NTNU – Trondheim
Norwegian University of
Science and Technology

Jantu

A Cognitive Agent Playing StarCraft: Brood War

Martin Tobias Holmedahl Sandsmark
Ken Børge Melhus Viktil

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Helge Langseth, IDI
Co-supervisor: Anders Kofod-Petersen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

It has been shown that most players enjoys playing video games against other human players instead of computer controlled opponents. But most research on artificial intelligence in gaming today focus on just winning in the most effective way possible, instead of making the agents more human-like.

Cognitive architectures are designed to emulate how the human brain operates when performing tasks. Very little research has been done on applying cognitive methods to the field of real-time strategy games.

In this paper we aim to research the use of a cognitive model that is capable of playing StarCraft: Brood War with decent results. The result is an AI agent implemented with a cognitive framework called LIDA. The resulting agent is only a proof of concept implementation, but we provide suggestions for how it can be improved in the future, and where the problems and limitations of our approach lies.

Sammendrag

Det har vist seg at folk som spiller dataspill foretrekker å spille mot andre mennesker, i motsetning til å spille mot kunstige spillere, men størstedelen av forskningen på kunstig intelligens i spill i dag fokuserer mer på å vinne mest mulig effektivt, istedenfor å se på hvordan man kan få de til å spille mer menneskelig.

Kognitive arkitekturer er laget for å emulere hvordan den menneskelige hjernen utfører oppgaver. Det er per dags dato nesten ikke noe forskning på bruk av kognitive arkitekturer innen sanntidsstrategispill.

I denne rapporten vil vi undersøke bruken av en kognitiv modell for å spille StarCraft: Brood War. Resultatet er en agent implementert med programvarerammeverket LIDA. Agenten er en prototypeimplementasjon, men vi kommer også med forslag til hvordan den kan forbedres i fremtiden, og hvilke problemer og begrensninger tilnærmingen vår har.

Acknowledgements

Firstly we would like to thank our supervisors; Helge Langseth and Anders Kofod-Petersen. We would also like to thank the Cognitive Computing Research Group at the University of Memphis, for their LIDA framework which is the basis of our project. Finally we would like to thank the developers of the BWAPI project, and in particular Adam Heinermann, as well as the rest of the members in the #BWAPI IRC channel on Freenode.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we introduce our project. Section 1.1 presents the background and our motivation for the project and describes what we will do in this project and our main goals. Section 1.3 contains an explanation of the name for the agent. Section 1.4 layouts the structure of this report.

1.1 Background and Motivation

1.1.1 The Problem

Real-time strategy(RTS) games have been a complex problem to solve for artificial intelligence(AI) researchers for several years now. Ever since Michael Buro published an article in 2003[9], where he encouraged researchers to focus more of their expertise on the RTS domain, a lot of new methods have been applied to these problems with varying degrees of success. Some have been very successful and are now able to beat top level human players in real-time matches.[12]

After Burro's paper several research platforms emerged, including OpenRTS[10] and Wargus[35]. Wargus is an open source WarCraft 2 clone, which uses the Lua scripting language for its agent creation, while OpenRTS is a platform originally created for AI research. These platforms have some limitations, because they are mostly developed by hobbyists and researchers and not professional game development companies, they lack both polish and wide-spread usage outside of academic environments. They also lack a lot in the way of individual unit management, path finding and generally the

possibilities for complex strategies and tactics, because of the very simple game mechanics.

As an answer to this the Brood War Application programming interface (BWAPI) was created. It supplies a comprehensive API for interacting with the popular RTS-game StarCraft: Brood War. The API lets developers easily create agents that play the game and can be run in tournaments against each other as well as against real players. It has gained a lot of popularity during the last years.[7] There are several competitions held each year, where programmers pit their BWAPI-based agents against each other, both for money and fame.[32] But these artificial agents still have some way to go before they can match human players.[36]

While just winning the game is an interesting and worthy goal, it is according to Arrabales et al much more engaging for human players to play with other humans than with artificial agents.[2] We therefore think that developing human-like intelligence for playing computer games is something that should have a greater focus then it has had so far. Cognitive architectures are based around trying to emulate how the human consciousness works and acts in certain situations. There has been done very little research so far into using cognitive agents for playing RTS games, but they have been utilized with great success in first person shooter games.[2]

1.2 The Project

In this project we will implement a proof of concept cognitive agent that plays StarCraft: Brood War. We will be using a cognitive framework called LIDA in our implementation. The first step in the project will be to integrate the LIDA framework with StarCraft and BWAPI, so the LIDA framework can control the StarCraft process. We also need to create a bridge between the in-game time references called frames, and the frameworks time references called ticks. After accomplishing this we should be able to control the execution of StarCraft from our framework, so we can pause, resume and step frame by frame and tick by tick for debugging purposes.

