M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Implementing Intelligent Agents in Games

Christian Kaysø-Rørdam (s082918)

Kongens Lyngby 2014
M.Sc.-2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

M.Sc.-2014

Summary

The aim of this report is implement and describe a framework for allowing the agent programming language Jason to interact with the game Starcraft and then use this framework for implementing some of the basic behavior needed for winning a game of Starcraft.

The first part of the report will described the implementation of the framework. By using it later chapters to implement the behavior of agents, the report shows that the framework allows Jason to interact with Starcraft. The later parts of the report are focused on implementing behavior in Jason using the framework, and show flexible ways to make agents perform some of the important tasks in a game of Starcraft.

The report concludes that the framework works and can be used to implement agents using Jason, and by having access to such a framework it becomes easier to use Starcraft as a platform for testing new theories in Artificial Intelligence.

Preface

This Master thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a MSc degree in Computer Science and Engineering.

Kongens Lyngby, July 1, 2014

Christian Kaysø-Rørdam (s082918)

Contents

CHAPTER 1

Introduction

Games today have become complex enough that they can be used as a testing ground for Artificial Intelligence. One genre of games with a lot of complexity, from an Artificial Intelligence point of view, is the Real Time Strategy genre. Games from this genre generally simulate war on various scales, and involve a lot of different units with different abilities that have to work together to achieve a common goal. One of the leading games in the RTS genre today is Starcraft which not only has the innate complexities of an RTS game, but was also designed in such a way that the rules of the games may be greatly altered. This makes Starcraft an ideal platform for testing various theories in Artificial Intelligence and it already has a large community that use it for just that. Many implementations of AIs for Starcraft are made in either C++ or Java, as tools for interacting with Starcraft exist for both of these languages. These languages are designed to be general purpose languages, and as such it can be hard to reach a point where the behavior of individual agents can be succinctly described. A group of programming languages designed specifically for Artificial Intelligence exist, referred to as Agent Programming Languages, which focus on allowing clear and concise descriptions of behavior. However, no tools have been made for allowing these languages to interact with Starcraft.

This report will show an implementation of an interface between the agent programming language Jason and the game Starcraft: Broodwar that allows Jason to control units inside Starcraft: Broodwar such that it can be made to play the game. It will further show implementations of agents in Jason that are capable of performing several tasks that are important to succeed in a game of Starcraft: Broodwar, such as building infrastructure and micromanaging units.

As Starcraft: Broodwar does not have an API for controlling units, an existing tool called BWAPI will be used for interacting with the game. The implementation itself will be built on top of BWAPI and written in Java. The implementation will conform to an interface, called Environment Interface Standard, that makes it possible to use any other agent programming language instead of Jason with minimal extra work.

1.1 Starcraft: Brood War

Starcraft is a video game developed by Blizzard Entertainment and released in 1999. Brood War is an expansion for the game, released a few years later. In Starcraft you chose to play one of three races: Terran, Protoss or Zerg. In this report we will only be using the Terran race.

Along with choosing a race, a level must also be chosen. A level can either have predefined rules or custom rules. Levels with custom rules are useful when attempting to write and AI, as you can design levels to test specific aspects of the AI. In a level with predefined rules, each player starts with a building, called the Command Center for Terran and four worker units, called SCVs for Terran. Each player also starts 50 Minerals, which is the most abundant of the two resources of the game, and 0 gas. The Command Center is always located next to an area containing additional minerals and gas. Worker units, or SCVs, are able to gather the gas and minerals which increases the players available resources. The Command Center is able to produce additional SCVs at the cost of some resources, and SCVs are able to build additional structures which in turn can build offensive units, again at the cost of resources. A player wins the game if all opposing players have no buildings left, this is achieved by using your offensive units to attack their buildings. Of course, opposing players can also produce offensive units to attack the player or defend their own buildings.

Custom rules in Starcraft can have major impacts on the game play, it is almost possible to create an entirely different game just by changing the rules.

1.2 Why Starcraft

Starcraft is a complex game with many difficult decisions on the path to victory. For instance, using your resources to produce additional workers will increase your income but does nothing to increase your offensive or defensive capabilities, at least not directly, so if your opponent decides to build offensive units instead, you are left vulnerable. Obviously, since your income is now higher than your opponent, you are able to produce offensive units at a higher rate than he is, so if it takes him too long to mobilize his force and launch an attack you may be able to defend and come out with a lead. This is the core of the Starcraft game play, balancing economy with military strength. Furthermore, each type of offensive unit also has different strengths and weaknesses, so it is also important to know what the strength and weaknesses of the army of your enemy is, such that you can produce and army that has the upper hand in a battle.

In a battle, there are many decisions to be made and they all have to be made in real time. Where should each unit position itself to maximize its efficiency, is it worth the lost attack time to move the unit at all, should the unit charge forwards in the hopes of finishing off a high priority target or pull back in the hopes having the enemy chase after it. A human player is not able to keep up with all the decisions that have to be made, and is forced to act on a more macroscopic level, controlling large groups of units at the same time rather than individuals.

All of these complexities together make Starcraft a well suited game for testing various AI theories, as there are a very long list of problems and it is possible to create scenarios for testing solutions to any combinations of these problems.

It is also immensely useful to have powerful custom rules, such that very specific scenarios can be created.

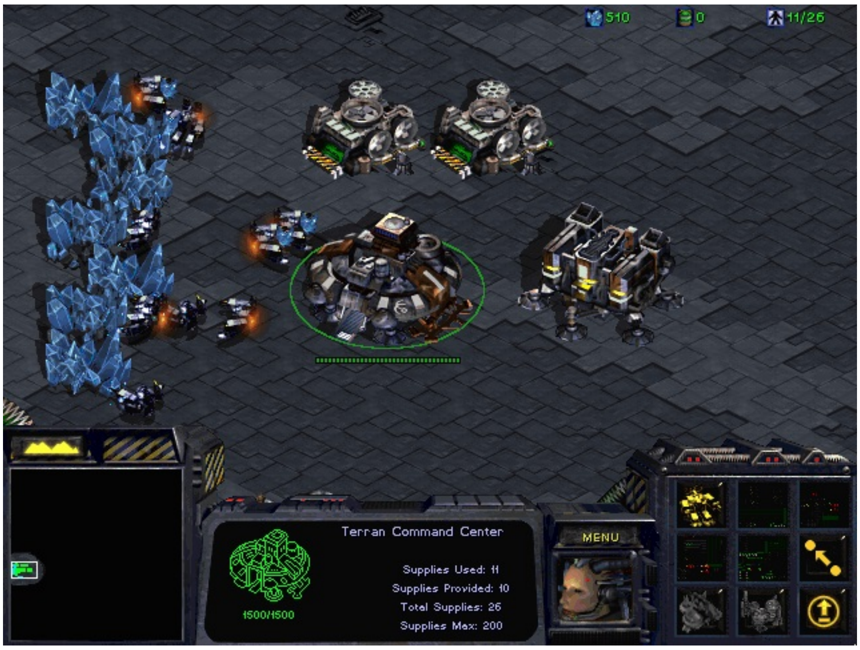


Figure 1.1: A screen capture of a game of Starcraft: Broodwar. The large center building is the Command Center, the two smaller ones above are both Supply Depots and the one off to the right is a Barracks for training offensive units. There are several workers in the process of gathering Minerals. In the top right corner, the current resources are displayed; 510 Minerals, 0 Gas and 11 out of 26 Supply.

CHAPTER 2 Intelligent Agents

In this chapter we will briefly some of the core concepts in agent programming. We will also introduce the agent programming language Jason, as this will be used in later chapters and a basic understanding of it will be necessary.

2.1 Agents

An agent is an entity that operates autonomously and can perceive the environment through sensor and act upon it through actuators. We will see what an environment is in the next section. An agent is considered intelligent if, through the use of its sensor and actuators, can work towards achieving some goal. In Starcraft an entity can be anything from an infantryman to a building, and any of these may be controlled by an agent. An agent will use the sensors and actuators of the entity it controls; the agent can be considered the brain and the entity the body.

When an agent perceives the environment, it receives a set of percepts, where a percept is simply a grouping of data. The agent decides what to perceive, but is limited by its sensors; it may decide to perceive nothing at all but even if it decides to perceive everything, the sensors may not have access to all the data. The agent may analyze the percepts and decide to perform an action, which affects the environment.

2.2 Environment

The environment is where all the agents exist. An environment can either be simulated or real; it could be the insides of a building or it could be a game played on a computer. The environment will initially be in some state and may change over time or change due to actions taken by agents. An environment has several important properties when seen from the perspective of an agent.

- Observability. An environment is considered fully observable if all of it can be perceived at any given state of the environment, if some data is not available in some state it is considered partially observable. For instance, if a closed container existed in some environment, the contents of it would not be perceivable unless the environment was in a state where the container was open, this would then be considered to be partially observable.

- Determinism. An environment is considered to be deterministic if the outcome if any action is deterministic, which means that an agent can know the resulting state of the environment if it performed a certain action. Otherwise the environment is considered to be non-deterministic; the outcome of an action may be random.

- Static or dynamic. An environment is considered static if it does not change while an agent is deciding to what to do, otherwise it is considered dynamic. Any turn based game is an example of a static environment, when it is the agents turn to act, the environment does not change until it decides. Any real time scenario is a good example of a dynamic environment, as simply the passage of time may be enough to change the environment and does not wait for the agent.

- Multi- or single-agent. If the environment only contains one agent, it is considered single-agent, otherwise it is multi-agent.

The environment in Starcraft is the level chosen before a game begins. It is partially observable; in a radius around every unit, the map is revealed to the player of the game, if the unit moves away, that area is again unobservable. The environment is not deterministic, as there are other players in the game, who may do whatever they want and when units attack other units, the outcome is random. The environment is also dynamic, as the other players may take actions while we are deciding what to do. It is also a multi-agent environment that is both competitive, the other players are opponents, and cooperative, all of our units have to work together to succeed.

2.3 Jason

Jason is an implementation of the programming language AgentSpeak, which is made for multi-agent programming. Jason is an Agent Programming Language that uses the Belief-Desire-Intention model and as such uses much of the same terminology. The programming language it self, and much of its syntax, is based on Prolog which is a functional programming language. This report assumes the reader is familiar with the basics of Prolog. Functional programming languages work well in combination with the reactive nature of the BDI model. Jason is a useful tool for implementing agents, the programming language itself is highly abstracted which, in this case, allows the user to write very succinct and concise code describing the behavior of the agents.

A Jason Agent will, once started, continuously run and react to event as well as try to achieve its current goals.

Jason uses various terms, which will be used in the report as well. Following is a description of the main terms in Jason.

Beliefs

Beliefs represent the information an agent has about the world and itself, they are not facts and as such may not actually be true.

All the beliefs of an agent is said to be contained in its belief base. While most beliefs come from perceiving the environment, it is also possible for an agent to draw conclusions from its belief base, thereby adding new beliefs or if the agent needs to remember something it can add a belief with that knowledge. The following is an example of a belief, also known as predicates in functional programming, which says that it is raining outside:

```
raining(outside).
```

This predicate only has a single argument, but it could have any number of them or none at all.

In this case *outside* is a constant because the first letter is lower case, if it started with an uppercase letter it would have been a variable. We will later see how predicates with variable can be used in plans.

Goals

Goals express what the agent wants to achieve in the environment. Jason uses the term goal instead of Desire.

Goals are written in the same way as beliefs, except they start with an exclamation mark:

```
!raining(outside).
```

This means that the agent has the goal of making it rain outside.

