environment Construction Tools

In this section, we will compare the XMAS engine to other frameworks that can be used to construct and manage MAS environments. In particular, we will consider CArtAgO[2] (*C*ommon "*Art*ifacts for *Ag*ents" *O*pen framework, henceforth referred to as Cartago), which have been used in several projects, and as a part of the *JaCaMo*[3] (*Ja*son, *Ca*rtago, *Mo*ise) project, which provides a complete framework for multi-agent systems consisting of the Jason MAPL (multi-agent programming language), the Cartago environment constructions API, and the Moise organizational system. We will also compare our engine to using plain EIS.

[2] `http://cartago.sourceforge.net/`
[3] `http://jacamo.sourceforge.net/`

7.4.1 Cartago

Agents and Artifacts

The Cartago framework uses the `A&A` (Agents and Artifacts) approach to designing environments. In the following, we will provide an overview of this model, which is presented in [ORV08].

In the A&A meta-model, a collection of computational entities called *artifacts* constitutes the environment. They are computational entities in the sense that they are meant to provide functionality exploitable by the agents, and can have a state and business logic, but are not meant to act autonomously. In fact, instead of agents having predefined actions that manipulates the state of the world when executed, they are aware of a collection of artifacts, which each provide a number of operations the agents can perform on them. The artifacts also provides percepts to the agents, and are as such the agents' means of interacting with the world.

Artifacts can not only be used to model objects with a physical presence in the world (such as analogues to the packages and dropzones in our reference implementation), but also more abstract concepts, such as control flow objects. For example, a communication artifact could be created, through which several agents could talk and listen, through operations and percepts, respectively. Since agents can create and destroy artifacts at will, such communication channels are easy to spawn in an ad hoc manner.

The A&A meta-model, as described in [ORV08], is to some extent based on the way humans interact in a working environments, as it draws on research in fields such as organisational sciences and anthropology. This lead to the introduction of artifacts as tools, service providers and communication devices, since they better describe such an environment. In general, the A&A approach focuses on incorporating these concepts as an integral part of the environment, so as to make it a functional and reactive part of a multi-agent system. This is in contrast to classical MAS engineering, where the environment is typically defined as a more static structure which agents can act in and retreive percepts from.

Cartago Implementation

The Cartago framework is implemented in Java and can natively connect to the Jason APL. Here, we will explain how Cartago implements the A&A meta-

model. A more thorough explanation can be found in [RPV11].

Agents In Cartago, agent programs are connected to agent bodies, which are – in that respect – conceptually similar to agents in the XMAS engine, as they represent a vessel for the agent in the environment, but no agent logic. In keeping with the A&A approach explained above, the agent API allows for creating and deleting artifacts, as well as executing operations on artifacts and retreiving percepts from them.

Sensing For handling perception, Cartago provides the concept of *sensors*. An agent contains a set of sensors, each collecting percepts from an artifact. The `sense` method of an agent takes a sensor as input and returns a percept gathered by it, whereafter the percept is removed from the sensor. The sensors can be overridden by the designer to – for example – control in what order it should return its contained percepts. A sensor is connected to an artifact via the `focus` method. Sensory inputs can be filtered, so that a sensor only picks up percepts matching a user-defined pattern.

Artifacts Artifacts specify operations that agents can execute as described by the A&A meta-model. Artifacts can generate events, which can be gathered by any connected agent sensors. Artifacts can describe how they are meant to be used, ie. what operations thay have and how they should be called. When an agent executes an operation on an artifact, a boolean value is immediately returned, signifying either success or failure. The calling agent can give a sensor as an argument to the method invocation, which will gather any percepts that the artifact generates as a result of executing the operation.

Agents and Artifacts in XMAS

The main purpose of the Cartago project is to provide a framework for designing MASs using the A&A meta-model. While that have not been the goal of the XMAS engine, it is general enough to support this approach, especially since the engine allows entities to incorporate state and business logic through entity modules. Below, we have described how artifacts, agents and perception would be implemented using the engine:

Agents would be XMAS agents, with a module for creating, destroying and containing (references to) artifacts, as well as a module for each sensor. The `sense` action used in the Cartago API is not entirely equivalent to our generic `getAllPercepts` action, as it only retrieves one percept from one specific sensor, but such an action could easily be implemented.

Artifacts would be represented as entities with a module for each operation the artifact provides. When new stimuli, ie. new percepts, would be available, an XMAS event would be raised.

Sensing The agents' sensor modules would be connected to the artifact entities by registering triggers on them, wchich would subscribe to the events raised by the artifacts. By using trigger conditions (cf. section 4.3), the percepts could be filtered as with Cartago sensors. The modules in XMAS already provides means for being queried for collections of percepts, so this functionality could be used to let them return all the percepts that have become available to them since the last invocation. Alternatively, they could incorporate some logic for the ordering of returned percepts, in case the user only wants one percept per `sense`.

The functionality described above could be encapsulated in an engine extension, providing the proper modules, events and actions. One issue with this is that entities in the XMAS engine are meant to have a position. As mentioned, this is not a strict requirement, as the position can be set to a null value and ignored. Additionally, recent versions of Cartago supports what they call workspaces, which serves to group the agents and artifacts together in different sub-environments, for which positions in the XMAS engine would be well suited.

7.4.2 Environment Interface Standard

While Cartago provides a very specific approach to designing an environment, EIS is at the opposite end of the spectrum, as it aims to provide a standard for designing and connecting to environments. The Java implementation of EIS is very bare bones in terms of functionality for managing an environment, as they have focused more on interfacing with different APLs.

This means that its goals are different from ours. What EIS is good at, is providing a way to comunicate with a language that otherwise had no way of doing so. While the goal of our engine is simply to use APLs that are capable of comunication (some of them being so through EIS).

Chapter 8

Conclusion

In this project, we have developed an engine for constructing scenarios to be used in multi-agent systems. A scenario in this sense is an environment where agents can interact, and with means to control the agents. Our project provides the following features:

- A way to set up environments.
- Means of communicating with APLs, which controls the agents inhabiting the environment.
- Provide a general way for agents to behave in the environment, by allowing them to *act* and *sense*.
- Reactivity in the environments, allowing agents and entities to dynamically respond to changes in the world.

8.1 Results of comparisons

As discussed in 7.4, our engine is placed somewhere in between Cartgo and EIS in terms of multi-applicability and features. That is, our engine is more general

than Cartago, while providing more convenient features for a MAS than EIS (and is, conversely, less feature complete than Cartago). We have argued that the meta-model used in the Cartago project can be implemented in Xmas, and shown that EIS can be used as an extension to the engine in order to exploit its APL compatibility.

8.2 Engine completion

We will now evaluate whether the goals we listed in the introduction have been reached:

Generality: As we discussed in section 7.1, we believe we have reached a good amount of generality in the engine, as we impose few restrictions on the design of environments. The eternal problem is the trade-off between generality and features, and much of our effort have been directed towards equalizing these two qualities. Without several implementations of the engine, it is difficult to say how well our solution caters to different environment systems and their needs.

