Agent-Oriented Programming Framework Design for Real-Time Strategy AI

XXXXXX and **XXXXXXX** and **XXXXXXX** and **XXXXXXX**
123 Alphabet Lane
Somewheresville, NotYourState 11111-2222

Abstract

In video game development, one of the many challenges for developers is creating a challenging AI for the user. Failure to create effective AI can cause the popularity of a game to decrease. This can be particularly noticeable in Real-Time Strategy (RTS) games, where AI can either be too challenging or too predictable. To address this problem, we present a Multi-Agent System Framework, KhasBot, to handle the Artificial Intelligence in a specific RTS Game, StarCraft: BroodWar.

We approach the AI model as a subset of individual Agents who are capable capable of solving the common AI problems found in RTS games, for example, path-finding, resource allocation, task management and reactive planning. This framework will be tested in the SC AI 2011 competition. In this paper we will discuss our reasoning for selecting JADE as our AI backbone, our design strategy, and our creation process in creating KhasBot.

Introduction

Real-Time Strategy (RTS) Games are an area of interest in Artificial Intelligence (AI) for many reasons. They must adapt quickly to changing environments, make decisions on non-complete information, and involve multiple complex AI issues in general (for example, reactive planning, path-finding, and resource scheduling). StarCraft, an RTS created by Blizzard in 1998, involves three different alien races in a battle for resources and superiority on a large variety of maps. A human player of this game must keep track of many things at once, such as resources, map locations, building placements and army units, including enemy player actions and locations. The role of the player is to command their force of units and structures through the use of mouse and keyboard; or in the case of our AI, the Broodwar API (BWAPI). Some players operate at a level of 200-300 actions, or commands, per minute (APM) at very high levels of play. Our AI player will be designed to emulate high level players in the context of unit control and strategy.

Controlling these units and structures throughout the game creates a dynamic and complex environment that poses a great challenge in creating an AI that can compete at expert levels. For our AI, we selected one race to operate, the Protoss. To get our AI functioning we will load it with pre-set build orders, a particular battle plan, and the means to execute those orders and plans. Based on the enemy's race selection and map layout, our AI will proceed with one of the appropriate build orders. As the game progresses, based on scouting information and enemy actions, we will change our build order accordingly in an effort to find the most effective way to defeat our opponent. KhasBot, the name of our AI, at this stage is being built for one-on-one battles. For the remainder of the paper, we will discuss the background of the BWAPI, JADE and RTS games, our design strategy and implementation, and finally our results and future work.

Background

To approach this problem, we combine two different frameworks together, the BWAPI and the Java Agent DEvelopment Framework (JADE) for use in Real-Time Strategy games. StarCraft does not have an officially released AI API, so the BWAPI is injected into StarCraft at runtime and through the use of a specially designed proxy client, JADE can interact externally with the game. BWAPI and JADE are described in further detail below.

Real-Time Strategy

Real-time strategy (RTS) games normally involve players commanding an army of units to build structures, collect resources and attack enemy players. The player accomplishes this by issuing orders, normally through the keyboard and mouse on a computer, to individual or groups of units and structures. By collecting needed resources on the map, the player can continue to build up their forces and bases. For example, in StarCraft, the player would issue commands to a worker unit to collect minerals and vespene gas to be able to command

another worker unit to build new structures or to order a building to create more units. Through the use of effective map knowledge and time, the player can create a strategy that will lead to victory, which usually involves defeating the enemy forces, capturing a target item or obtaining a certain resource goal. For the StarCraft games our AI will be participating in, it will be focused on one-on-one matches with the goal of eliminating the enemy from the map.

BWAPI

An issue with community development of AI for RTS games, and many games in general, is that many of them are closed source. Users cannot directly program logic into the game and must use external tools to accomplish this. For StarCraft, community members developed the BWAPI for directly injecting the logic into the game. The BWAPI is an open-source Application Programming Interface (API) that enables AI creation for Starcraft: Broodwar.[1] It was developed in C++, but many extensions have allowed access to other languages such as Python, C#, and Java, which we use for our own AI.