The next step will be to create a proof of concept implementation of a cognitive bot that can play the game, with decent results. This will allow us to investigate what kind of challenges present themselves when one implements an agent like this, and where there is room for improvement, and what the future might look like.

1.3 The Name

Figure 1.1: The name of our agent written in Hindi.

The name *Jantu* is hindi, and is translated as "Pertaining to the merely sentient part of a creature, as distinguished from the intellectual, rational, or spiritual part; as, the animal passions or appetites."[22], which reflects what we are trying to achieve with our work in this project.

1.4 Report Structure

This report is structured into five main chapters:

- Chapter 1: **Introduction**
  This chapter describes the background and motivation, as well as what we hope to accomplish during the project. It also describes the general structure of the report.
- Chapter 2: **Theory**
  In this chapter we presents the relevant theory for understanding the tools and technologies used during the project. We presents some general information about StarCraft as a game, we also present some theory on the cognitive architectures in general, and more detailed about the model and framework we will be using.
- Chapter 3: **Implementation**
  In this chapter we present how we integrated the framework with StarCraft and what features we implemented for our agent.
- Chapter 4: **Results**
  In this chapter we present the results of running the agent, how well

it performed and some insight into how the bot ended up interpreting and acting in the given environment.

- Chapter 5: **Evaluation**
  Here we summarize and evaluate results and the project. We discuss what the agent did good and why it had problems in the situations it did. We also outline how we think the agent can be improved in the future.

Chapter 2

Theory

In this chapter we will present background theory that is important for our project. In Section 2.1, we have a general explanation of the game, and also explain the BWAPI architecture and how it utilizes shared memory and JNI to allow bots written in Java. Section 2.2 describes models of cognition, the theory behind them and some architectures/implementations. Section 2.3 describes the LIDA model of cognition as well as the LIDA framework that implements this model.

2.1 StarCraft

StarCraft is a multi-player, real-time strategy(RTS) game, with a heavy focus on both economic strategies, as well as effective management of individual in-game units. Brood War is the expansion pack for the original StarCraft game that introduced new units, maps and upgrades. The game has been praised for its emergent complexity even though the mechanics are simple to understand. StarCraft: Brood War has been played on a high competetive level for over 10 years, and it has been evolving all that time with new strategies and tactics.

2.1.1 Races

The game has three races that a player selects between. The insect-like Zerg, the advanced alien race Protoss and the human Terrans. Each race has its own units and buildings as well as different game mechanics.

Figure 2.1: StarCraft: Brood War

Terran

Figure 2.2: The in-game logo of the Terran race.[33]

The terran are human colonists originally from earth. This is the most balanced race, which relies on a mixture of large numbers and powerful units. They have great mobility in their biological armies, and great defense and slow turtling in their mechanical units like tanks.

The Terran worker is a Space Construction Vehicle (SCV), and unique for Terran is that it can repair any mechanical unit or building if they are damaged. Another unique mechanic with Terran is that their buildings can lift off and fly around to reposition themselves.

Several of the buildings can also be upgraded with addons that are smaller

buildings that are constructed and connected to the main building. Addons unlock new upgrades and units to be constructed from the building it is connected to. Utilizing the lift off mechanic, buildings can swap addons after they are constructed. This increases the build order diversity, as addons can be switched between production buildings.

All terran units can be healed after they take damage; the mechanical units and buildings can be repaired using an SCV (but doing so means that the worker will not be gathering resources for that time), and to heal damaged biological units you have to train special units called Medics.

Zerg

Figure 2.3: The in-game logo of the Zerg race.[38]

The zerg is an insect-like collection of different biological races assimilated under a central intelligence. They rely on biological abilities that are selectively evolved instead of relying on technology like the other two races.

The individual units in the Zerg army are usually not very powerful, but their strength lies in superior numbers as well as the ability to quickly reinforce the army with new units when some die.

The Zerg worker unit is called a drone. What separates this worker from the other races is that in order to create a new building the worker has to "sacrifice" itself to morph into the new building, the worker becomes the building.

Zerg has a unique game mechanic for creating new units. The main building, called the hatchery, creates larva on regular intervals, up to a maximum of three, which can be morphed into new units. Both workers and army units are created from the larva, and because of the limited availability on these a player has to prioritize at all times whether he wants to create workers or army units, balancing military strength against economic advantage.

To increase the production capacity they have to either expand to a new base, or create extra support hatcheries in their current bases to gain access to more larva.