Ease of use: While our engine features tools that can ease the development of larger systems, it is not comparatively easier to use in small scale aplications. Our minimal example implementation (Vacuum World, see Appendix C) consists of a total of 17 C# classes. It should be noted, however, that the example does not use any extensions apart from the very basic logger extension. Extensions can be used to define some of the abstract notions such as position, and thus ease the development burden. It is also worth noting that most of the classes contains very little code.

Cross platform compatibility: We have compiled the project against the mono platform, which provides the .NET platform on Mac, Linux and Windows. We have developed the engine in both Linux and Windows, and it works as it should on both platforms. At the moment, our reference implementation can not be run on Linux or Mac due to a bug in Mono regarding text buffer sizes when printing to a terminal. While the reference implementation is an important part of this project, we do not consider it a part of the core of the Xmas engine, and therefore conclude that this goal has been reached.

In summation, we believe that the goals for the engine have mostly been reached.

However, There are many aspects of the engine which could be severely improved. These include, but is not limited to,

• The specific way actions and events are designed. For instance, the distinction between environment events and entity events, as discussed in section 5.2.3.

• The way the view is supposed to communicate with the model

• The way agent controllers are forced to have their own threads. This is unnecessary when APLs runs all agents on a single thread, however inefficient that may be.

8.3 Future work

Here, we will list some of the possible additions to the engine, which would make it easier to use by providing functionality for a wide array of MAS scenarios.

- Extensions to communicate with other APLs, that do not support the EIS standard.
- A collection of common environments, such as:
  - Three dimensional worlds
  - A graph based world
- An extension allowing construction and management of discrete worlds.
- Extending the reference implementation to include agents cooperating to find packages.
- Copying the environment and functionality of the Agents on Mars scenario, to showcase and test our engine against an established implementation.
- A collection of commonly used agent actions and percepts, such as means of communicating with each other.

Appendix A

Domain Model UML Diagram for XMAS Model

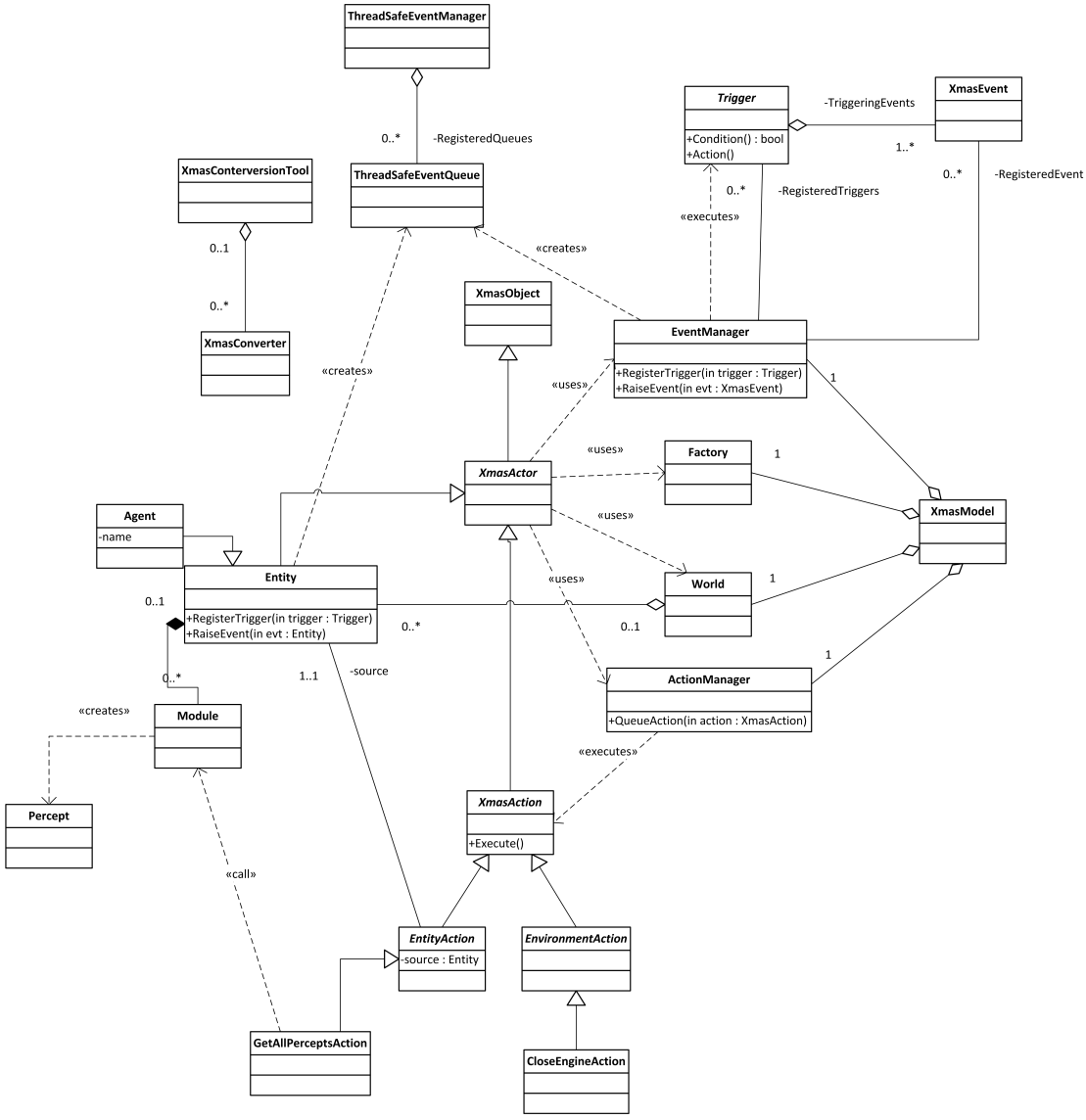

ThreadSafeEventManager
0..*
-RegisteredQueues
ThreadSafeEventQueue
Trigger
+Condition() : bool
+Action()
-TriggeringEvents
XmasEvent
1..*
0..*
-RegisteredEvent
0..*
-RegisteredTriggers
«executes»
XmasConterversionTool
0..1
0..*
XmasConverter
«creates»
XmasObject
EventManager
+RegisterTrigger(in trigger : Trigger)
+RaiseEvent(in evt : XmasEvent)
«creates»
«uses»
1
«uses»
Factory
1
XmasActor
«uses»
XmasModel
Agent
-name
Entity
+RegisterTrigger(in trigger : Trigger)
+RaiseEvent(in evt : Entity)
0..1
0..*
«uses»
World
1
0..*
0..1
1..1
-source
1
ActionManager
+QueueAction(in action : XmasAction)
«creates»
Module
«executes»
Percept
XmasAction
+Execute()
«call»
EntityAction
-source : Entity
EnvironmentAction
GetAllPerceptsAction
CloseEngineAction