Plans

In Jason, plans are what make agents able to act. A plan consists of three different parts:

- The event that triggers the plan. Such events include addition or change of a belief and addition of a new goal.
- The context, which is used for checking if the plan is applicable by looking at the belief base.
- The body, which contains actions to perform and possible new goals or beliefs to add.

The overall structure of a plan is as follows:

```
trigger : context <- body.
```

The trigger must be the first to appear. The context must always be proceeded by a colon and can only appear between the trigger and the body. The body must be proceeded by the reverse arrow '<-' and must appear as the last thing in the plan. All plans must be terminated by a full stop. Optionally a plan can be given a label by adding *@somelabel* to the front of it.

A plan does not need to have a context and a body, although plans without bodies are of limited use. There can be many plans for the same event but no more than a single plan is chosen in response to the event.

```
@p1 +raining(outside) : at(inside).
@p2 +raining(outside) : at(outside) <- move(inside).
```

This is an example of an agent with two plans, labeled *p1* and *p2*, both for the event of *+raining(outside)*. When the belief *raining(outside)* is added to the agents belief base (or if the value changed to 'outside'), it will look through its set of plans to find an applicable one, if it finds such a plan it will be added to the agents intentions otherwise the agent will do nothing in response to the event.

In this example, our agent also holds a belief about where he is, which is used in the context in both plans. In *p1* the context looks through the agents belief base for a belief named *at* with the value *inside* and if it finds it the plan is applicable and will be selected, but finishes immediately as it has no body. What the plan actually expresses, is that if it begins raining outside and the agent is currently inside, it will do nothing. But if the belief base contains the belief *at(outside)* then *p1* is not applicable, and the next plan will be checked for applicability. We see that if this is the case, then *p2* must be applicable, and the agent adds the plan to its list of intentions. When the intention is executed, the action *move(inside)* is performed, which will move the agent inside.

In Jason agents continuously perform reasoning cycles, in which they look for events and plans to handle them, perceives the environment to keep its belief base updated and makes progress on intentions it has previously adopted. An agent can have adopted multiple intentions at the same time, but only one intention will be progressed (an action executed) in each cycle. Jason decides the order in which intentions are progressed, though the user is free to implement its own logic to control this behavior. With the default implementation it is possible for plans to be arbitrarily interleaved, which can be a problem if the body of a plan specifies many actions to be performed and it is important that they happen in that exact sequence. In the case where it is important that the actions in the body of a plan is not interleaved with any actions from other plans, the plan may be marked as atomic, indicating that once it is adopted as an intention, it must be chosen for progress in every cycle until it is finished.

As mentioned earlier, it is also possible for the body of a plan to added new beliefs and goals.

```
@p1 +raining(outside) : at(inside).
@p2 +raining(outside) : at(outside) <- !move(inside).

@p3 +move(inside) : not at(inside) <- walkTo(inside).
```

Here we see the plan from earlier, but we have now changed action *(*move(inside)) to be a goal instead, denoted by the proceeding exclamation mark. Now, when it is raining outside and we are outside, the goal *(*!move(inside)) is added, and plan *p3* is chosen to handle is event. The plan checks if the agent currently believes that it is not inside, I.E. the belief base does not contain the belief *at(inside)*, and then walks inside.

The context of a plan may contain multiple beliefs to be checked, each separated by &, and they must all evaluate to true for the plan to be applicable. Similarly, a plan may also have many actions in its body which are separated by a semicolon.

Variables

So far we have only seen the use of constants, values starting with a lowercase letter, but variable can also be used, which start with uppercase letters. We can rewrite our earlier example to use variable instead of only constants.

```
@p1 +raining(RainLocation) : at(Location) \& RainLocation == Location \&
    Location == outside <- move(inside).
@p2 +raining(_).
```

Initially variables are unbound, that is they have no value. In plan *p1* the variable *RainLocation* will be bound to the value of in the *raining* belief and the variable *Location* will be bound to the value in the *at* belief. The plan then check if they are both equal to the constant *(*outside), and if that is the case the agents move to *inside*. If any part of the context evaluates to false, the plan in not applicable, and we will look at plan *p2*. It ignores whatever value *raining* may have and since it has no context it is always applicable so it is chosen and immediately finishes as it has no body. Again, this is the behavior we had earlier, namely, move inside if its is raining outside, otherwise do nothing.

A variable can be bound to another value, if there were more than one possible value it could be bound to and the previous one made the plan not applicable.

```
colors(red).
colors(green).
colors(black)

@p1 +!paintItBlack : colors(C) \& C == black <- paint(black).
```

Here we have an agent who initially has the belief that it has the three colors red, green and black. When the goal *!paintItBlack* is added, the agent will look for a belief named *colors* and bind the value of C to the value of that belief. But we see here that there are three such beliefs, and C should only be bound to one of them. C will simply be bound to one of them and the agent will check the next part of the context $C == black$. If C was not bound to black, then C is not equal to black, but this does not mean that the plan is inapplicable as we have not yet tried all possible values of C. The agent then goes back and tries binding C to another value, and performs the check again. At some point, it will bind C to black which means the plan in applicable.

CHAPTER 3

Interacting with Starcraft

Before we can start doing any form of agent programming for Starcraft, we must first develop a framework that lets us use Jason to control units inside of Starcraft. To do this, we will first have a look at the tools already available for controlling units inside Starcraft.

3.1 Chosing a framework

As Starcraft: Brood War (SC:BW) does not come with any form of API and was never intended to be controlled by third party software, the only way to interact with the game via software is to edit the memory used by SC:BW. Making a robust framework for doing this is tedious and time consuming as no documentation exists for this purpose. Fortunately a group of people have already developed such a framework called BWAPI. It is currently the only framework for interacting directly with SC:BW. As this is written in C++ and we will be working with Java, we will use the JNI BWAPI which is simply a wrapper for BWAPI made in Java.

3.2 BWAPI

BWAPI, or Brood War API, is a third party open source API made for SC:BW written in C++. The BWAPI works by way of DLL-Injection. This allows it to both read and write data from SC:BW, so that through this API it is possible to do anything a player would normally do, and more. Despite reading the raw memory on SC:BW, the API follows OO programming, and as such exposes a series of classes, where the two most important ones are Unit and Game. Unit is a representation of a unit in SC:BW, this could either be an infantryman or a building. This class is used for controlling all units and also for reading data from units, such as remaining health and movement speed. The Game class contains information about the current state of the game, such as the layout of the current level, number of players in the game. It also serves as a lookup for most entities in the system, such as getting the current player, getting a Unit based on its Id. Here is an example of looking up a unit via its Id and commanding it to move to a set of coordinates.

```
Unit* unit = Broodwar->getUnit(unitId);
```

```
unit->move(x,y, false);
```

The full documentation can be found at: `https://code.google.com/p/bwapi/wiki/BWAPIOverview`

3.3 JNI BWAPI

JNI BWAPI is an open source Java version of the BWAPI that uses the Java Native Interface (JNI) to interact with the C++ version of the BWAPI. It attempts to mimic the BWAPI exactly, and as such has all of the same classes with most of the same methods, all written in Java. However, JNIBWAPI is still early in its development and does therefore not contain all of the functionality of the BWAPI, but it can read all the same information and send all the same commands. What it cannot do it, is use the query-like methods of the BWAPI, such as finding all units within a certain radius of another unit. Here is the same example from earlier, but this time in JNI BWAPI:

```
Unit unit = api.getUnit(unitId);
unit.move(x,y, false);
```

The full documentation can be found at `https://code.google.com/p/jnibwapi/`

We will be using JNI BWAPI, which we will refer to as BWAPI, since Jason is based on Java. This makes it very easy to have to two interact.

CHAPTER 4

Connecting BWAPI to Jason

In this chapter we will see what is required for Jason to be able to control units in Starcraft through BWAPI and how these requirements can be met. The solution will utilize the Environment Interface Standard, such that it becomes very easy to replace Jason with any other Agent Programming Language.

4.1 Environment Interface Standard EIS

EIS is an interface designed to be a bridge between an Agent Programming Language (APL) and an Environment. The idea of EIS is to be an abstraction of the environment, but with no connections to a specific APL, such that only a single implementation for the specific environment exists and everyone is free to use their APL of choice for interacting with it.

At the core of EIS sits the entities, which serve as a mapping between agents in the APL and actors in the environment. In EIS an entity is simply represented by a name and nothing else. Entities have the ability to observe the environment in the form of a list of Percepts, where a percept simply a tuple where the first entry is the name of the percept and the following entries are values associated with the percept. For instance, getting a percept of the weather could look like this: ('Weather', 'Raining', 8), where the last entry is the windspeed. Entities may receive a lot of different Percepts when they observe the environment. It also has the ability to perform actions in the environment, where an action can also be viewed as a tuple with the first entry being the name of the action and the other entries being information relevant to performing the action. None of the entities will do anything unless told to by some agent. Perceiving and acting do not have any form of default implementations, and it is up to the programmer to provide implementations of these things for the specific environment, in this case Starcraft.

As EIS is just an interface, it is up the programmer to decide which parts of the actual environment to expose through it.

Furthermore, as APLs are not designed to integrate directly with EIS, the programmer must also provide an implementation for this integration between the APL and EIS.

When an entity wants to perform an action, the action first goes through several different tests to see if it applicable, the implementation of these tests are left up to

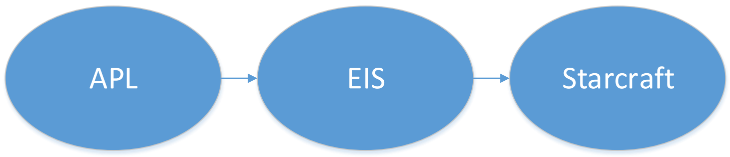


Figure 4.1: A figure showing the relationship between the APL, EIS and Starcraft

the programmer. Theses are the tests, and they are performed in the order as shown below.

- Is it supported by the environment? This should test whether the action even exists in the current environment. The signature of this method is *boolean isSupportedByEnvironment(Action action).*
- Is it supported by the entity type? This should test whether the type of the entity can perform such an action, note that EIS does not associate entities with any types, this is there for cases where the programmer wants to associate entities with more than just a name. The signature of this method is *boolean isSupportedByType(Action action, String entity).*
- Is it supported by the specific entity? This should whether this specific entity is able to perform the action in its current state. This may seem very similar to the type test, but for instance in Starcraft, buildings can move but only if they are currently flying, so a move action for a building would pass the type test, but fail this test if it was not flying at the time the action was requested. The signature for this method is *boolean isSupportedByEntity(Action act, String name).*

If an action passes all these tests, only then will EIS attempt to execute it. Note that the execution of an action may still fail, for instance if the parameters for the action were invalid. The implementation of the code that attempts to execute the action is also left up to the programmer, so any further tests of the applicability of the action can be made here, and appropriate exceptions can be raised.

4.2 Connecting EIS to Jason

For Jason to be able to talk with EIS, we need to implement a mapping between the concepts in Jason and their equivalent in EIS. This is very simple to do for Jason, as it uses the same concepts as EIS (EIS was designed to make the implementation of this simple in most cases). The translations we need are as follows:

- From EIS Percept to Jason Literal, such that Jason agents can have information about their environment.
- From Jason Literal to EIS Action, such that Jason agents can perform actions in the environment. In Jason, Literals are also used to represent actions.
- From Jason Term to EIS Parameter, for internal use to convert the arguments for a Jason Action to their EIS equivalent.
- From EIS Parameter to Jason Term, for internal use to convert the arguments for an EIS Percept to their Jason equivalent.