The BWAPI breaks down the StarCraft game into 5 basic objects:

- *Game*: Holds all information about the current game being played, including known unit locations, resource nodes, etc.
- *Player*: Holds all information a player would have, for instance current resources, buildings, and controllable units.
- *Unit*: Represents a piece in the game, be it a mineral, building, or army unit.
- *Bullet*: Represents a projectile from an attack from units that are capable of ranged attacks.
- *Force*: Represents a collection of players attempting a singular objective. This is similar to a co-operative description, where some players may be on the same team.

In addition to the game objects, a game control module *AIModule.cpp* is injected as a *dll* into the core StarCraft code. This module contains the reasoning needed to process and execute commands in StarCraft. This module can also use sockets to communicate outside of the game with other AI. For example, KhasBot is an external Java program running along with StarCraft, and our AI communicates through ProxyBot, which then relays our information to the AIModule. An analogy to this can be that BWAPI is a tool-box, and the AIModule is a untrained worker on location with a radio, which is the ProxyBot. Communicating through this radio is KhasBot, giving orders to the worker on how to operate the tools.

The basic flow of how BWAPI works is as follows, when the actual game begins BWAPI will call upon the *onStart()* method from the *AIModule* which will setup all the above objects with the beginning of the game information. After this, an *onFrame()* method is called every frame of the game. This is the looping method which holds all the reasoning and computational information of the actual AI. The *AIModule* returns orders to the StarCraft game via *UnitCommand*s. By giving units in control of the AI player various actions or commands (such as building, attacking, researching, and upgrading) the game can progress towards a climax where either player can obtain victory. Victory is achieved when the AI player has eliminated the enemy force from the map.

Agent-Oriented Programming

Agent-Oriented Programming is a programming paradigm started back in the 1990s. The basic premise is to focus on the concept of 'agents' as opposed to objects in Object-Oriented Programming. By doing this, one can focus more readily on the reasoning processes and communication paths of a system, rather than method calls (**?**). Aside from communication, agents can have a focused goal or set of goals to accomplish.

The basic flow of Agent-Oriented Programming, which our KhasBot is designed upon, is that agents hold a set of beliefs and then based off changes to them, the agents belief set will change, causing a possible reaction or change in the execution, or attempted actions of said Agent.

For this particular framework, we used JADE to model our agents. JADE is a Java middle-ware framework designed to handle and run agents developed from a package given by JADE. JADE also handles running the agent platform along with all the communication (**?**).

JADE

The basic setup of JADE is that all applications are an agent and as such extends the *Agent* class. These Agents perform actions based on their behaviours, which is another JADE class *Behaviours*.[2]

Behaviours can be several different things: some behaviours only happen once, some happen over a period of time, and others happen at certain intervals. These are expressed through the three JADE Behaviour classes: *OneShotBehaviour*, *CyclicBehaviour*, and *TickerBehaviour*, respectively.

On top of having a core behaviour, most agents have multiple additional behaviours. JADE is by design, non-preemptive, that is while one behaviour is executing, the others will wait for that one to complete. In order to achieve more complicated behaviours and allow multiple behaviours to work together, JADE offers some additional behaviour classes. These combine the simple behaviours in various ways such as doing a list of behaviours, having a state machine of behaviours, doing

[1] `http://code.google.com/p/bwapi`

[2] As the JADE Framework uses the British spelling for behaviour, we will use that version for continuity.

the behaviours all at the same time, or having the behaviours be individual threads. The last is what we use in Khasbot, which is the *ThreadedBehaviours* class.

To keep the information related to multiple and possible threaded behaviours the same, JADE employs a centralized data concept. JADE has an object called *DataStore* which works just as it sounds, each agent has a *DataStore* that works like a hash-table to store information as it runs. By making all the behaviours use the same *DataStore* we can keep everything up to date.