Figure A.1

Appendix B

XMAS Engine Component Diagram

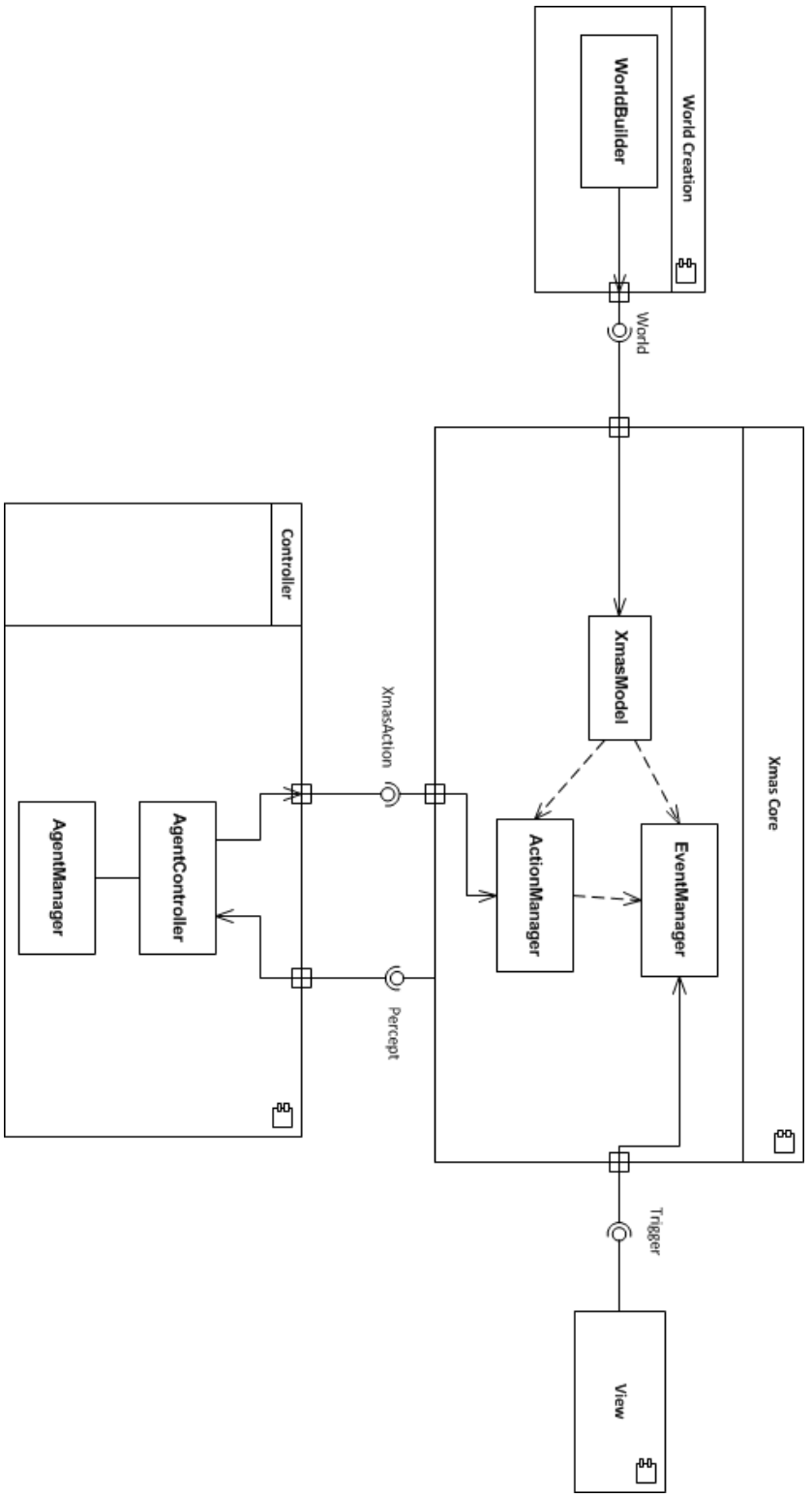

World Creation
WorldBuilder
World
Xmas Core
XmasModel
ActionManager
EventManager
XmasAction
Percept
Controller
AgentManager
AgentController
Trigger
View

Figure B.1

Appendix C

Vacuum World Example

To properly understand the engine it is often needed to have good examples of how to use the engine, as such this part will cover how to make the Vacuum Cleaner World that is a basic Agent based world.

Basically the idea is that the world consists for two tiles, which for the purpose of this example we will call them Tile 0 and Tile 1. The agent is a vacuum cleaner and has the ability to move from one tile to another, furthermore the agent can sense if there is dirt on its current tile. The agent also has the ability to suck if it sucks on a tile with dirt then the dirt is removed. The job of the agent is thus to remove all dirt from both of the tiles, in a more advanced version the tiles gets dirt after an uncertain amount of time, however in this example the dirt is only there from the beginning of the world creation.

The full code for this example can also be found in the `VacuumCleanerWorldExample` project as part of our source code.

C.1 World

We start by creating the world for the Vacuum Cleaner world, however before we do this we must first define what a position in the world is and what information

is need to add an agent.

Position in the vacuum world:

```
1  public class VacuumPosition : XmasPosition
2  {
3      public int PosID { get; set; }
4
5      public VacuumPosition(int posId)
6      {
7          this.PosID = posId;
8      }
9
10     public override EntitySpawnInformation GenerateSpawn()
11     {
12         return new VacuumSpawnInformation(PosID);
13     }
14
15     public override string ToString()
16     {
17         return "Tile("+PosID+")";
18     }
19 }
```

Information need to add an agent to the vacuum world:

```
1  public class VacuumSpawnInformation : EntitySpawnInformation
2  {
3      //Since the vacuum cleaner doesn't require more
           information upon entering the world this information
           is empty
4      public VacuumSpawnInformation(int spawn) : base(new
           VacuumPosition(spawn))
5      {
6
7      }
8
9
10 }
```