The implementation of these translation are simple because Jason and EIS use the same concepts with the same information in them, so it is simply a problem using the correct types in Java. Here is an example of converting a JASON Literal to an EIS Action:

```java
    public static Action literalToAction(Literal action) {
        Parameter[] pars = new Parameter[action.getArity()];
        for (int i = 0; i < action.getArity(); i++) {
            pars[i] = termToParameter(action.getTerm(i));
        }
        return new Action(action.getFunctor(), pars);
    }
```

Where the getFunctor method returns the name of the action.

Later, we will see how such an EIS Action is actually executed. We have now seen how a translation can be done, but we also need a place to do actually do the translation, this happens in our of the implementation of the Environment class that comes with Jason. The Environment class has two methods we are interested in overriding:

- getPercepts(string agentName) This method is called when Jason wants to get percepts for one of its agents, and it is here we will create percepts for that entity in EIS and translate them for Jason.
- executeAction(string agentName, Literal action) Again, this is the method Jason calls when an agent wants to perform an action. The Structure object contains the name of the action to be performed and its parameters, if any. This is where we translate the Jason Literal to an EIS Action and attempt to execute it.

Providing an implementation of these two methods is enough to integrate JASON with EIS.

4.3 Connecting BWAPI to EIS

We have to map the concepts of EIS to their equivalent in BWAPI. This is easily done for Actions, as they can be viewed as giving a unit a command in BWAPI. For instance, if an agent wants to move to some position, this would mean commanding the unit the agent controls to move to that position.

```java
    public void execute(Unit unit, Action action) throws ActException {
        LinkedList<Parameter> parameters = action.getParameters();
        int x = ((Numeral) parameters.get(0)).getValue().intValue();
        int y = ((Numeral) parameters.get(1)).getValue().intValue();

        api.move(unit.getID(), x, y);
    }
```

Here we see the method that is called when an agent wants to move to a position. At this point, we already know that we are dealing with a move action, so we simply get the coordinates from the parameters of the action and call the appropriate method on the API.

In EIS, entities are able to observe the environment around them, but in BWAPI a unit is only able to observe itself, that is, the Unit object only contains information about the state of the unit. While entities will need to have information about their own state, they will not be useful to us if they cannot see the environment. In BWAPI all information about the environment is associated with either the Game object or the Player object, so we will need to look at the information they contain and provide it as percepts to the relevant entities. Not all units need to receive the same Percepts, as some might not have any use for certain percepts because they either cannot take any action that affects the percept or simply do not care about the percept. For instance, a worker unit can spend and gather resources and would therefore be interested in percepts that contain information about the current amount of resources available, whereas an offensive unit can do neither of those and therefore does not care about those percepts.

BWAPI contains many different events, that are raised in response to events in the game. It is through these events we discover when a new unit is created and create an entity in EIS to represent it. An event is also raised when a unit leaves the game, and at this point we should remove the entity from EIS. All of the other events are not relevant to EIS, as it just sits and waits for the APL to ask for new Percepts or perform an Action. In other words, EIS does not tell the APL of changes to the game, it is the job of the APL to continually perceive the game and observe the changes.

To keep track of these things in a sensible way, it makes sense to create a model of Starcraft, albeit a much simplified one, that looks the way EIS expects an environment to look, namely with units having their own percepts of the environment and being able to perform actions.

4.4 Modelling Starcraft for EIS

This model should be both lightweight and extensible; we are not interested in constantly updating values to match what they are in Starcraft and we would like to be easily able to add new features to the model.

As we have seen earlier, EIS uses entities, percepts and actions, so we should make models for each of these concepts. EIS calls the following method each time it wants to get the Percepts of an entity:

```
LinkedList<Percept> getAllPerceptsFromEntity(string entity);
```

With this being the case, there is no reason why we should constantly keep our model updated with values from the environment; we can simply read the desired values when EIS wants them.

As we saw earlier, EIS considers an entity to be no more than a name, but we want to be able to distinguish entities based on what type of unit in the environment they control. To do this, we will make a model for each type of unit in Starcraft, and associate each EIS entity with an instance of one of our models. When a new unit appears in Starcraft, an event is raised in BWAPI with the Id of the unit as a parameter, this is the callback we use in EIS to add a new entity, and since we have the Id of the unit being created, we can use BWAPI to find out which type it is.

In the following we will see how Units, Perceptions and Actions have been modeled.

Perceptions

Given the simplicity of a Percept and the fact that EIS already has its own class for it, there is no reason why we should create yet another version of it, so we will simply continue using the EIS one. What we do need however, is a way of generating Percepts. We know that different types of units will need to have different Percepts, but many units will also have some of the same Percepts. For instance, all units that can move, will need a Percept with information about their current position.

Clearly, our way of generating Percepts has to be able to elegantly handle all the different types of units there are in the game and do so without redundancy. A simple solution to this, is to have a series of Percept generators, or Perceivers, with each being able to generate a set of perceptions. A unit will then have a set of these Perceivers, and when Percepts are requested for that unit, we will simply query each Perceiver for Percepts and join all the returned sets.

A set of Percepts returned by a Perceiver, should contain as few as possible Percepts and if there is more than one, that should only be because they each do not make sense on their own. For instance, we could have a Resource Perceiver, that generated Percepts with information about the currently available resources, one Percept for minerals and one for Gas. All units able to spend resources are able to spend both types, so there is no reason to each of these into separate Perceivers.

One could argue that there should then simply not be a Percept for each, but rather just a single Percept with the information of both. The reason for choosing to have two Percepts rather than one, is to allow for easier composition of Perceivers. As mentioned earlier, a game of Starcraft can either follow predefined rules or custom rules, and in a game with custom rules one could easily imagine that units that spend resources could only spend one or the other.

By having each Percept contain as little information as possible, it becomes much easier to design Perceivers of any scenario.

In the implementation, a Perceiver has to implement the following interface:

```java
public interface IPerceiver {
    public List<Percept> perceive();
}
```

And here we can see the actual implementation of the perceive method for available resources perceiver:

```java
    @Override
    public List<Percept> perceive() {
        ArrayList<Percept> percepts = new ArrayList<>();
        Player self = api.getSelf();

        percepts.add(new MineralsPercept(self.getMinerals()));
        percepts.add(new GasPercept(self.getGas()));
        percepts.add(new SupplyPercept(self.getSupplyUsed(), self.
            getSupplyTotal()));
        // units may only be built while there is sufficient supply left
            over
        return percepts;
    }
```

Any unit who then needs to know about the currently available resources will then have an instance of this perceiver.

When EIS wants the percepts of a given entity, we will then find the unit associated with this entity and simply call the perceive method. This gives us all the Percepts for that entity, which EIS can then return to the APL.

Actions

As we have seen earlier, and EIS Action is nothing more than a tuple of values, so if we want an Action type that can actually be executed in Starcraft we will have to create our own, that can use the values stored in an EIS Action. We will require that our actions are easily reusable and new ones can implemented quickly and easily, such that we can support arbitrary game rules that has other actions. An obvious way to implement actions would be to simply add each action as a method on the appropriate Unit model, and then use inheritance to avoid implementing the same action several times. But this can very quickly become a mess, and it can also be difficult to get a good overview over which units have do what. Also, as EIS refers to actions by their names, it may not be trivial to translate this name to a method on an object.

Instead, we will choose the approach of having actions completely separate of the Unit model. This makes it very easy to add new actions as we can have an interface to represent an action; adding a new action simply becomes a matter of implementing an interface. The issue with this approach is that without an association to a unit, it becomes possible to attempt to execute an action that a unit would never be able to perform. Earlier we saw that EIS actually has considered this problem, as it will test whether a unit can actually perform an action. And given that, in the end, it is the APL that decides which action a unit should try to take, the programmer of the APL should be aware of which actions each units are able to take, and not try to perform illegal actions.

The Actions we design should have methods for determining whether they are valid or not, such that we can call these methods in action tests in EIS. We would like to be able to test whether the action is well formed, that is, does it have the correct number of parameters and are they of the correct type. We would also like to be able to test if a given Starcraft unit is actually able to perform the action in it current state. This gives us an interface of the action with the following methods.

- *boolean isValid(EISAction action);*. This method will test if the action is well formed. As EIS does not have a method designed for such a test prior to executing, we will test this just before the execution step ourselves.
- *boolean canExecute(Unit unit, EISAction action).* This will test if the given Starcraft unit can execute the action. This fits with the EIS test for whether an entity can perform an action, and should be tested there.
- *void execute(Unit unit, EISAction action);*. This method will execute the action by calling the appropriate method in the BWAPI. This is the method we will call during the execution step of an action in EIS.

Since EIS refers to actions by name, we will need to maintain a map of names to their corresponding actions. This map can also be used for the EIS check of whether an action is supported by the environment.

Units

Given the way we have chosen to implement Actions and Perceivers, the unit model will now only need to contain a list of Perceivers. As the perceivers are enough to the describe the unit, there is not need to make a unit model for each type of unit in the game, the type of the unit is implicitly described by the perceivers it has. This allows us to easily model any type of unit in the game, we just compose a different set of perceivers.

The downside to this approach is that the unit model on its own does not describe anything, and given that it accepts the perceivers as part of its constructor it may be cumbersome to actually create an instance of a unit. This problem is solved by a helper method which translates the name of an entity to a unit with the correct perceivers.

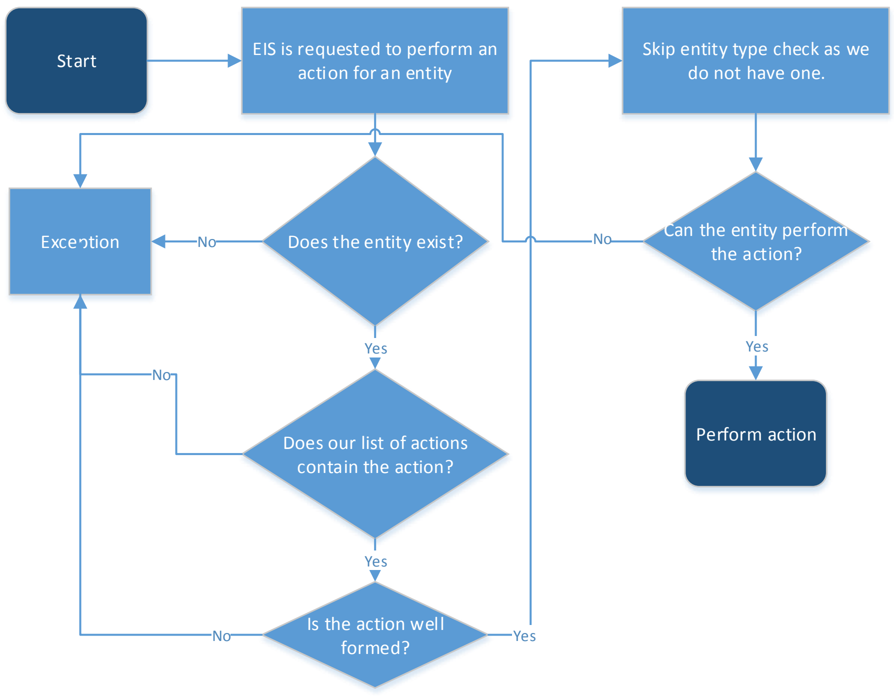

Start
EIS is requested to perform an action for an entity
Skip entity type check as we do not have one.
Exception
No
Does the entity exist?
No
Can the entity perform the action?
Yes
Yes
No
Does our list of actions contain the action?
Perform action
Yes
No
Is the action well formed?
Yes

Figure 4.2: A figure showing the flow of an action request

4.5 Putting it all together

Now that we have a model EIS can use and a way for EIS to talk to BWAPI and Jason to talk to EIS, we are able to use Jason to write agents that act inside Starcraft.