Another important ability in Agent-Oriented Programming is message passing, that is that agents have the ability to talk to other agents. JADE accomplishes this through a protocol established by the Foundation for Intelligent Physical Agents (FIPA) for passing messages and using pre-built protocols.[3] These protocols are then treated just like a *Behaviour* from the perspective of the agent.[4]

Related Work

We found that other researchers have approached this problem using other frameworks, some not explicitly agent based, to develop their AI. A group of AI writers (**?**) used the BWSAL to implement a simple strategy with an agent minded outset.[5] We feel that, although they had modeled a sound agent system, they had a very limited test scope.

A management design similar to ours (detailed later) built an agent framework for use in Open Source RTS (**?**). They implement their agent system using ABL in Wargus and have encouraging results. They, however, only test their agent system against two scripted strategies and focus on different tasks than our own system.

Another implementation using ABL (**?**) focused on reasoning to reactively change strategies. They had positive results against the built-in AI, but less than positive against human players. While they have a strong reasoning system, our system will be focused more on creating a modular framework.

Creators of the Overmind AI at UCBerkeley created a successful StarCraft AI that uses reinforcement learning to individually control attack units with great precision.[6] At the time of this writing, we have not found any papers discussing the use of JADE in creating an AI for StarCraft: BroodWar. More so, we have not found any using JADE for the development of AI frameworks in other video games. This presents an opportunity for more future research.

[3] `http://www.fipa.org`

[4] The API for JADE can be found here `http://jade.tilab.com/doc/api/index.html`

[5] `http://code.google.com/p/bwsal`

[6] `http://overmind.cs.berkeley.edu/`

Design

Motivation

In order to accomplish the task of creating an AI for StarCraft, we decided to approach it by having each important task in the game represented by an agent concerned with solving that task. While an intuitive approach would be to represent each unit and structure as an agent, we quickly abandoned this due to many issues that design would entail such as coordination and cooperation of the units, hardware allocation of that many threads at once, and actually being able to effectively communicate with BWAPI.

Instead, we conceptualized what the problem sets would potentially be and we came up with the following tasks: path-finding, resource management, building placement, controlling military units, and managing upgrades. On top of these tasks, we also knew a few more agents would be needed to maintain and combine these problem sets to the goal of the current game. Based on the idea that agents are focused on one or two tasks that lead to our current organization.

Organization

Initially, our first design was based on the RTS AI framework, BWSAL. However, this is developed in C++ and is not Agent-Based. The idea behind using an Agent-Oriented design was to offload reasoning of playing the RTS to individual agents that only focused on one facet of the overall problem. We then split the problem into the following 7 agents (see Fig.1):

- *Commander Agent* : Concerned with keeping the information up to date and controlling current active build orders. The Commander Agent is the heart of KhasBot, it is also the communication gateway to and from StarCraft. All game information passes through the Commander Agent and the Commander Agent processes all the commands to pass back to StarCraft.
- *Building Manager Agent* : Concerned with placement of future buildings and builds structures based on the issued build order. The building manager will analyze the current map data to determine where buildings can be placed and can communicate with the Unit Manager Agent to obtain a worker to build the planned building.
- *Structure Manager Agent* : Concerned with training units and upgrading technology. Any command that would be normally issued by a player to a building is managed by this agent.
- *Battle Manager Agent* : Concerned with all combat matters including scouting, the act of sending a unit out into the fog-of-war to obtain valuable information. The Battle Manager Agent controls all attack units, battle tactics, and makes decisions with those units based on scouting information, enemy movements and build orders.