Zerg can only build new buildings on creep, a biological substance that covers the ground and spreads from existing buildings as well as from Creep Colonies, a special building that is used only for extending creep coverage on the map.

Zerg units have a unique ability to regenerate health after they are damaged as long as they are not in combat. This makes retreating and regrouping a good tactic for Zerg players as their army will have time to regenerate back to full health after a defeat on the battleground. Together with another unique ability some of the Zerg units have called burrow, which allows them to burrow underground and hide or escape, it can be difficult to kill units since they can escape and regenerate the lost health.

Protoss

Figure 2.4: The in-game logo of the Protoss race.[27]

The protoss is a highly advanced race with powerful mental abilities.

They are both technologically and military advanced, and usually rely on few, but very powerful individual units. They have very expensive units that can crush a much bigger army by themselves.

The protoss worker is called a probe. Probes don't need to construct the buildings, they only tag an area and then the building gets warped in from the protoss home world. This means a single probe can start the construction of several buildings at the same time and then return to gathering resources.

Similar to Zerg, Protoss can't simply construct buildings anywhere, they have to be constructed on a power grid that is generated by pylons, the Protoss supply building.

A unique Protoss feature is that all the units and buildings have shields that protect them from damage and regenerates over time. For an enemy to damage the unit it first has to deplete the shield, then it can do damage to the health of the unit. But damage taken after the shield is down cannot be healed or repaired.

2.1.2 Gameplay

Micro vs. macro

Two well-known concepts in the RTS communities, and the StarCraft community in particular, are micro management and macro management. Macro management refers to large-scale economic and strategic decisions, while micro management refers to smaller-scale control of individual units, or groups of units.

Good micro means being able to group units together effectively, e.g. to minimize "splash" damage taken (splash damage is when attacks affect a larger area than a single unit, so that if the units stand too close together all of them take some damage when one of them is hit). Another example is being able to "target fire" (attacking a single unit), for example to quickly be able to take down already damaged enemy units.

Good macro is essential to be able to take advantage of the good micro. Without good macro the opponent can simply flood you with units, without much regards for micro.

So while exceptional good micro can balance out a lacking control of the macro management and vice versa, a good control of both is needed for a successful agent.

Supply

Supply is a term for a limit imposed by the game on how many units a player is allowed to posses at any given time. To increase the supply, a player can build a special kind of unit or structure; for the Protoss race this is *pylons*,

for Zerg it is *overlords* and for Terran it is *supply depots*. The name stems from the terran unit needed, supply depots.

A player is not able to build more units unless supply for them is available. And because making supply costs money itself, when and how much supply to build is an interesting decision making problem.

Fog of war and scouting

Fog of war is a well-known term from RTS games, which refers to unobserved parts of the environment or map. In StarCraft this is shown as fog that covers the map where you have no units. This leads to partial observability, and gives rise to uncertainty about the rest of the map, about what the other player is doing and what resources he has exhausted. To counteract this, it is common to *scout*, that is, to send out units to simply observe the other player, and see where he for example expands to or how big his military presence is, and when he moves out to attack.

2.1.3 BWAPI

To ease the development of third-party agents that play the game, an application programming interface (API) has been developed for StarCraft: Brood War, by third-party developers. It works by injecting itself into the process the game runs in, and hooking into various functions used by the game, as well as reading and writing various memory areas directly.[7]

There has grown a sizable community around this effort, and there are several tournaments where programmers can participate with their own agents.[7][32]

There are also several third-party addons and extra libraries for easing the development of agents, such as the Brood War Terrain Analyzer (BWTA), which provides easy-to-use functions for analyzing the maps for finding chokepoints and suitable locations for various buildings[8], and the Brood War Standard Addon Library (BWSAL) which is both a generic, modular framework for BWAPI agents as well as default implementations for a large part of the modules needed for implementing such an agent.[11]

BWAPI only provides a C++ API, so for using it from other languages various types of bindings are needed.[34]

There are two modes for loading agents using BWAPI; loading it directly into the StarCraft process, or using a shared-memory area to communicate state

between the agent process and the StarCraft process in which the BWAPI code is running.[7]

JNIBWAPI

When using Java for developing an agent the most well-supported approach is to use the JNIBWAPI project, which uses JNI to provide Java-bindings for BWAPI. It utilizes the shared-memory approach of BWAPI to avoid having to load the Java virtual machine into the StarCraft memory.

2.2 Cognitive Architectures

Cognitive architectures are architectures that base themselves on some model of human cognition. There are several competing theories about how cognition works, and one of the most well-supported is the Global Workspace Theory.