It is important to note that only the actions and perceivers that we will be using later on in the report have been implemented. Most actions that are available in Starcraft have actually been implemented, as they rather generic in nature. For instance, to make a Unit use one of its special abilities, Stimpack if it is a Marine, we have an action that takes the ability to use as a parameter, rather than have an action for each special ability.

The perceivers that have been implemented cover much less of what can actually be perceived, as a perceiver has to be implemented for every single set of data one could imagine using. Some basic ones have been implemented, so that an agent has information about its own state, but only limited information about its surroundings.

Given the way perceptions are generated, it should be possible to generate any perceptions one would require for writing intelligent agents. The reason for this is, perceivers have access to the entire game state, including all units in it, and any of the knowledge that can be learned from the state can be turned into a percept. The only information that is not available, is which percepts some other agent currently has, since no way of retrieving percepts from an agent as been implemented. However, this should not be needed, as agents in Jason are able to communicate with each other, so if an agent requires some information about some other agent, it can simply ask for it.

CHAPTER 5

Micromanaging units

In this chapter we will see how we can make agent cooperate with each other without explicitly communicating with each other. For simpler tasks, not using communication simplifies an agents logic as the overhead of having a communication protocol can be avoided.

To make agents cooperate without communication, they will have to observe the behavior of the other agents in the environment and factor this into their decision making.

We need an environment with a problem that requires the agents to cooperate. Their task in the environment will be to distribute themselves evenly and attack the targets. When a target is destroyed, the agents attacking it should then distribute themselves among the remaining targets.

There are eight targets and 13 agents, so some targets will initially only have a single agent attacking it. The agents are not initially in range of any of the targets, and will have to move in order to attack them.

See figure 5.1 for a drawing of the environment.

Splitting up a group of units is very useful in Starcraft; if applied properly it can help optimize how workers gather minerals. It can be used for setting up for a flanking maneuver and avoiding dangerous attack that affect an area. Against human players, splitting ones army in several evenly sized pieces and attacking multiple different locations is very difficult for the defending player to react to properly, as he is only able to view and command units in a single area at a time.

5.1 Distributing agents

We can view the agents as having two different states.

- The agent does not have a target. This is the initial state of the environment, and whenever a target is destroyed the agents who had it has their target will again have no target.

- The agent has a target. This is the case when an agent has found a target and is attacking it or moving to attack it.

Depending on which state an agent is in, it will have to use different logic to determine if it should pick a target and which to pick. When the agent does not have a target, it is clear that it must pick one, and to ensure a roughly even distribution of

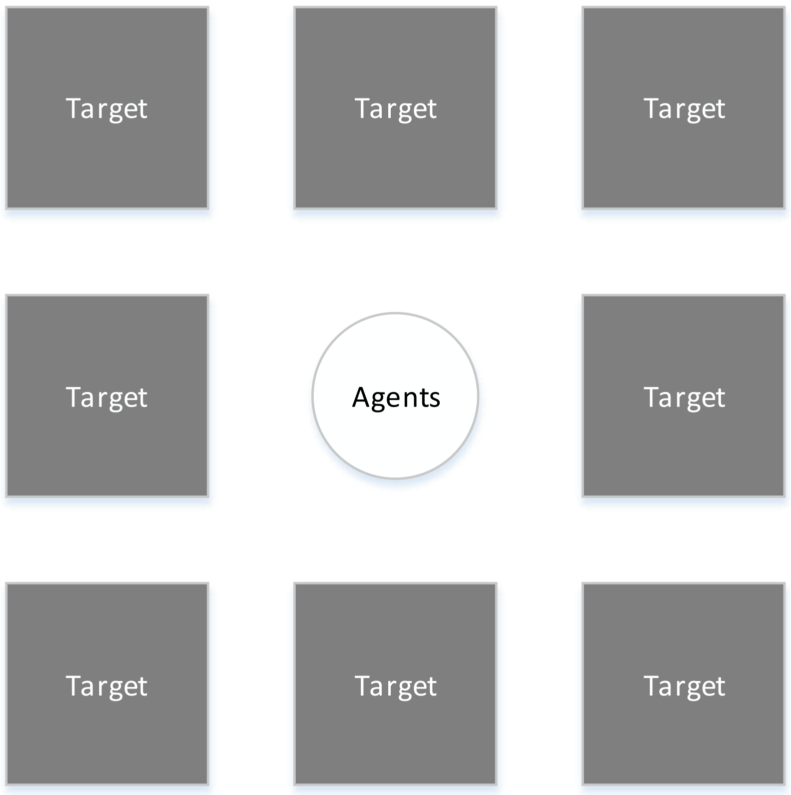


Figure 5.1: An environment for testing cooperation of agents without communication with a total of 8 targets and 13 agents who are initially located inside the circle.

agents it should pick the target with the fewest attackers. If there are many targets that all have the fewest number of attackers, the agent should pick the closest one, simply to spend as little time as possible walking.

If the agent already has a target, it should only pick a new target if there is a target with at least two fewer attackers than the current target of the agent, otherwise the distribution will not be improved by the agent moving. Furthermore, it is important that if the distribution is as even as it can be and all agents are participating, that no agents move for any reason as this will waste time for no gain.

The implementation of this behavior in Jason is as follows.

```
weigh(EID, W) :- .count(attacking(_ ,EID), N) & id(ID) & jia.distance(ID,
    EID, D) & W = w(EID, N, D).

+!attack : id(ID) & attacking(ID, T)
    & weigh(T, w(T, TN, TD))
    & .findall(t(N, D, EID), (enemy(_, EID, _, _) & weigh(EID, w(EID, N, D))
         & N + 1 < TN), WS)
    & .min(WS, t(_, _, EID))
        <- attack(EID); .wait(300); !attack.

+!attack : id(ID) & not attacking(ID, _)
    & .findall(w(N, D, EID), (enemy(_, EID, _, _) & weigh(EID, w(EID, N, D))
        ), WS) & .min(WS, w(_, _, EID))
        <- attack(EID); .wait(300); !attack.

+!attack <- .wait(300); !attack.
```

The third plan for the attack goal is there for the case where an agent has a target and cannot find a better one. The *attacking* percept tell us the current target of each agent, and is used to determine how many attackers a given target has.

5.2 Acting simultaneously

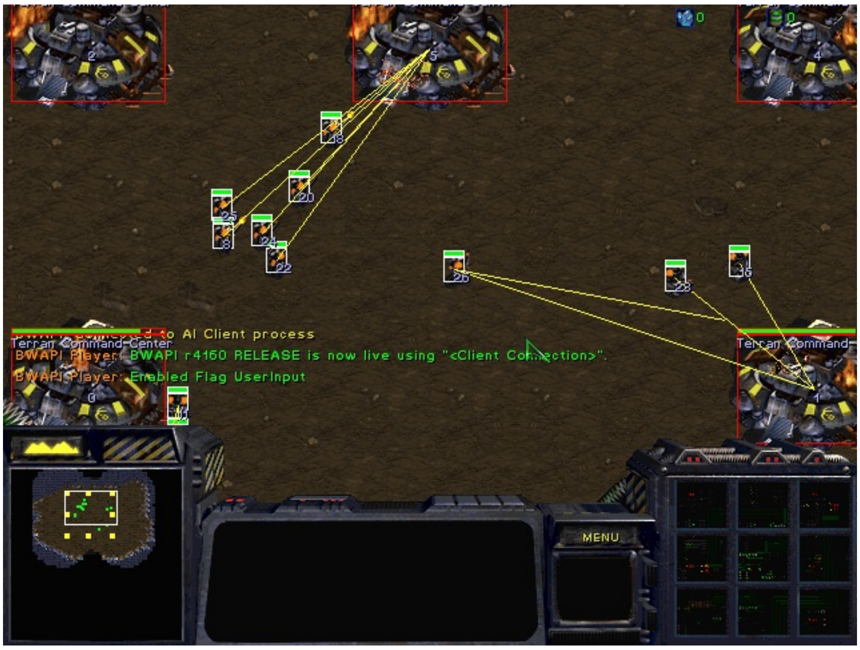
With this Jason code in hand, we can now try running our agents in the environment, where we expect them to distribute themselves evenly. But, what actually happens is that all the agents make the same decisions, resulting in them moving around in a big clump. When the clump reaches a target, all the agents see that the distribution is off and find a better target, but they all find the same new target.

This behavior continues for a while, and sometimes a single agent will find another target than the group did. This happens because the agents cannot all occupy the exact same position, which means sometimes an agent will chose another target because it is closer. Obviously, this behavior wastes a lot of time, as most agent run around all the time, never attacking any of the targets.

This problem stems from the fact that the logic for selecting a target is based on a few values, but each agent observes these values at the exact same time meaning they all get the same values. The only value that is actually different from agent to agent, is the position of the agent and given that they start almost on top of each other, the position is almost the same for each agent.

The agents have to be able to observe the decisions of the other agents if they are to behave differently. This is not happening when they observe and act at the exact same time, so a solution to this problem would be to ensure that they do so at different times.

We can easily control when the agents start acting, as this is decided by when the *attack* goal is added. Currently, it is added to all agents when the game starts, which meas they all start acting at the same time. By offsetting the time at which the goal is added, we can make the agents observe and act at different times.



to AI Client process
BWAPI Player: BWAPI r4160 RELEASE is now live using "<Client Connection>".
BWAPI Player: Enabled Flag UserInput

Figure 5.2: A screen capture showing the uneven distribution of agents. The yellow lines from the agents to the targets indicate what the agents are targeting. The lower three targets are not visible in the screen capture, but looking in the bottom left corner an overview of the level can be seen with all eight targets in Yellow.

One way to do this, would be to insert a delay relative to the id of the agent, as the ids of agent are unique. This would let the agent with the lowest id act first and the second agent would then observe the actions of the first, which would result in picking another target than it did. The problem with this approach is that there may be several agents who are all closest to the same target as the first agent is closest to, but the first agent may actually be farthest away of all those agents.

The delay could also be based on the distance of each agent to its closest target, this would minimize the total distance the agents would have to walk before they all reached their first target. But it does nothing to minimize the distance to their second or third targets.

The solution chosen here is to insert a random delay after each time an agent decides to attack a new target. This does not eliminate the problem of some agents not getting their best target, but it does not bias agents towards better or worse targets overall. The alternative would be to very accurately keep track of when each agent observes the environment and makes decisions, and include this in the logic somehow. But given that agents already chose the closest target they can, the benefits in reduced walking distance is not worth the added complexity.

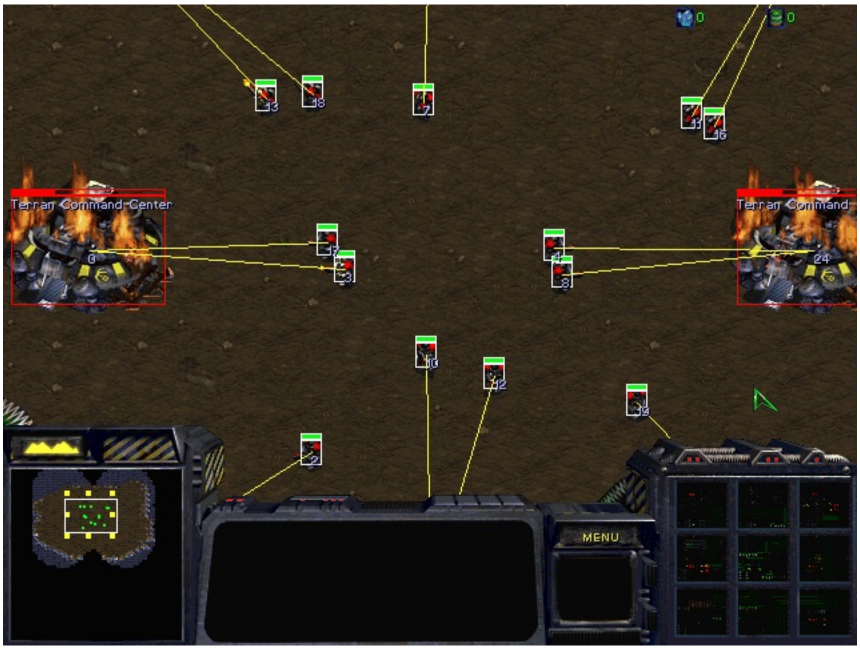


Figure 5.3: A screen capture showing an even distribution of agents.