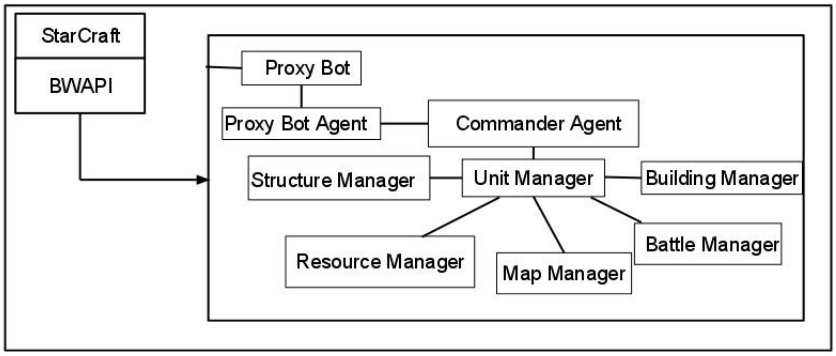

Figure 1: Overall System Design for *KhasBot*

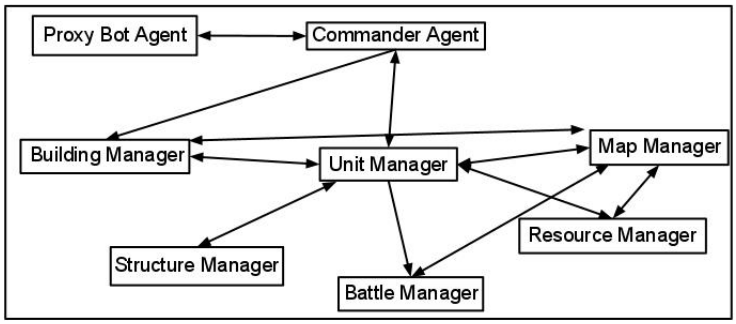

Figure 2: Various Protocol Paths used in *Khasbot*

- *Resource Manager Agent* : Concerned with all resource gathering, including rate of gathering, worker management and needed materials for segments of the build order.
- *Map Manager Agent* : Concerned with maintaining and updating map information and determines efficient path finding for any units in AI control.
- *Unit Manager Agent* : Concerned with maintaining unity between all the units used by the other Manager Agents and merging together unit commands for the Commander Agent.

In addition to the above agents, we also developed the *Proxy Bot Agent*. This agent is in charge of interfacing with the BWAPI and getting the socket information from the StarCraft game and converting that information into messages the agent system could understand. It also converts messages from the agent system to something the BWAPI and socket information could understand.

Most of our manager agents follow a similar behavior to each other. Due to this, we were able to create a superclass *KhasBotAgent* that took care of most of the repetitive tasks. With many agents working on separate areas, the key here is organizing their communication with each other. To aide this, all of our protocols exhibit a request/inform type flow, which is a core design structure of the JADE Framework. Game updates are sent to the *Unit Manager Agent* by the *Commander Agent* and all other managing agents are updated by the *Unit Manager Agent* when relevant information changes to their goal.

The actual StarCraft game is mapped to a *GameObject* that the agents use to keep track of what they are doing. As the game progresses, the Multi Agent System (MAS) continues to get game updates via a *GameUpdateObject*, which is a reduced game object with core information our agents need. The *GameObject* is broken into player information and Map Information. Map Information includes items like where is it safe to build or walk and what the starting locations are for the map in the game. Player information handles all units and ways the units can do research or upgrades. These units include structures, mineral patches, and all other pieces in the game that can change.

Implementation

The basic idea that each manager agent follows is they are each an agent inside of JADE that have multiple threaded behaviours focusing on different protocols between the various Manager Agents.

Due to the possibility multiple behaviours at once, each agent uses the DataStore ability from JADE, a central data location on the platform, to keep its information consistent with the other behaviours. The DataStore is also used as a means of passing information

from one behaviour to another. Any information that is shared among one or more behaviours is placed in the DataStore.

The main implementation of our manager agents can be found in our abstract class *KhasBot Agent*. All other manager agents take this as a superclass. *KhasBot* works on the following idea: all information for that agent is stored in its own *DataStore*, which is setup to be used by all behaviours of the agent. Since each agent has different behaviours, this is something that has to be overridden in their own *setup()* method. All behaviours for the agents are *ThreadedBehaviours* due to the fact that most of our behaviours are protocols between each managing agent and each protocol should not wait for a different protocol to finish, we found this to be the easiest way to implement non-preemption in JADE.