Now we construct the world:

```
1  public class VacuumWorld : XmasWorld
2  {
3      //The two tiles that can contain the vacuum cleaner
4      private VacuumCleanerAgent[] vacuumTiles = new
           VacuumCleanerAgent[2];
5
```

```
6       //these two tiles contain the dirt
7       private DirtEntity[] dirtTiles = new DirtEntity[2];
8
9       //Override this to provide a way to insert the agent
10      protected override bool OnAddEntity(XmasEntity
            xmasEntity, EntitySpawnInformation info)
11      {
12          var spawn = (VacuumSpawnInformation)info;
13          var spawnloc = (VacuumPosition)spawn.Position;
14
15          if (xmasEntity is VacuumCleanerAgent)
16          {
17              vacuumTiles[spawnloc.PosID] = xmasEntity as
                    VacuumCleanerAgent;
18              return true;
19          }
20          else if (xmasEntity is DirtEntity)
21          {
22              dirtTiles[spawnloc.PosID] = xmasEntity as
                    DirtEntity;
23              return true;
24          }
25
26          return false;
27      }
28
29      //this method provides the world the ability to remove
            entities
30      protected override void OnRemoveEntity(XmasEntity entity
            )
31      {
32          //since the vacuum cant be removed there is no need
                to provide this in the world
33          if (entity is DirtEntity)
34          {
35              var vpos = (VacuumPosition) this.
                    GetEntityPosition(entity);
36              dirtTiles[vpos.PosID] = null;
37          }
38      }
39
40      //override this for the world to provide the location of
             the vacuum cleaner
41      public override XmasPosition GetEntityPosition(
            XmasEntity xmasEntity)
42      {
43          //Go through each tile to see if the agent is
```

```
            contained within if not return the posion of -1
44          if (vacuumTiles[0] == xmasEntity)
45              return new VacuumPosition(0);
46          else if (vacuumTiles[1] == xmasEntity)
47              return new VacuumPosition(1);
48
49          //same is done for dirt
50          else if (dirtTiles[0] == xmasEntity)
51              return new VacuumPosition(0);
52          else if (dirtTiles[1] == xmasEntity)
53              return new VacuumPosition(1);
54          else
55              return new VacuumPosition(-1);
56      }
57
58      //Override this to provide the ability for the world to
            change position of the entity
59      public override bool SetEntityPosition(XmasEntity
            xmasEntity, XmasPosition tilePosition)
60      {
61          //This can be implemented by removing the entity
                from its last location
62          //and re-add it to the tile of its new position
63          var pos = (VacuumPosition) tilePosition;
64          var lastpos = (VacuumPosition)this.GetEntityPosition
                (xmasEntity);
65          if (xmasEntity is VacuumCleanerAgent)
66          {
67              this.vacuumTiles[lastpos.PosID] = null;
68              this.vacuumTiles[pos.PosID] = xmasEntity as
                    VacuumCleanerAgent;
69              return true;
70          }
71          else if (xmasEntity is DirtEntity)
72          {
73              this.dirtTiles[lastpos.PosID] = null;
74              this.dirtTiles[pos.PosID] = xmasEntity as
                    DirtEntity;
75              return true;
76          }
77          return false;
78
79      }
80
81
82      //Override this to get all entities on the world at a
            specific location
```

```
83      public override ICollection<XmasEntity> GetEntities(
            XmasPosition pos)
84      {
85          var vpos = (VacuumPosition)pos;
86
87          //check if the vacuum cleaner is located at the
                position
88          //then return the entity, if not give an empty
                collection
89          XmasEntity[] vacuum;
90
91          if (this.vacuumTiles[vpos.PosID] != null)
92              vacuum = new XmasEntity[] { this.vacuumTiles[
                    vpos.PosID] };
93          else
94              vacuum = new XmasEntity[0];
95
96          //Check if dirt is located on the given position
97          XmasEntity[] dirt;
98
99          if (this.dirtTiles[vpos.PosID] != null)
100             dirt = new XmasEntity[] { this.dirtTiles[vpos.
                    PosID] };
101         else
102             dirt = new XmasEntity[0];
103
104         //Concatenate the two collections of dirt and vacuum
                cleaner
105         //and make them into an array for the data to be
                immutable
106         return vacuum.Concat(dirt).ToArray();
107     }
108 }
```

Finally we create the factory that is meant to build the world on command from the engine:

```
1  public class VacuumWorldBuilder : XmasWorldBuilder
2  {
3      //This explains to the engine how to construct the world
           we just made
4      protected override XmasWorld ConstructWorld()
5      {
6          return new VacuumWorld();
7      }
8  }
```

C.2 The Entities And Agents

The job of the agent in vacuum world is suck up dirt as such we must add a dirt entity:

```
1  //Here we define a dirt entity
2  public class DirtEntity : XmasEntity
3  {
4  }
```

Before we can add the Vacuum cleaner we must first create the modules it uses, in our case it only needs a module to see the dirt on its current location.

However before that we need to make a percept for the agent, this percept we will call VacuumVision and it will show the vacuum's current position and if the tile it is on is dirty.

```
1  public class VacuumVision : Percept
2  {
3      //Property that tells if dirt is located at the vacuum
           cleaner's position
4      public bool ContainsDirt { get; set; }
5
6      //Property that tells the position of the vacuum cleaner
7      public VacuumPosition VacuumCleanerPosition {get; set;}
8
9      public VacuumVision(bool containsDirt, VacuumPosition
           position)
10     {
11         this.ContainsDirt = containsDirt;
12         this.VacuumCleanerPosition = position;
13     }
14 }
```

Now we can add the module

```
1  public class VacuumVisionModule : EntityModule
2  {
3      //Override this method for the module to provide
           percepts to the VacuumCleaner
4      public override IEnumerable<Percept> Percepts
5      {
6          get
7          {
8              //This module checkes if there are any dirt
                   located on the Vacuum cleaners position
```

```
9               //if dirt is located it returns true if not it
                    returns false as part of the vision percept
10              if (EntityHost.World.GetEntities(this.EntityHost
                    .Position).Any(Ent => Ent is DirtEntity))
11                  return new Percept[] { new VacuumVision(true
                        ,(VacuumPosition)EntityHost.Position) };
12              else
13                  return new Percept[] { new VacuumVision(
                        false, (VacuumPosition)EntityHost.
                        Position) };
14
15          }
16      }
17
18  }
```

And at last we can finally add the Vacuum Agent:

```
1   public class VacuumCleanerAgent : Agent
2   {
3       public VacuumCleanerAgent(string name)
4           : base(name)
5       {
6           //Register the Vacuum vision module to the agent, to
                allow it to recieve percepts
7           this.RegisterModule(new VacuumVisionModule());
8
9       }
10  }
```

C.3 Actions and Events

The vacuum cleaner can perform two actions which are to suck and to move. We would also like that an event gets raised when either action is done.

Thus we must first define both events

```
1   public class VacuumMovedEvent : XmasEvent
2   {
3       //The position the vacuum moved from
4       public XmasPosition From{get; private set;}
5
6       //The position the vacuum moved to
7       public XmasPosition To { get; private set; }
```

```
8
9       public VacuumMovedEvent(XmasPosition from, XmasPosition
            to)
10      {
11          this.From = from;
12          this.To = to;
13      }
14  }
```

```
1   //Define a Vacuum sucked event
2   public class VacuumSuckedEvent : XmasEvent
3   {
4   }
```

Now that the events has been defined it is time to make the actions.