CHAPTER 6

Coordination and base building

In this chapter we will see how we can make our AI construct a functioning base i Starcraft by having them communicate with each other. In order to do so effectively we must solve several problems.

- There is only one resource pool and every unit has access to it. We have to find a way coordinate the usage of these resources such that we do not end up with units who are starved of resources and therefore unable to complete their goals.
- When an agent discovers the need to a specific building, we have to make sure that only one such building will be built. Given that we use autonomous agents, many of which have access to the same data, it is likely that several of them will conclude that a specific building is needed at the same time. This could lead to building multiple identical buildings, even if we only needed one. The solution to this problem must preserve the autonomy of our agents.

6.1 Coordinating resource usage

As discussed earlier, Starcraft uses three different resources, Minerals, Gas and Supply, where Minerals were abundant and Gas was scarce and both were collected from certain locations in the level by workers, and permanently spent when constructing units or buildings. Supply functions differently, when a unit is created some free supply is reserved for that unit, if the unit later dies the supply is freed. One can increase the amount of free supply by constructing certain buildings.

When an agent performs an action that requires resources, the resources will be deducted from the total or the action will fail if there were not enough resources. If the action fails, one could simply try to perform it again, until it succeeds. This poses several problems; as resources increment in small amounts but very frequently, this behavior essentially gives us a priority queue, where the highest priority goes to the unit who requires the least amount of resources. This happens because all the units will be trying to spend the resources all the time, and the resources have to reach lower number before reaching the higher number, as they change in small increments. Once the resources reaches the low requirement, the unit then spends the resources it needs, probably leaving the resources at zero.

We would like more fair solution, such as a queue. Coordinating this between all the agents would be difficult as all agents would have to talk to all other agents, a simpler approach would be to have an agent designated to managing the resources who all the other agents must coordinate with.

Such a dedicated resource manager should have the following properties:

- Receive requests for resources and remember the order in which they were received.
- Grant requests at a later time only when there are enough resources available.
- Grants should be given first in first out.
- When a request has been granted, the requested resources must remain available until the requester decides to spend them. The requester may spend those resources at its own discretion at any point in time after the grant was given.

We will not require it to be able to refuse requests, as that would only really make sense in a case where it could be sure that such a request could never be granted. The other agents should simply never request resources that will never be available. We also see that it will be possible for agents to request resources but never spend them, thereby denying other agents of those resources, we will assume that agents do not request resources they never intend to use and they will request them only at a reasonable time before needing them. While not a property of the resource manager, it is important that agents who request resources are free to perform any other action while they wait for a grant.

To keep track of the order of the requests and to ensure that grants are given first in first out, we will store all pending requests in a queue. From Starcraft we have access to both the current amount of available resources but also the total amount collected. The current amount changes whenever resources are spent or gathered, and given that we need to guarantee that the required resources are available after a grant, we will have to keep track of our own current amount of resources.

We will keep track of how many resources we have granted the use of, and how many total resources have been collected, and only grant the use of additional resources enough ungranted resources available. By doing it this way, we consider resources spent as soon as we grant the use of them, and will not grant the use of these resources to any other agent. This ensures that from this point until the requesting agent actually spends the resources, the resources will be available for use. Whenever a request is granted, we will remove the request from the queue and add the requested amount to our spent resources.

We also have to figure out when to test if we can grant the request currently at the head of our queue. Given that we know when we are likely to be able to grant resources, it makes the most sense to test at those times rather than testing in a loop.

- When the resource total changes, we may be able to grant the request at the head of the queue.

- When the head of the queue changes, for instance when the first element is added or a request is granted, as we may be able to grant the new head.

The test for whether we can grant a request is simple, as can be seen here:

```
@i[atomic] +!checkQueue : totalRes(_, MT, _, GT, SC, ST) & peek(entry(AN, M,
     G, S, X)) &
                spent(MU, GU) &
                M <= MT - MU &
                G <= GT - GU &
                S <= ST - SC
                <-  !dequeue(_);
                    -+spent(MU + M, GU + G);
                    .send(AN, tell, grant(M, G, S, X)); !checkQueue.
@j[atomic] +!checkQueue.
```

Where the totalRes is (current minerals, total minerals, current gas, total gas, current supply, total supply) and peek retrieves the head of the queue.

As we can also see, it has been marked as atomic, which tells Jason that it must keep the plan checkQueue as it current intention until it is finished. All the other plans are also marked as atomic to ensure that the queue is not changed from the outside during the execution of any intention. For instance, in the checkQueue plan, if we had not marked it as atomic, the element at the head of the queue we find using peek in the context may not be the element that is dequeued from the queue in the body. This could lead some requests getting lost and others getting granted more than once.

With this agent in place, agents are able to coordinate resource usage. A downside to this approach is that the implementation of the other agents will have to be more complex. When an agent concludes that it needs resources to complete its current goal, it now has to send a request to the resource manager, it then has to keep track of which requests it has sent as we do not want to send duplicate requests. When the grant arrives, it may then continue to complete the goal. However, agents should not be idle while waiting for the grant to arrive; it should continue trying to complete any other goals it may have.

A way to achieve this behavior is to redefine what its goal is. For instance, when an SCV gets the goal of building a Supply Depot, a plan in Jason is then executed and when the plan successfully finishes the goal is satisfied. We know that this goal cannot be satisfied until a request for the required resources has been sent and the grant for those received, and so to keep the goal of building the Supply Depot, we would have to alternate between waiting and checking if we have received the grant yet. If we instead change the meaning of the goal to build a Supply Depot, to be request resources for a Supply Depot, we may consider the goal satisfied when we send the request.

Clearly, this does not lead to a Supply Depot actually getting built, so before sending the request we add a percept to the agent with information about what will be requested and what the eventual grant should be used for. When we then finally receive the grant, we can see that the amount of resources we have been allowed to use matches with one of the percepts we previously recorded and we then perform the task stored in the percept, building a Supply Depot in this case. A task in this case is simply a Jason literal that may be translated to an action.

With this behavior, agents are no longer explicitly waiting for the grant to arrive, and are free to achieve other goals in the meantime.

```
+!request(Min, Gas, Supp, Task, X) : .my_name(Me) <- +requested(Min, Gas,
     Supp, Task, X); .send("resourceManager", tell, request(Me, Min, Gas,
     Supp, X)).

+grant(Min, Gas, Supp, X) : requested(Min, Gas, Supp, Task, X) <- -requested
     (Min, Gas, Supp, Task, X); +perform(Task, X).

+grant(Min, Gas, Supp, X) <- .print("no request made for these resources!").

+perform(buildOne(B), _) : id(MyId) & jia.findBuildingLocation(MyId, B, X, Y
     ) <- build(B, X, Y).
```

This is the implementation in Jason of the described behavior. The goal *request* is the goal that is added in place of any other goal that requires resources. As we can see, it stores the relevant information about the request and then sends a request to the resource manager.

We see that the agent also sends its own name and X in the request, where X is a value that uniquely identifies all requests made by this agent. This extra information is to help, both the resource manager but also the agent it self, keep track of requests made and grants received. Also, Jason will not trigger a plan if a percept is add that has the same name and values as an existing one, which would be the case if the same agent had to build multiple Supply Depots.

When the grant arrives, it finds the stored information, deletes it and starts executing the associated task. Currently only the *buildOne(B)* task can be successfully executed, but that is the only resource requiring task an SCV can perform.

It should be noted that Jason does support a waiting action that wait until some belief is added, *grant* in this case, which could simplify the plans for dealing with the resource manager, but the resulting behavior would be the same. The version shown here more explicitly states what happens.

6.2 Requirements for building a base

A functioning base in Starcraft requires several things.

- A Command Center, to produce additional workers and serve as a drop off point for resources.

- Workers, to gather resources and build buildings. More workers can gather resources faster.

- Training facilities, in the form of other buildings that build units for offense and defense.
- Infrastructure, in the form of Supply Depots, so that more units can be built, and Refineries so that Gas can be collected.

The base the player starts with already has a Command Center and a few workers, but all the rest is missing. Initially there is only enough supply to build an extra few workers, but to collect resources at a higher rate we will need more than a few extra. Normally, a single base should have around 30 workers, which gives the highest collection rate of resources. To be able to have this many workers, we need to build additional Supply Depots, but we also need the existing workers to gather the resources needed for both building those Supply Depots but also for training additional workers. Later on, training facilities will also have to be added.

Clearly, right from the start the need for additional workers and Supply Depots must be identified. The Command Center can go a head and request resources for building an extra worker, and continue doing this until we have enough of them. As we saw earlier, the resource manager also manages supply, and will not grant a request if there is insufficient supply. The workers need to start collecting resources, so we can build the Supply Depots we are sure to need shortly. We should only build Supply Depots when they are absolutely needed, as they are expensive and the resources could otherwise be spent on training more workers who can increase our resource collection rate.

If we leave the decision to build Supply Depots to the workers, they may all decide to do it at the same time due to having the same information. This could cause us to build more than we need in total, and will certainly cause us to build more than we need at that moment. To make this work, the workers could communicate with each other to decide who actually ends up building a supply depot, but such communication protocol would have to be very involved to work in such an asynchronous environment.

Instead, if only one agent could conclude that more Supply Depots are needed, then we could avoid this problem all together. But this approach also poses some problems. If the agent is a worker, then that worker could end up bottlenecking the construction of buildings as it would need to build all of them, and building a building takes a lot of time, and if the base gets bigger, we will certainly want to build more than one building at a time.

The agent could also tell other agents what to build instead, this would avoid bottlenecks as it could tell many different agents to build something at the same time. But then those other agents would be much less autonomous, as they could not chose to ignore such an order. The commanding agent would also need to keep track of all the other agents, to make sure that they are actually capable of completing such an order, this could also become very extensive.

If we could ask agents to perform certain tasks, rather order them to, their autonomy would be preserved as they could chose to say no. Since they should know if they are capable of performing the task it should be easy to make them say no in cases where they are not able to. Given that agents can say no, we may need to ask more than a single agent, but rather than ask one agent and wait for a response before asking the next one, we could simply ask all of them at the same time, and then chose which agent should actually perform the task amongst the agent who answered in the affirmative.

The Contract Net Protocol has this behavior, and additionally also allows for agents to specify how willing they are to perform the task, in the form of a price, which allows us to not just select a random agent who said yes, but the agent who is best able to perform the task.

6.3 Contract Net Protocol

The Contract Net Protocol (CNP) is designed for sharing tasks between agents. An agent may want to share a task with one or more other agents, in the case that it itself is not capable of performing the task or may not be able to finish it before a deadline.

There are two different roles agents can play in a CNP

- Manager. The manager of a CNP is the agent who wants to share a task.
- Contractor. The contractors of a CNP are the agent who will be offered task.