The communication or protocols of the manager agents falls into two categories, *INFORM*s or *REQUEST*s. We will discuss how *Resource Manager Agent* is set up as an example: ResourceManagerAgent has up to 7 behaviours. 3 work as one-shot requests asking for either workers, minerals, or gas. The other 4 are longer lasting protocols that wait on updates of the game, respond to requests of worker needs or supply queries, and one last one to do the agents 'default' behaviour which is to assign workers to gather resources. *INFORM* messages are used to send information from one agent to another. *REQUEST* messages follow the FIPA-Request message framework, where one agent make a request to another agent for resources or information.

An important aspect of our framework comes from the protocols between the agents and who can talk to who. Figure 2 shows the links between all the managing agents.

Build Orders

Build Orders are a part of StarCraft strategy that aim to follow a highly optimized, pre-set list of commands for the early part of a game. These are used to provide a foundation to achieve victory. One of the goals of our AI is to select or adjust the build-order on demand, based on map and scouting information. In order to aid in lowering the complexity of our agent framework, we decided to implement an outside structure to maintain the list of commands the agents needed to complete. This would allow us to apply different implementations and observe their effects on our AI. The solution we choose to implement this was an external XML file. This created an easy to manage medium that allows us to test different ideas within the structure without having to rebuild our framework.

The XML structure is based on the race selected and the in-game situation. An example is given below. The strategy subsets are based on the supply resource and process the different commands held as sub-items when the required supply count is met. The sub-item(s) can be `unit`, `structure`, or `action`. The `unit` item has attributes for the unit name. The `structure` item has attributes for the structure name and location of placement, which could be either a map location or near another structure. The `action` item has attributes for the action name, by name of the unit, location to achieve the action and importance. For more advanced control in different in-game situations, it can maintain a list of commands for different opposing races and for each playable race inside the `versus` tag.

```
<?xml>
<build>
  <race name="Protoss">
    <versus race="default">
      <step supply="7">
        <structure name="Pylon" />
      </step>
      <step supply="8">
        <structure name="Gateway" />
      </step>
    </versus>
  </race>
</build>
```

The XML structure, like the one seen above, is loaded into an Java Object named *BuildLoader* that will contain the list of orders. As the game progresses, the Commander Agent will send the current game information to *BuildLoader*. It will look at the current in-game supply count and read specified list of commands for that condition. Updates for these commands are required before it will continue through the list. An update will run through the list of commands up to the last supply count which it returned results to make sure everything is still true. These checks are made as the battle situation could change aspects of our build order. Special location constants will be asked from the Unit Manager Agent which will allow *BuildLoader* to check the location of a new structure that will be created in-game and determine if they are close to where a certain structure needs to be built. Unit item commands will notify the Commander Agent to tell the Unit Manager Agent that specified unit needs to be created. Action commands will work the same way by providing an action for a named unit type, for example, scout action to a probe or attack to a group of battle units.

Results

At this stage in development, initial tests have been positive. The message passing between agents have resulted in simple 1 to 2 line build orders being executed. Buildings are placed next to labeled pylons, as planned, and units are training according to build order. As we are more focused on getting the framework set up, we have not fully tested our system against real game scenarios against the StarCraft default AI or a human player. Our future work will continue to expand our build order and unit building capability.

Conclusion and Future Work

We have found that building StarCraft AI is viable using an Agent-Oriented Programming Framework. The communication structure established by JADE allowed us to create agents capable of specific goals and relaying information based on those goals to other agents and to StarCraft. The real achievement of this framework, we believe, will be the ability and adaptability to change out reasoning for various parts of the game, say a more efficient path-finding algorithm, without having to touch or re-debug other parts of the game.

We believe that splitting the computation needed in an AI among agents provides another avenue for creating StarCraft AI and ultimately video-game, military, or management AI.

In the future, we will continue to refine KhasBot with more build orders and potential decisions for those build orders. We will also look into generalizing KhasBot for all three races, as our Build Order parser will allow for that.