We start by creating the move action, to prevent the agent from being too fast we slow it down by putting a delay on its move action.

```
1   public class MoveVacuumCleanerAction : EntityXmasAction<
        VacuumCleanerAgent>
2   {
3       //Override this for the action to be executable
4       protected override void Execute()
5       {
6
7           //Create a timer so the move is delayed by a certain
                speed
8           //(Otherwise the agent will be able to move
                instantanious only limited by CPU power)
9           XmasTimer timer = this.Factory.CreateTimer(this, ()
                =>
10              {
11                  //the old position of the vacuum cleaner
12                  var pos = (VacuumPosition)this.Source.
                        Position;
13
14                  //the new position of the vacuum cleaner
15                  XmasPosition newpos = new VacuumPosition(-1)
                        ;
16
17                  //Moves the Vacuum to the other tile ei. if
                        on tile 0 move it to tile 1
18                  if (pos.PosID == 0)
19                      newpos = new VacuumPosition(1);
20                  else if (pos.PosID == 1)
21                      newpos = new VacuumPosition(0);
```

```
22
23                  this.World.SetEntityPosition(this.Source,
                        newpos);
24
25                  //Raises the event that the vacuum cleaner
                        has moved
26                  this.Source.Raise(new VacuumMovedEvent(pos,
                        newpos));
27
28                  //Tells the engine the action is complete
29                  this.Complete();
30              });
31
32          //Start the timer, when the timer is done the action
                added above will be queued safely to the engine
33          timer.StartSingle(1000);
34
35
36      }
37  }
```

Next, we create the suck action:

```
1   public class SuckAction : EntityXmasAction<
        VacuumCleanerAgent>
2   {
3       //Override this for the action to be executable
4       protected override void Execute()
5       {
6           //goes through all entities at the vacuum cleaner's
                position and removes all dirt
7           foreach (XmasEntity ent in this.World.GetEntities(
                this.Source.Position))
8           {
9               if (ent is DirtEntity)
10                  this.World.RemoveEntity(ent);
11          }
12
13          //Raises the event that the vacuum cleaner has
                sucked
14          this.Source.Raise(new VacuumSuckedEvent());
15
16          //Tells the engine that the action is done
17          this.Complete();
18
19      }
20  }
```

C.4 Controller

In order for the agent to be able to react to its environment we must provide it with an controllers, in this example we will use C# as the agent language.

First we start by making the controller, the controller contains logic on how the agent should behave, in this example we make the agent suck if there is dirt otherwise it will simply move and look for dirt elsewhere.

```
1  public class VacuumAgentController : AgentController
2  {
3      private EntityXmasAction nextAction = null;
4
5      public VacuumAgentController(Agent agent) : base(agent)
6      {
7          this.PerceptsRecieved += agent_perceptRecieved;
8      }
9
10     //Override this method is the first and only thing
           executed in its controller thread
11     public override void Start()
12     {
13         while (true)
14         {
15             //Force the vacuum cleaner to generate all its
                   percepts
16             this.performAction(new GetAllPerceptsAction());
17
18             //Perform its next action based on its decission
19             if (this.nextAction != null)
20                 this.performAction(this.nextAction);
21
22         }
23
24     }
25
26     //this method will be called when an agent recieves a
           percept
27     private void agent_perceptRecieved(object sender,
           UnaryValueEvent<PerceptCollection> evt)
28     {
29         //go through all percepts
30         foreach (Percept percept in evt.Value.Percepts)
31         {
32
33             if (percept is VacuumVision)
34             {
```

```
35                  //Check if the vacuum cleaner is in a tile
                        with dirt then call suck
36                  //otherwise move the vacuum cleaner
37                  VacuumVision vision = percept as
                        VacuumVision;
38
39                  if (vision.ContainsDirt)
40                      this.nextAction = new SuckAction();
41                  else
42                      this.nextAction = new
                            MoveVacuumCleanerAction();
43              }
44          }
45      }
46  }
```

Second we make an `AgentManager` class this class's object will be responsible for constructing a controller for the agent as such it must be provided with the name of the agent so it can be located in the world:

```
1   public class VacuumAgentManager : AgentManager
2   {
3       private string name;
4
5       //Provide the manager with the name of the agent it
            wishes to construct an agentcontroller for
6       public VacuumAgentManager(string name)
7       {
8           this.name = name;
9       }
10
11      //Override this method which will be called repeatably
            once the engine starts
12      protected override Func<KeyValuePair<string,
            AgentController>> AquireAgentControllerContructor()
13      {
14
15          //The name of the agent the manager will try to
                locate
16          Agent agent = this.TakeControlOf(name);
17
18          //Lambda function that constructs new agent
                controllers, this constructor will be called in
                the agent controller's own thread
19          Func<KeyValuePair<string,AgentController>>
                LambdaConstructor;
20
```

```
21          LambdaConstructor = () => new KeyValuePair<string,
                AgentController>(name,new VacuumAgentController(
                agent));
22
23          return LambdaConstructor;
24      }
25
26      //Override this method to change how the manager
            business logic works, since we only want to start one
             agent controlller we use the manager's thread
            instead
27      public override void Start()
28      {
29          //Aquire constructor(in form of a lambda function)
                for the agent controller
30          Func<KeyValuePair<string, AgentController>> con =
                this.AquireAgentControllerContructor();
31
32          //Construct the agent controller
33          KeyValuePair<string,AgentController> agentcontroller
                 = con();
34
35          //Run the agent controller's start method
36          agentcontroller.Value.Start();
37      }
38  }
```

C.5 View

All other parts are now done and it is time to make a visualization of the world and the agent, to keep the example as simple as possible we will use an extension meant for making loggers and use a log as our view. This way we can track all actions taken by the vacuum agent.

```
1   public class VacuumWorldView : LoggerView
2   {
3       private ThreadSafeEventQueue evtq;
4
5       public VacuumWorldView(XmasModel model,Logger log) :
            base(model,log)
6       {
7           //Construct an ThreadSafe Event queue which triggers
                 can be registered to while keeping the code
                thread safe
```

```
8           this.evtq = model.EventManager.ConstructEventQueue()
                ;
9           this.ThreadSafeEventManager.AddEventQueue(evtq);
10
11          //Register the triggers that track the vacuum
                cleaners actions
12          this.evtq.Register(new Trigger<VacuumMovedEvent>(
                vacuum_Moved));
13          this.evtq.Register(new Trigger<VacuumSuckedEvent>(
                vacuum_Sucked));
14      }
15
16      //Log that the Vacuum cleaner sucked
17      private void vacuum_Sucked(VacuumSuckedEvent obj)
18      {
19          this.Log.LogStringWithTimeStamp("Vacuum sucked",
                DebugLevel.All);
20
21      }
22
23      //Log that the vacuum cleaner moved
24      private void vacuum_Moved(VacuumMovedEvent obj)
25      {
26          this.Log.LogStringWithTimeStamp("Vacuum moved from:
                " + obj.From + ", to: " + obj.To,DebugLevel.All);
27      }
28
29
30  }
```