A new CNP will be created for each task that any agent may want to share, and as such, an agent can be the manager in one CNP and contractor in another.

The CNP starts with an agent who identifies a task that it needs help with, the agent then initiates the CNP with itself as the manager. The manager then announces this task to all other agents who are then considered contractors. The manager does not need to announce to all other agents, it may announce to only a subset of them if it knows which agents actually capable of ever performing the task. After the announcement, the contractors can chose to either reject or accept the offer, and if they chose to accept they must also provide a price for performing the task. The price is a measurement of the cost of the agent performing the task and may include multiple factors, such as time taken, cost of postponing its current goals. Once all the contractors have made a response to the manager, the manager then choses one or more agents amongst the agents who accepted the offer, based in their price, and tells all the contractors who was chosen.

We will use this protocol for centralizing some of the decisions about how to build our base.

6.4 Implementing CNP in Jason

Here we will see how CNP may be implemented in Jason in a way that can be used for Starcraft. We will look at what each plan does, and how this can be extended for more general uses in Starcraft.

```
hasAllResponded(Id) :- contractsSent(Id, C) & .count(propose(Id, _, _), Ps)
    & .count(refuse(Id, _), Rs) & C = Ps + Rs.

+!startCNP(Id, buildOne(B)) <- +cnpState(Id, single, propose); .findall(AN,
    friendly(AN, "Terran SCV", _, _, _), ANS); !sendContracts(Id, buildOne(B
    ), ANS).

+!sendContracts(Id, T, ANS) <- .length(ANS, C); +contractsSent(Id, C); .send
    (ANS, tell, cnp(Id, T)).

+propose(Id, O, _) : cnpState(Id, _, propose) & hasAllResponded(Id) <- .
    print("finalize begin"); !finalize(Id).
+refuse(Id, _) : cnpState(Id, _, propose) & hasAllResponded(Id) <- .print("
    finalize begin"); !finalize(Id).

@fs[atomic]
+!finalize(Id) : cnpState(Id, single, propose) <- +cnpState(Id, single,
    contracting);
                .findall(offer(O,AN), propose(Id, O, AN), Os);
                Os \== [];
                .min(Os, offer(WO, WAN));
                !announceResult(Id, Os, WAN);
                +cnpState(Id, single, done);
                -propose(Id, _, _);
                -refuse(Id, _, _).
```

- *hasAllResponded(id)*. This is not a plan but a rule, which is used for checking if all the contractors in the CNP have responded. The id parameter is the id of the CNP to check for.

- *+!startCNP(Id, buildOne(B))*. When this goal is added, the plan initiates a CNP by announcing it to all the SCVs. This specific goal is added when a single building is to be built, other goals can be added for other tasks, as they may require announcing to different agents or the CNP should be of a different type. We see that we also add a literal *cnpState*, with the id of the CNP, the type of CNP and the state of the CNP. The type is an addition to allow for selecting how many contractors it should chose to perform the task, in this case only a single contractor should be chosen as only one is needed for building a building. The state is for keeping track of which step in the protocol we are currently at.

- *+propose(Id, O, _)*. This belief is added when a contractor accepts the task. The plan checks if all contractors have now responded and proceeds to the next step if they have. The id here is again that of the CNP, 'O' is the contractors price and the third argument is the name of the contractor.

- *+refuse(Id, _)*. Similar to the *propose* belief, this is added in the case a contractor refuse the task. Again we must check if this was the last contractor to respond.

- *+!finalize(Id).* This goal is added after all contractors have responded. A plan exists for each type of CNP, here we look at the single type. We look through all the contractors who accepted the task, and chose the one with the lowest cost as the winner. This result is then announced to the winner.

As we can see, there are modifications to the protocol, namely the addition of a type and the result of the CNP is only announced to any winners.

The type is useful for tasks such as resource collection, as we are not interested in only having a single agent collect resources for us. For collecting minerals, the CNP would use the 'all' type, which blindly accepts all contractors accepted the task. An alternative to this would be to accept all contractors with a price below some threshold, this would allow agent who considered themselves to be doing something more important to still accept the task, but at a very high price. Currently, the agents who can collect resources, the workers, simply reject the task if they think their current goal is more important.

Only announcing the result to the winners is just a simplification, agents who have accepted the task go about their business as they normally would, telling them that they lost the CNP would not change this.

6.5 Building a base

With the CNP in place, we can have our Command Center be in charge of ensuring that the essential buildings are built. Since there is only one Command Center, at least initially, it becomes trivial to avoid situations where multiple agents conclude that they must build the same building. Additional Command Centers may be build, such that more than one exists, which would lead us to have this same problem once again, but in this case it is not a problem to simply only have one of the Command Centers making decisions about buildings. Recall that the problem with designating a single worker agent to take care of building all the buildings, could lead to bottlenecking how fast we could construct buildings, but since the Command Center uses the CNP to issue contracts, there is no bottleneck anymore.

As we initially stated, we also wanted our agents to remain autonomous. By using the CNP, our agents may decide for themselves if they want to bid on any contracts, thereby preserving their autonomy. The base that the agents build is a good starting point for a fully developed base, but it still lacks many of the more advanced buildings needed for winning a game of Starcraft, but with the behavior of the agents it is easily possible to add goals of building these buildings.

The current implementation will also only build the initial infrastructure of the base, but does very little to provide any sort of defense against attacks from the opposing player, but similarly to how the Command Center builds SCVs, other buildings that built offensive units can be made to some to help defend the base and in the end conquer the enemy.

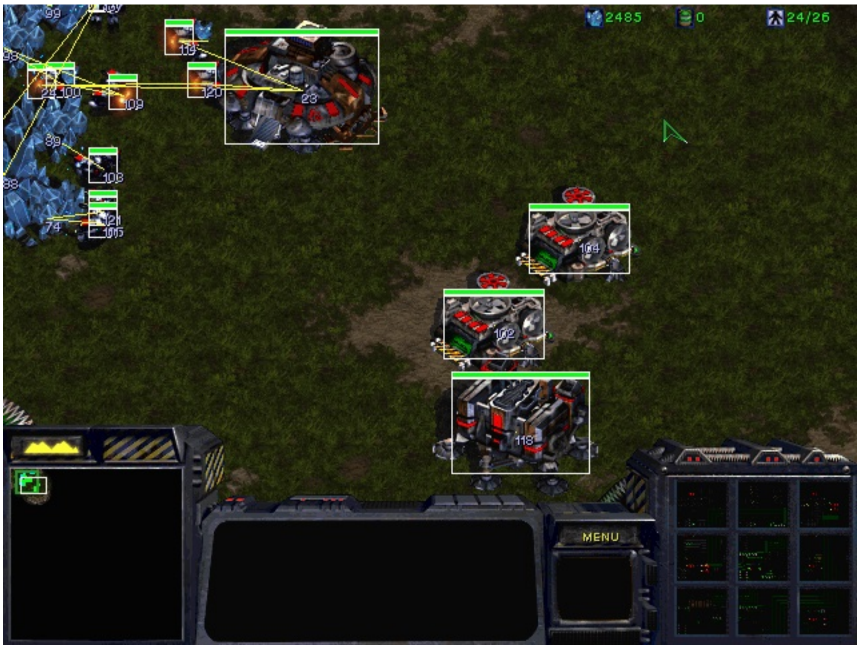


Figure 6.1: A screen capture showing a base built by the agents.

CHAPTER 7
Conclusion

An implementation for allowing Jason to interact with Starcraft has been produced. This implementation has successfully been used to create agents in Jason that are capable of building the basic infrastructure of a base in Starcraft. Furthermore, agents have also been implemented that are capable of dividing themselves into groups who cooperate in completing goals, this can be further extended to other areas of Starcraft using the same strategy.

The implementation is not complete however, many of the common actions used in Starcraft are supported and the perceivers that have been implemented cover some of the information agents need to carry out their goals. The implementation makes it easy to add more actions and more perceivers to it, as the need for them arise. The JNI BWAPI still lacks some of the useful functions present in the C++ version of it, so if some of these functions are needed for future perceivers or actions, the JNI BWAPI will have to be extended for this.

With the ability to use Jason for programming agents in Starcraft, it is easier to use Starcraft as a platform for testing new theories in Artificial Intelligence.

Appendix

```
/* Controls access to Mineral and Gas resources */
peek(Z) :- queue(Q) & .reverse(Q, [entry(AN,M,G,S,X)|_]) & Z = entry(AN, M,
    G, S, X).
queue([]).
spent(0 ,0).

@a[atomic] +!enqueue(entry(AN, M, G, S, X)) : queue(Q) <- -+queue([entry(AN,
     M, G, S, X)|Q]).
@b[atomic] +!enqueue(_) <- .print("enqueued wrong type!!!!!!!!").

@c[atomic] +!dequeue(Z) : queue(Q) & .reverse(Q, [entry(AN, M, G, S, X)|T])
    <- Z = entry(AN, M, G, S, X); -+queue(T).
@d[atomic] +!dequeue(_) <- .print("dequeue failed??!!!!!!").

@e[atomic] +!peek(Z) : queue(Q) & .reverse(Q, [entry(AN,M,G,S,X)|_]) <- Z =
    entry(AN, M, G, S, X).
@f[atomic] +!peek(_).

@g[atomic]+request(AN, M, G, S, X) <- .print("request recieved"); !enqueue(
    entry(AN, M, G, S, X)); !checkQueue.

@h[atomic]+totalRes(A, B, C, D, E, F) <- !checkQueue.

@i[atomic] +!checkQueue : totalRes(_, MT, _, GT, SC, ST) & peek(entry(AN, M,
     G, S, X)) &
                spent(MU, GU) &
                M <= MT - MU &
                G <= GT - GU &
                S <= ST - SC
                <-  !dequeue(_);
                    -+spent(MU + M, GU + G);
                    .send(AN, tell, grant(M, G, S, X)); !checkQueue.
@j[atomic] +!checkQueue.

/*
.print(M, ":", MT, ":", MU, " GAS:", G, ":", GT, ":", GU, " SUPPLY:", S, ":"
    , SC, ":", ST)
*/
```

”../Java/Branch 2 JNI v0.4/BroodwarAgents/src/asl”/player.asl

```
!reason.

+!reason
```

```
  : queue(Q)
  & Q < 2
  & minerals(M) & M >= 50
  & supply(X,Y) & X < Y
  <- train("Terran_Marine"); .wait(500); !reason.
+!reason <- .wait(1000); !reason.
```

"../Java/Branch 2 JNI v0.4/BroodwarAgents/src/asl"/terranBarracks.asl

```
hasAllResponded(Id) :- contractsSent(Id, C) & .count(propose(Id, _, _), Ps)
    & .count(refuse(Id, _), Rs) & C = Ps + Rs.

counter(0).

+!incCounter(X): counter(C) <- X = C + 1; -+counter(X).

+gameStart <- .wait(200); !!buildSCVs; !!keepWorkersBusy.

+!startCNP(Id, gatherM, []) <- +cnpState(Id, all, propose); .findall(AN,
    friendly(AN, "Terran SCV", _, _, _), ANS); !sendContracts(Id, gatherM,
    ANS).
+!startCNP(Id, gatherM, ANS) <- .print("cnp got list of agents"); +cnpState(
    Id, all, propose); !sendContracts(Id, gatherM, ANS).
+!startCNP(Id, buildOne(B)) <- +cnpState(Id, single, propose); .findall(AN,
    friendly(AN, "Terran SCV", _, _, _), ANS); !sendContracts(Id, buildOne(B
    ), ANS).

+!sendContracts(Id, T, ANS) <- .length(ANS, C); +contractsSent(Id, C); .send
    (ANS, tell, cnp(Id, T)).

+propose(Id, O, _) : cnpState(Id, _, propose) & hasAllResponded(Id) <- .
    print("finalize begin"); !finalize(Id).
+refuse(Id, _) : cnpState(Id, _, propose) & hasAllResponded(Id) <- .print("
    finalize begin"); !finalize(Id).


+supply(N,M) : (M - 6) <= N & 400 > M  <- !incCounter(X); !startCNP(X,
    buildOne("Terran Supply Depot")).
+supply(N,M) : N > 24 & not friendly(_, "Terran Barracks", _, _, _) <- !
    incCounter(X); !startCNP(X, buildOne("Terran Barracks")).


@fs[atomic]
+!finalize(Id) : cnpState(Id, single, propose) <- +cnpState(Id, single,
    contracting);
                .findall(offer(O,AN), propose(Id, O, AN), Os);
                Os \== [];
                .min(Os, offer(WO, WAN));
                !announceResult(Id, Os, WAN);
                +cnpState(Id, single, done);
                -propose(Id, _, _);
                -refuse(Id, _, _).

@fa[atomic]
```

```
+!finalize(Id) : cnpState(Id, all, propose) <- +cnpState(Id, all, contracting);
                .findall(AN, propose(Id, _, AN), ANS);
                ANS \== [];
                !announceWinners(Id, ANS);
                +cnpState(Id, all, done);
                -propose(Id, _, _);
                -refuse(Id, _, _).

@ff[atomic]
-!finalize(Id) <- .print("finalize failed, no offers").

@f +!finalize(Id).

+!announceWinners(Id, ANS) <- .send(ANS, tell, accept(Id)).

+!announceResult(_, [], _).
+!announceResult(Id, [offer(O,WAN)|T], WAN) <- .send(WAN, tell, accept(Id)); !announceResult(Id, T, WAN).
+!announceResult(Id, [offer(O, LAN)|T], WAN) <- .send(LAN, tell, reject(Id)); !announceResult(Id, T, WAN).

/* Build SCVs until there are enough of them */

+!buildSCVs : queue(0) & .count(friendly(_, "Terran SCV", _, _, _), C) & C < 24 & not requested(_, _, _, buildSCV, _) & counter(X) <- !incCounter(_); !request(50, 0, 2, buildSCV, X); .wait(500); !buildSCVs.
+!buildSCVs : queue(N) & N > 0 <- .wait(500); !buildSCVs.
+!buildSCVs : requested(_, _, _, buildSCV, _) <- .wait(2000); !buildSCVs.
+!buildSCVs.

+!request(M, G, S, T, X) : .my_name(Me) <- .print("request sent"); +requested(M, G, S, T, X); .send("player", tell, request(Me, M, G, S, X)).
+grant(M, G, S, X) : requested(M, G, S, T, X) <- .print("grant received"); -requested(M, G, S, T, X); !execute(T).
+grant(M, G, S, X) <- .print("no request made for these resources!").

+!execute(buildSCV) <- train("Terran SCV").

+!keepWorkersBusy : .findall(AN, idleWorker(AN), ANS) & ANS \== [] <- !incCounter(X); !startCNP(X, gatherM, ANS); .wait(2000); !keepWorkersBusy.
+!keepWorkersBusy <- .wait(2000); !keepWorkersBusy.
```

"../Java/Branch 2 JNI v0.4/BroodwarAgents/src/asl"/terranCommandCenter.asl

```
weigh(EID, W) :- .count(attacking(_ ,EID), N) & id(ID) & jia.distance(ID, EID, D) & W = w(EID, N, D).

+gameStart <- .random(X); .wait(500 * X); !!attack.

+!attack : id(ID) & attacking(ID, T)
    & weigh(T, w(T, TN, TD))
    & .findall(t(N, D, EID), (enemy(_, EID, _, _) & weigh(EID, w(EID, N, D))
         & N + 1 < TN), WS)
    & .min(WS, t(_, _, EID))
        <- attack(EID); !attack.

+!attack : id(ID) & not attacking(ID, _)
    & .findall(w(N, D, EID), (enemy(_, EID, _, _) & weigh(EID, w(EID, N, D))
        ), WS) & .min(WS, w(_, _, EID))
        <- attack(EID); !attack.

+!attack <- .wait(300); !attack.
```

"../Java/Branch 2 JNI v0.4/BroodwarAgents/src/asl"/terranMarine.asl

```
distance(X, Y, XX, YY, D) :- D = (X - XX) + (Y - YY).

cost("Terran Supply Depot", M, G) :- M = 100 & G = 0.
cost("Terran Barracks", M, G) :- M = 150 & G = 0.
cost(B, M, G) :- false.

price(gatherM, X) :- X = 1.
price(buildOne(B), X) :- X = 3.
price(T, X) :- false.

counter(0).
+!incCounter(X): counter(C) <- X = C + 1; -+counter(X).

+cnp(Id, buildOne(B))[source(AN)] : not constructing & position(X, Y) & id(
    MyId) &
                                    jia.findBuildingLocation(MyId, B, XX, YY
                                        ) & jia.tileDistance(X, Y, XX, YY, D
                                        )
                                    <- +offer(Id, AN, buildOne(B), D); .
                                        print(D); .my_name(Me); .send(AN,
                                        tell, propose(Id, D, Me)).
+cnp(Id, gatherM)[source(AN)] : not constructing & price(T, O) <- +offer(Id,
     AN, T, O); .my_name(Me); .send(AN, tell, propose(Id, O, Me)).
+cnp(Id, T)[source(AN)] <- .print("refused ", T); .my_name(Me); .send(AN,
    tell, refuse(Id, Me)).

+accept(Id)[source(AN)] : offer(Id, AN, T, _) <- !execute(T).
+accept(Id)[source(AN)] <- .print("cannot accept, refused cpf: ", Id).
+reject(Id)[source(AN)] <- -offer(Id, AN, _, _).

+!execute(buildOne(B)) : cost(B, M, G) <- !incCounter(X); !request(M, G, 0,
    buildOne(B), X).
+!execute(gatherM) : .findall(Id, mineralField(Id, _, _), L) & .shuffle(L,S)
     & .member(X,S) <- gather(X).
+!execute(T) <- .print("fail exe").

+!request(M, G, S, T, X) : .my_name(Me) <- .print("request sent"); +
    requested(M, G, S, T, X); .send("player", tell, request(Me, M, G, S,X)).
+grant(M, G, S, X) : requested(M, G, S, T, X) <- .print("grant received"); -
    requested(M, G, S, T, X); +granted(T, X).
+grant(M, G, S, X) <- .print("no request made for these resources!").
```

```

+granted(buildOne(B), _) : id(MyId) & jia.findBuildingLocation(MyId, B, X, Y
    ) <- build(B, X, Y).
-granted(_, _) <- .print("granted failed?!").
```

"../Java/Branch 2 JNI v0.4/BroodwarAgents/src/asl"/terranSCV.asl

```
package eisbw;

import eis.*;
import eis.exceptions.*;
import eis.iilang.*;
import eisbw.actions.*;
import eisbw.percepts.GameStartPercept;
import eisbw.percepts.perceivers.*;
import eisbw.units.*;
import java.util.*;
import java.util.Map;
import java.util.logging.*;
import jnibwapi.*;
import jnibwapi.types.UnitType;
import jnibwapi.util.BWColor;

public class BWAPIBridge extends EIDefaultImpl {

    // TODO: Check 64bit version of client bridge
    private static final Logger logger = Logger.getLogger(BWAPIBridge.class.
        getName());
    public static final int TRAINING_QUEUE_MAX = 5;
    private final Thread apiThread;
    private static JNIBWAPI bwapi;
    private BWApiUtility bwApiUtility;
    private final StarcraftUnitFactory unitFactory;
    private final Map<String, Unit> units;
    private final Map<Integer, String> unitNames;
    private List<Percept> mapPercepts = null;
    private final ActionProvider actionProvider;
    private boolean gameStarted = false;

    private Map<Unit, Action> pendingActions = new HashMap<>();

    public static JNIBWAPI getGame() {
        return bwapi;
    }

    public static void main(String[] args) throws ManagementException {
        BWAPIBridge env = new BWAPIBridge();
        env.init(Collections.EMPTY_MAP);
    }

    public BWAPIBridge() {
        super();
        this.units = new HashMap<>();
        this.unitNames = new HashMap<>();
        bwapi = new JNIBWAPI(bwApiListener, true);
```

```java
        this.bwApiUtility = new BWApiUtility(bwapi);
        this.unitFactory = new StarcraftUnitFactory(bwapi, this.bwApiUtility
            );
        this.actionProvider = new ActionProvider();
        actionProvider.loadActions(bwapi);

        apiThread = new Thread() {
            @Override
            public void run() {
                bwapi.start();
            }
        };
    }

    @Override
    public void init(Map<String, Parameter> parameters) throws
        ManagementException {
        super.init(parameters);
        try {
            addEntity("player");
        } catch (EntityException ex) {
            Logger.getLogger(BWAPIBridge.class.getName()).log(Level.SEVERE,
                null, ex);
        }
        apiThread.start();
    }

    @Override
    protected LinkedList<Percept> getAllPerceptsFromEntity(String entity)
        throws PerceiveException, NoEnvironmentException {
        LinkedList<Percept> percepts = new LinkedList<>();
        if (!gameStarted) {
            return percepts;
        }

        if (mapPercepts != null) {
            percepts.addAll(mapPercepts);
        }

        Unit unit = units.get(entity);
        if (unit != null) {
            StarcraftUnit scu = this.unitFactory.Create(unit);
            percepts.addAll(scu.perceive());
        } else if ("player".equals(entity)) {
            percepts.addAll(new TotalResourcesPerceiver(bwapi).perceive());
        }

        Map<UnitType, Integer> count = new HashMap<>();
        for (Unit myUnit : bwapi.getMyUnits()) {
            UnitType unitType = myUnit.getType();
            if (!count.containsKey(unitType)) {
                count.put(unitType, 0);
            }
            count.put(unitType, count.get(unitType) + 1);
        }
        for (UnitType unitType : count.keySet()) {
            percepts.add(new Percept("unit", new Identifier(unitType.getName
                 ()), new Numeral(count.get(unitType))));
        }

        for (Unit u : bwapi.getNeutralUnits()) {
            UnitType unitType = u.getType();
            if (u.isVisible()) {
                if (UnitTypesEx.isMineralField(unitType)) {
                    Percept p = new Percept("mineralField");
                    p.addParameter(new Numeral(u.getID()));
                    p.addParameter(new Numeral(u.getResources()));
                    p.addParameter(new Numeral(u.getResourceGroup()));
                    percepts.add(p);
                } else if (UnitTypesEx.isVespeneGeyser(unitType)) {
                    Percept p = new Percept("vespeneGeyser");
                    p.addParameter(new Numeral(u.getID()));
                    p.addParameter(new Numeral(u.getResources()));
                    p.addParameter(new Numeral(u.getResourceGroup()));
                    percepts.add(p);
                } else if (UnitTypesEx.isRefinery(unitType)) {
                    Percept p = new Percept("refinery");
                    p.addParameter(new Numeral(u.getID()));
                    p.addParameter(new Numeral(u.getResources()));
                    p.addParameter(new Numeral(u.getResourceGroup()));
                    percepts.add(p);
                }
            }
        }

        percepts.add(new GameStartPercept());

        return percepts;
    }

    @Override
    protected boolean isSupportedByEnvironment(Action action) {
        return actionProvider.getAction(action.getName()) != null;
    }

    @Override
    protected boolean isSupportedByType(Action action, String string) {
        return true;
    }

    @Override
    protected boolean isSupportedByEntity(Action act, String name) {
        Unit unit = units.get(name);
        String actionName = act.getName();

        StarcraftAction action = actionProvider.getAction(actionName);

        // if action is invalid, we need to provide a failure message (which
             can only be provided by performEntityAction, so return true in
            that case)
        return !action.isValid(act) || action.canExecute(unit, act);
    }

    @Override
    protected synchronized Percept performEntityAction(String name, Action
        act) throws ActException {
        Unit unit = units.get(name);

        // cant act during construction
        if (unit.isBeingConstructed()) {
            return null;
        }

        if (!pendingActions.containsKey(unit)) {
            String actionName = act.getName();

            StarcraftAction action = actionProvider.getAction(actionName);
            // Action might be invalid
            if (action.isValid(act)) {
                logger.info("[" + name + "] Pending action: " + act.toProlog
                    ());
                pendingActions.put(unit, act);
            } else {
                throw new ActException(ActException.FAILURE, "Action must be
                     of the form " + action.toString() + " (was " + act.
                    toProlog() + ").");
            }

        }
        return null;

    }

    public void register(Unit u) throws RuntimeException {
        String unitName = bwApiUtility.getUnitName(u);
        units.put(unitName, u);
        unitNames.put(u.getID(), unitName);
        try {
            addEntity(unitName);
        } catch (EntityException ex) {
            throw new RuntimeException(ex);
        }
    }

    @Override
    public String requiredVersion() {
        return "0.3";
    }
    private final BWAPIEventListener bwApiListener = new BWAPIEventListener
        () {
        @Override
        public void connected() {

        }