C.6 Designing the map and wiring the parts together

Before the engine can be started its needed to design a map for the engine, there are multiple options how the world looks we have chosen the one where the agent is on Tile 0 and there is only dirt on Tile 1.

```
1   public class VacuumMap1 : VacuumWorldBuilder
2   {
3       public VacuumMap1()
4       {
5           //This explains to the engine how to generate one
                instantiation of the map
```

```
6           //if the user wishes to make multiple different map
                then it would be need to make new classes
7           //that extends the VacuumWorldBuilder
8           this.AddEntity(new VacuumCleanerAgent("
                vacuum_cleaner"),new VacuumSpawnInformation(0));
9           this.AddEntity(new DirtEntity(), new
                VacuumSpawnInformation(1));
10      }
11  }
```

When the map has been designed the only part left of the engine is to wire all its components together, this is done in the C# programs main method:

Notice! that the log is written to file called viewlog.log

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           //The factory responsible for constructing
                components needed by the engine
6           XmasModelFactory factory = new XmasModelFactory();
7
8           VacuumMap1 map1 = new VacuumMap1();
9
10          //Construct the model with all its required
                components
11          XmasModel model = factory.ConstructModel(map1);
12
13          //makes a file where all view info is logged
14          StreamWriter sw = File.CreateText("viewlog.log");
15
16
17          //Construct the view for the vacuum world
18          VacuumWorldView view = new VacuumWorldView(model,
                new Logger(sw, DebugLevel.All));
19
20          //Construct the manager for the agent controller
                with the name of the agent
21          VacuumAgentManager controller = new
                VacuumAgentManager("vacuum_cleaner");
22
23          //Instantiate and start the engine with the view and
                 the controller
24          XmasEngineManager engine = new XmasEngineManager(
                factory);
```

```
25          engine.StartEngine(model, new XmasView[] { view },
                new XmasController[] { controller });
26
27      }
28  }
```

C.7 Testing the Vacuum World

Opening the `viewlog.log` we can see how the agent has progressed:

```
Vacuum moved from: Tile(0), to: Tile(1)
Vacuum sucked
Vacuum moved from: Tile(1), to: Tile(0)
Vacuum moved from: Tile(0), to: Tile(1)
```

We can see that the agent moved to Tile 1 and sucked up the dirt, after which it went back and forth to locate more dirt.

Appendix D

GOAL Part of Reference Implementation

D.1 Agent Desicion Flow Chart

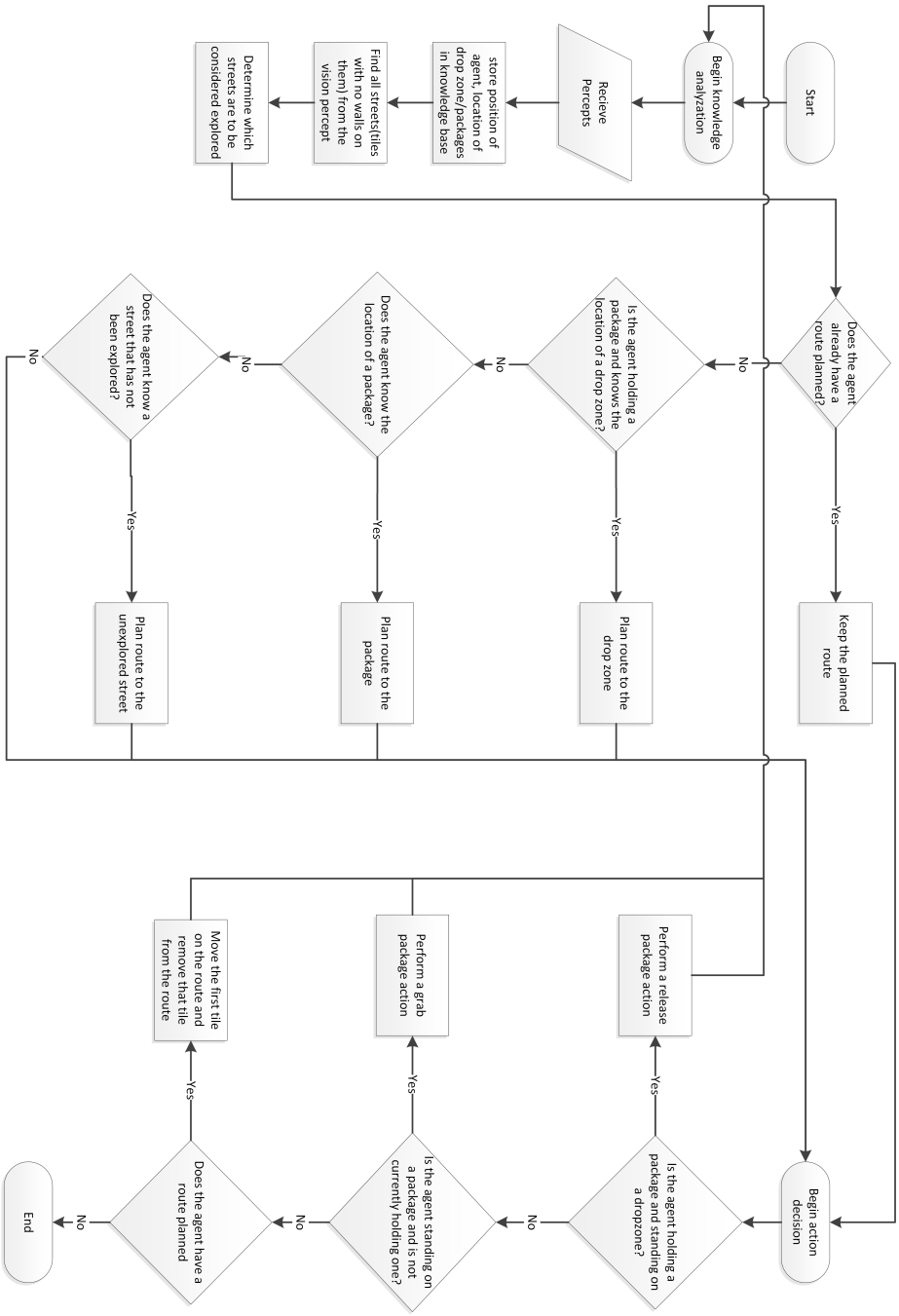


Figure D.1

D.2 GOAL Source Code for Reference Implementation

```
1  init module {
2      knowledge{
3          % Determines if a vision M, that is a tile on the
               map, contains no wall types
4          % findStreet(+M)
5          findStreet(M) :- M = vision(X,Y, HL), Z=(X,Y), (HL =
               []; member(E,HL), E\=wall, E\=impassablewall).
6
7          % Transposes the cordinate X, Y by the position of
               the agent and returns them as Nx, Ny
8          %transpose( (+X, +Y), (-Nx, -Ny) )
9          transpose( (X, Y), (Nx, Ny)) :- position(Px,Py), Nx
               is X + Px, Ny is Y + Py.
10
11         % Checks if a vision contains a certain entity and
               transposes the vision by the agent's position
12         %visionContainsAt(+Vision, ?Entity, -Tx, -Ty)
13         visionContainsAt(vision(X,Y,L), Entity, Tx, Ty) :-
               transpose((X,Y),(Tx,Ty)), member(Entity,L).
14
15         % Detemines if a cordinate is adjacent to another
               cordinate
16         % adjSquare(+X,+Y, -Nx, -Ny)
17         adjSquare(X,Y,Nx,Y) :- Nx is X + 1.
18         adjSquare(X,Y,Nx,Y) :- Nx is X - 1.
19         adjSquare(X,Y,X,Ny) :- Ny is Y + 1.
20         adjSquare(X,Y,X,Ny) :- Ny is Y - 1.
21
22         % Determines if tile on the map can be said to be
               explored when the vision is added to the
               knowledge base
23         %isSquareExplored(+X,+Y,+VisibleTiles)
24         isSquareExplored(X,Y,VisibleTiles):-
25             setof((OX,OY),(adjSquare(X,Y,OX,OY)),OL),
26             setof((NX,NY),(member((NX,NY),OL),(street(NX,NY)
                   ;member((NX,NY),VisibleTiles))),OL).
27
28
29         % find the streets immediateley adjacent to a given
               postition
30         nextStreet(X,Y,Nx,Y) :- Nx is X + 1, street(Nx, Y).
31         nextStreet(X,Y,Nx,Y) :- Nx is X - 1, street(Nx, Y).
32         nextStreet(X,Y,X,Ny) :- Ny is Y + 1, street(X, Ny).
33         nextStreet(X,Y,X,Ny) :- Ny is Y - 1, street(X, Ny).
```

```
34
35          % removes a member M from a list L1 and returns the
                list as L2
36          % delMember(+M, +L1, -L2)
37          delMember(_, [], []) :- !.
38          delMember(X, [X|Xs], Y) :- !, delMember(X, Xs, Y).
39          delMember(X, [T|Xs], Y) :- !, delMember(X, Xs, Y2),
                append([T], Y2, Y).
40
41          %Transpose Vision percept
42          transposedVision(vision(X,Y,Ents),vision(TX,TY,Ents)
                ):- transpose((X,Y),(TX,TY)).
43
44          %Calculates the move direction vector
45          calcMoveVector((TX,TY),(VX,VY)):- position(X,Y),VX
                is TX - X, VY is TY - Y.
46
47
48
49          % ----- A* algorithm Start -----
50
51          % Checks if the A* algorithm is going in circles
52          isNotGoingInCircle(_,node(_,'root',_)).
53          isNotGoingInCircle(Value,node((Value),_,_)):-!,fail.
54          isNotGoingInCircle(Value,node(_,Parent,_)):-
                isNotGoingInCircle(Value,Parent).
55
56          % removes the Heuritiscs information from a node
57          % unWrapHeu(+HeuriticNode, -Node)
58          unWrapHeu(HN,N):-HN=(_,N).
59          unWrapHeu(N,N).
60
61          % Checks if a node(that might contain heuristic data
                ) is the lowest cost to the cordinates X, Y
62          % isBestOnList(+HN,+L)
63          isBestOnList(HN,L):-unWrapHeu(HN,N),N=node(street(X,
                Y),_,SCost), member(OtherNode,L), OtherNode =
                node(street(X,Y),_,OCost),SCost < OCost.
64          isBestOnList(HN,L):-unWrapHeu(HN,N),\+ member(N,L).
65
66          % Calculates the heuristics from Node1 to Node2
67          % calcHeu(+Node1, +Node2, -H)
68          calcHeu(node(street(X,Y),_,_),node(street(TX,TY),_,_
                ),H):-H is sqrt((TX-X)^2+(TY-Y)^2).
69
70
```

```
71          % Determines if a succussor to NodeCurrent should be
                placed on the open list
72          % asSuc(+Open,+Closed,-NOpen,-NClosed,+NodeCurrent,+
                NodeSuccussorValue,+Goal)
73          asSuc(Open,Closed,NOpen,NClosed,NodeCurrent,
                NodeSuccussorValue,Goal):-
74              NodeCurrent = node(street(_,_),_,Cost),
75              NodeSuccussorValue = street(SX,SY),
76              isNotGoingInCircle(NodeSuccussorValue ,
                    NodeCurrent),
77              NodeSuccussor = node(NodeSuccussorValue,
                    NodeCurrent,SCost),
78              SCost is Cost+1,
79              isBestOnList(NodeSuccussor,Open),
80              isBestOnList(NodeSuccussor,Closed),
81              delMember((_,node(street(SX,SY),_,_)),Open,
                    AlmostNewOpen),
82              delMember(node(street(SX,SY),_,_),Closed,NClosed
                    ),
83              calcHeu(NodeSuccussor,Goal,Heu),
84              HeuCost is Heu+SCost,
85              NOpenNonSort = [(HeuCost,NodeSuccussor)|
                    AlmostNewOpen],
86              msort(NOpenNonSort,NOpen).
87
88          % A recursive loop for going through each successor
89          % asLoopSuc(+NodeCurrent,+ListOfSuccessors,+Open,+
                Closed,-FinalOpen,-FinalClosed,+Goal)
90          asLoopSuc(_,[],O,C,O,C,_).
91          asLoopSuc(NodeCurrent,[Suc|Successors],Open,Closed,
                FinalOpen,FinalClosed,Goal):-
92              asSuc(Open,Closed,NOpen,NClosed,NodeCurrent,Suc,
                    Goal),
93              asLoopSuc(NodeCurrent,Successors,NOpen,NClosed,
                    FinalOpen,FinalClosed,Goal).
94
95          % A recursive loop for going through each node until
                the goal node has been reached
96          % asCur(+Open,+Closed,-FinalOpen,-FinalClosed,+Goal)
97          asCur([(_,NodeCurrent)|_],_,_,_,NodeCurrent).
98          asCur([(_,NodeCurrent)|Open],Closed,FinalOpen,
                FinalClosed,Goal):-
99              \+ (NodeCurrent = Goal),
100             NodeCurrent = node(street(X,Y),Parent,_),
101             findall(Suc,(nextStreet(X,Y,SX,SY),\+ (Parent =
                    node(street(SX,SY),_,_)),Suc=street(SX,SY)),
                    SL),
```