```

```java
        @Override
        public void matchStart() {
            logger.info("Game started...");

            // set game speed to 30 (0 is the fastest. Tournament speed is
                20)
            // You can also change the game speed from within the game by "/
                speed X" command.
            bwapi.setGameSpeed(5);
            bwapi.enableUserInput();

            bwapi.drawIDs(true);
            bwapi.drawHealth(true);
            bwapi.drawTargets(true);

            mapPercepts = new ArrayList<>();
            gameStarted = true;
            //jnibwapi.Map map = bwapi.getMap();
        }

        @Override
        public void matchFrame() {
            synchronized (BWAPIBridge.this) {
                Iterator<Unit> it = pendingActions.keySet().iterator();
                while (it.hasNext()) {
                    Unit unit = it.next();
                    Action act = pendingActions.get(unit);

                    String actionName = act.getName();

                    StarcraftAction action = actionProvider.getAction(
                        actionName);
                    logger.info("[" + bwApiUtility.getUnitName(unit) + "]
                        Performing action: " + act.toProlog());
                    try {
                        action.execute(unit, act);
                    } catch (ActException ex) {
                        logger.log(Level.WARNING, "Could not execute " + act
                            .toProlog(), ex);
                    }

                    it.remove();
                }
            }
//              for (int x = 0; x < bwapi.getMap().getSize().getBX(); x++) {
//                  for (int y = 0; y < bwapi.getMap().getSize().getBY(); y++)
     {
//                      bwapi.drawBox(new Position(x, y, Position.PosType.
    BUILD), new Position(x + 1, y + 1, Position.PosType.BUILD), BWColor.
    Yellow, false, false);
//                  }
//              }

        }
        @Override
        public void keyPressed(int i) {
        }

        @Override
        public void playerLeft(int i) {
        }

        @Override
        public void nukeDetect() {
        }

        @Override
        public void unitDiscover(int i) {
        }

        @Override
        public void unitEvade(int i) {
        }

        @Override
        public void unitShow(int i) {
        }

        @Override
        public void unitHide(int i) {
        }

        @Override
        public void unitCreate(int i) {
            Unit u = bwapi.getUnit(i);
            if (bwapi.getMyUnits().contains(u)) {
                register(u);
            }
        }

        @Override
        public void unitDestroy(int i) {
            if (unitNames.containsKey(i)) {
                String unitName = unitNames.get(i);
                units.remove(unitName);
                try {
                    deleteEntity(unitName);
                } catch (EntityException | RelationException ex) {
                    throw new RuntimeException(ex);
                }
            }
        }

        @Override
        public void unitMorph(int i) {
        }

        @Override
        public void sendText(String text) {
        }

        @Override
        public void receiveText(String text) {
        }

        @Override
        public void unitRenegade(int unitID) {
        }

        @Override
        public void saveGame(String gameName) {
        }

        @Override
        public void unitComplete(int unitID) {
        }

        @Override
        public void playerDropped(int playerID) {
        }

        @Override
        public void matchEnd(boolean winner) {
//            throw new UnsupportedOperationException("Not supported yet.");
     //To change body of generated methods, choose Tools | Templates.
        }

        @Override
        public void nukeDetect(Position pstn) {
            //throw new UnsupportedOperationException("Not supported yet.");
                 //To change body of generated methods, choose Tools |
                Templates.
        }
    };
}
```

"../Java/Branch 2 JNI v0.4/EISBW/src/eisbw"/BWAPIBridge.java

```
package eisbw.actions;

import eis.exceptions.ActException;
import eis.iilang.*;
import java.util.LinkedList;
import jnibwapi.*;
import jnibwapi.types.UnitType;

public class Build extends StarcraftAction {

    public Build(JNIBWAPI api) {
        super(api);
    }

    @Override
    public boolean isValid(Action action) {
```

```java
        LinkedList<Parameter> parameters = action.getParameters();
        if (parameters.size() == 3) {
            return parameters.get(0) instanceof Identifier && utility.
                getUnitType(((Identifier) parameters.get(0)).getValue()) !=
                null
                    && parameters.get(1) instanceof Numeral
                    && parameters.get(2) instanceof Numeral;
        }
        return false;
    }

    @Override
    public boolean canExecute(Unit unit, Action action) {
        return unit.getType().isWorker();
    }

    @Override
    public void execute(Unit unit, Action action) throws ActException {
        LinkedList<Parameter> params = action.getParameters();
        String type = ((Identifier) params.get(0)).getValue();
        int tx = ((Numeral) params.get(1)).getValue().intValue();
        int ty = ((Numeral) params.get(2)).getValue().intValue();
        //UnitType unitType = unit.getType();

    // TODO: Check that it is a building
        boolean result = api.build(unit.getID(), tx, ty, utility.getUnitType
            (type).getID());
        if (!result) {
            throw new ActException(ActException.FAILURE);
        }
    }

    @Override
    public String toString() {
        return "build(Type, X, Y)";
    }
}
```

"../Java/Branch 2 JNI v0.4/EISBW/src/eisbw"/actions/Build.java

```java
package eisbw.units;

import eis.iilang.Percept;
import eisbw.percepts.perceivers.IPerceiver;
import java.util.ArrayList;
import java.util.List;
import jnibwapi.JNIBWAPI;
import jnibwapi.Unit;

public class StarcraftUnit {

    protected final Unit unit;
    protected final JNIBWAPI api;
    protected final List<IPerceiver> perceivers;

    public StarcraftUnit(JNIBWAPI api, Unit unit, List<IPerceiver>
        perceivers) {
        this.api = api;
        this.unit = unit;
        this.perceivers = perceivers;
    }

    public List<Percept> perceive() {
        ArrayList<Percept> percepts = new ArrayList<>();
        for (IPerceiver perceiver : this.perceivers) {
            percepts.addAll(perceiver.perceive());
        }
        return percepts;
    }
}
```

"../Java/Branch 2 JNI v0.4/EISBW/src/eisbw"/units/StarcraftUnit.java

```
package eisbw.units;

import eisbw.*;
import eisbw.percepts.perceivers.*;
import eisbw.units.terran.*;
import eisbw.units.terran.buildings.*;
import java.util.*;
import jnibwapi.*;
import jnibwapi.types.UnitType.UnitTypes;

public class StarcraftUnitFactory {

    private final JNIBWAPI api;
    private final BWApiUtility util;

    public StarcraftUnitFactory(JNIBWAPI api, BWApiUtility util) {
        this.api = api;
        this.util = util;
    }

    public StarcraftUnit Create(Unit unit) {
        ArrayList<IPerceiver> perceptGenerators = new ArrayList<>();
        perceptGenerators.add(new GenericUnitPerceiver(api, unit));
        perceptGenerators.add(new MapPerceiver(api));
        perceptGenerators.add(new EnemyPerceiver(api));
        perceptGenerators.add(new PlayerUnitsPerceiver(api, util));

        String un = unit.getType().getName();
        if (un.equals(UnitTypes.Terran_Command_Center.getName())) {
            perceptGenerators.add(new AvailableResourcesPerceiver(api));
            perceptGenerators.add(new IdleWorkersPerceiver(api, util));
            perceptGenerators.add(new QueuePerceiver(this.api, unit));
            perceptGenerators.add(new IdleWorkersPerceiver(api, util));
            return new CommandCenter(api, unit, perceptGenerators);
        } else if (un.equals(UnitTypes.Terran_Barracks.getName())) {
            perceptGenerators.add(new QueuePerceiver(this.api, unit));
            return new Barracks(api, unit, perceptGenerators);
        } else if (un.equals(UnitTypes.Terran_SCV.getName())) {
            perceptGenerators.add(new BuilderUnitPerceiver(api, unit));
            perceptGenerators.add(new GathererUnitPerceiver(api, unit));
            return new SCV(api, unit, perceptGenerators);
        } else if (un.equals(UnitTypes.Terran_Marine.getName())) {
            perceptGenerators.add(new StimUnitPerceiver(api, unit));
            perceptGenerators.add(new AttackingUnitsPerceiver(api));
            return new Marine(api, unit, perceptGenerators);
        } else {
            return new StarcraftUnit(api, unit, perceptGenerators);
        }
    }
}
```

"../Java/Branch 2 JNI v0.4/EISBW/src/eisbw"/units/StarcraftUnitFactory.java

```
package eisbw.percepts.perceivers;

import eis.iilang.Percept;
import eisbw.percepts.EnemyPercept;
import java.util.*;
import jnibwapi.*;

public class EnemyPerceiver extends Perceiver {

    public EnemyPerceiver(JNIBWAPI api) {
        super(api);
    }

    @Override
    public List<Percept> perceive() {
        ArrayList<Percept> percepts = new ArrayList<>();
        List<Unit> enemies = api.getEnemyUnits();

        for (Unit unit : enemies) {
            percepts.add(new EnemyPercept(api.getUnitType(unit.getTypeID()).
                getName(), unit.getID(), unit.getX(), unit.getY()));
        }

        return percepts;
    }
}
```

"../Java/Branch 2 JNI v0.4/EISBW/src/eisbw"/percepts/perceivers/enemyPerceiver.java

Bibliography

[1] Tristan Behrens, Environment Interface Standard, `http://apleis.sourceforge.net/`, 2009, accessed 2014/6/24

[2] Jomi F. Hübner & Rafael H. Bordini, Jason, `http://jason.sourceforge.net/wp/`, accessed 2014/6/24

[3] BWAPI, `https://code.google.com/p/bwapi/`, accessed 2014/6/24

[4] JNI BWAPI, `https://code.google.com/p/jnibwapi/`, accessed 2014/6/24

[5] Blizzard Entertainment, Starcraft: Brood War, `http://eu.blizzard.com/en-gb/games/sc/`, 1998, accessed 2014/6/24

[6] Rafael H. Bordini & Jomi Fred Hübner & Michael Woolridge, Programming multi-agent systems in AgentSpeak using Jason, Wiley, 2007

[7] Michael Woolridge, An Introduction to MultiAgent Systems, Wiley, 2009