```
102             asLoopSuc(NodeCurrent,SL,Open,Closed,NOpen,
                    AlmostClosed,Goal),
103             NClosed = [NodeCurrent|AlmostClosed],
104             asCur(NOpen,NClosed,FinalOpen,FinalClosed,Goal).
105
106         % Converts the path to a list PL by taking the
                parent of each node from the goal until the start
                 node has been reached
107         %unWrapPath(+GoalNode,-Pl)
108         unWrapPath(node(_,'root',_),[]).
109         unWrapPath(node(street(X,Y),P,_),L):-unWrapPath(P,NL
                ),append(NL,[(X,Y)],L).
110
111         %Finds the path from (FX, FY) To (TX, TY) by using
                an A* algorithm
112         % aStarSearchOnStreet( (+FX,+FY), (+TX,+TY), -Path,
                -Cost)
113         aStarSearchOnStreet((FX,FY),(TX,TY),Path,Cost):-
114             EndNode = node(street(TX,TY),_,Cost),
115             asCur([(0,node(street(FX,FY),'root',0))],[],_,_,
                    EndNode),
116             unWrapPath(EndNode,Path).
117
118         % ----- A* algorithm End -----
119
120         % finds the closets tile that has not been explored
                yet according to the knowledge base.
121         % returns a path to that tile, from the agents
                current position
122         % closestUnexplored(-ShortPath)
123         closestUnexplored(ShortPath) :-
124             findall((Cost,Path),( street(ToX,ToY),
125             ( \+ explored(ToX,ToY) ),
126             ( \+ position(ToX,ToY) ), position(FX,FY),
127             aStarSearchOnStreet((FX,FY),(ToX,ToY),Path,Cost)
                    ),UnexploredList),
128             sort(UnexploredList,S),
129             S = [(_,ShortPath)|_].
130     }
131
132     beliefs{
133         % The route is initialized a nothing, since the
                agent knows nothing about the world yet
134         route([]).
135     }
136
137     goals{
```

```
138
139     }
140
141     program {
142
143     }
144
145     actionspec {
146
147         % Clears the move taken from the planned route
148         move(X,Y) {
149             pre { route([(MX,MY)|L]) }
150             post { not( route([(MX,MY)|L])), route(L)  }
151         }
152
153         release {
154             pre { true }
155             post { true }
156         }
157
158         grab {
159             pre { true }
160             post { true }
161         }
162
163
164     }
165 }
166
167 main module{
168     program {
169         % Checks if the agent is standing on a dropzone and
                is holding a package, if so it releases the
                package
170         if bel(position(Tx,Ty), on(Tx,Ty,dropzone), percept(
                holdingPackage) ) then release.
171
172         % Checks if the agent is standing on a package and
                is currently not holding a package, if so it
                grabs the package
173         if bel(position(Tx,Ty), on(Tx,Ty,package), not(
                percept(holdingPackage) )) then grab.
174
175         % Moves to the next tile bases on the route it has
                planned
176         if bel( route([(X,Y)|_]), calcMoveVector((X,Y),(VX,
                VY))) then move(VX,VY).
```

```
177     }
178 }
179
180 event module{
181     program{
182         % Updates the agent's speed and position as they are
                recieved from the engine
183         if bel( percept(position(_,_)), position(X,Y)) then
                delete ( position(X,Y) ).
184         if bel( percept(position(X,Y)) ) then insert (
                position(X,Y) ).
185         if bel( percept(speed(X)) ) then insert ( speed(X) )
                .
186
187         % Updates the agent's knowledge base if it can see
                either a package or a dropzone
188         if bel( V = vision(_,_,_), percept(V),
                visionContainsAt(V, package, Tx, Ty) ) then
                insert (on(Tx, Ty, package)).
189         if bel( on(Tx,Ty,package), percept(vision(X, Y, L)),
                 transpose((X, Y), (Tx, Ty)), \+ member(package,L
                ) ) then delete (on(Tx,Ty,package)).
190         if bel( V = vision(_,_,_), percept(V),
                visionContainsAt(V, dropzone, Tx, Ty) ) then
                insert (on(Tx, Ty, dropzone)).
191
192         % Updates the agent's knowledge base on new streets(
                Tiles without walls) it can see
193         forall bel (percept(V), V = vision(X,Y,Ents),
                findStreet(V), transpose( (X,Y),(Nx,Ny)) ) do
                insert (street(Nx,Ny)).
194
195         % Updates the agent's knowledge base if it can
                determine if any of its streets
196         % are considered explored(ei. it knows about all
                adjacent tiles to a given street)
197         forall bel ( V = vision(_,_,_),
198                      percept(V),
199                      findStreet(V),
200                      transposedVision(V,vision(X,Y,_)),
201                      not( explored(X,Y) ),
202                      findall((AX,AY),(adjSquare(X,Y,AX,AY),
203                      AdjVision = vision(_,_,_), percept(
                             AdjVision),
204                      transposedVision(AdjVision,vision(AX,AY,
                             _))) , VT),
205                      isSquareExplored(X,Y,VT) )
```

```
206                 do insert ( explored(X,Y) ).
207
208         % Clears the route if it has no path
209         forall bel ( route([]) ) do delete ( route([]) ).
210
211         % if the agent is holding a package and knows the
                location of a drop zone, then the agent plans a
                route the drop zone
212         if bel ( not(route(_)), percept(holdingPackage), on(
                X,Y, dropzone), position(PX,PY),
                aStarSearchOnStreet((PX,PY),(X,Y),Path,_) )
213             then insert ( route(Path) ).
214
215         % if the agent is not holding a package and knows
                the location of a package then the agents plans a
                 route to that package
216         if bel ( not(route(_)), not ( percept(holdingPackage
                ) ), on(X,Y, package), position(PX,PY),
                aStarSearchOnStreet((PX,PY),(X,Y),Path,_) )
217             then insert ( route(Path) ).
218
219         % if the agent has nothing better to do it will
                explore the clostest tile that has not beed
                explored.
220         if bel ( not(route(_)), closestUnexplored(Path) )
                then insert ( route(Path) ).
221
222
223     }
224 }
```

Bibliography

[BHD11] TristanM. Behrens, KoenV. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, 2011.

[Hin09] KoenV. Hindriks. Programmingrationalagents in goal. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 119–157. Springer US, 2009.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.

[PL05] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[RNC+96] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[RPV11] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[Tala] Misko Hevery, Google Tech Talks. The clean code talks – unit testing. `http://www.youtube.com/watch?v=wEhu57pih5w`. [Published: 30-10-2008].

[Talb] Misko Hevery, Google Tech Talks. The clean code talks - global state and singletons. http://www.youtube.com/watch?v=-FRm3VPhseI. [Published: 13-11-2008].

[Wika] Wikipedia. Factory method pattern. http://en.wikipedia.org/wiki/Factory_design_patternr. [Accessed: 02-06-2013].

[Wikb] Wikipedia. Law of demeter. http://en.wikipedia.org/wiki/Law_of_Demeter. [Accessed: 02-06-2013].

[Wikc] Wikipedia. Model–view–controller. http://en.wikipedia.org/wiki/Model_view_controller. [Accessed: 07-06-2013].

[Wikd] Wikipedia. Test-driven development. http://en.wikipedia.org/w/index.php?title=Test-driven_development. [Accessed: 25-05-2013].