There hasn't been widespread research done with cognitive architectures. Arrabales et al put forth that this is perhaps because of poor understanding of cognitive research amongst researchers in the field of classical AI. [2]

There are many different cognitive architectures available today[14]. We chose to focus on those based on the Global Workspace theory, for several reasons. It's widely accepted as the currently best model of understanding consciousness[18], implementations using it generally provide believable results[2][30] and it has a solid grounding in neurological and psychological models, experiments and empirical evidence[18].

2.2.1 Global Workspace Theory

Global Workspace Theory is a model of conciousness that is well supported by experimental data, and is one of the most widely accepted models. [13] It has been used to implement processes that imitate human decision making (for example for solving the problem of assigning people to jobs in the US Navy).[3][16]

It is based around an understanding of the brain as a set of many independently processing modules, working together by utilizing a shared workspace (hence the "global workspace"). Every "cognitive cycle" all the processes

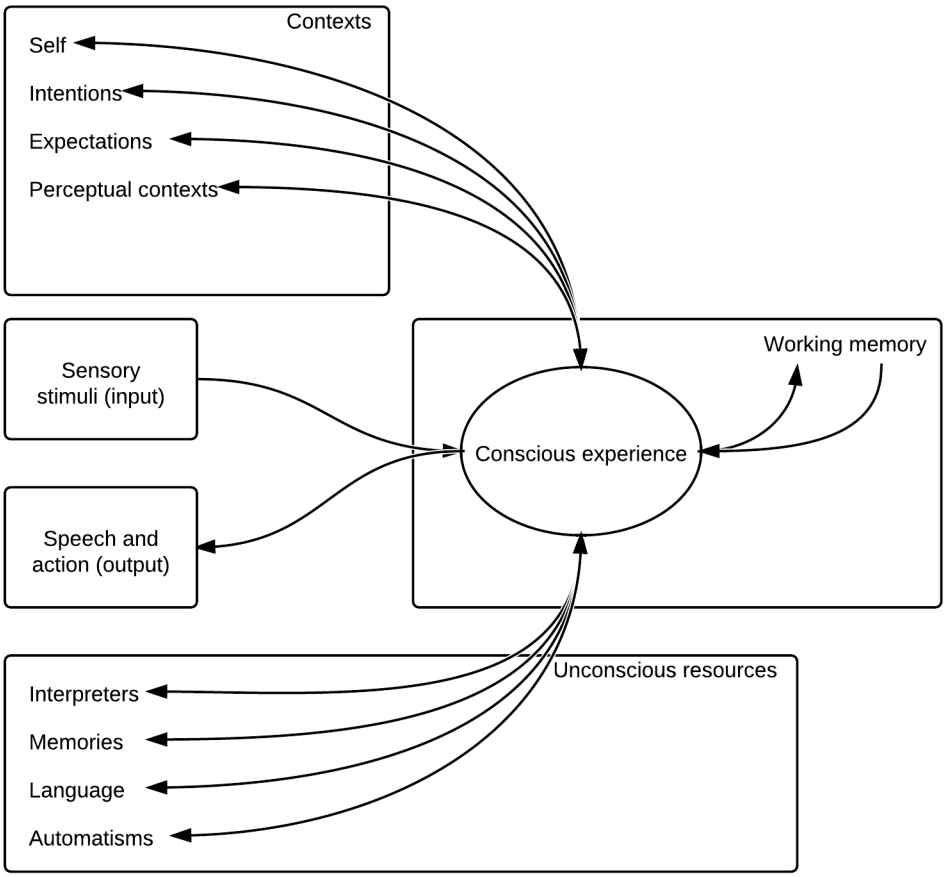

Figure 2.5: A schematic of the Global Workspace theory[3]

compete for attention, and a single one gets the spotlight shone upon it.[3] When this happens, all the other modules/processes receive the "broadcasted" data from that module, and use it as they see fit; for example the temporal lobe stores the information received as memories. This solves the relevancy problem[1], and it lets parts of the brain collaborate on problems that can't be solved unconsciously by individual modules.

See figure 2.5 for a graphical overview of the theory.

[1]The relevancy problem is part of the frame problem, the problem of knowing what to apply where in problem solving, and was considered to be intractable.[29]

The theatre metaphor

A common metaphor used is the *theatre metaphor*. Here we view the mind as a figurativly theatre, with an audience and a stage with a spotlight shining on it.

Then various actors on stage get the spotlight shone upon them (according to the pandemonium theory which is often used; the one who shouts the loudest[28]), and is allowed to share its message with the the audience.

Behind the stage, are the various executive processes (represented by script writers, directors, etc.) who aren't visible but still help form what is seen.

Neurological basis for the Global Workspace Theory

According to R. Llinás et al.[25], the looped neural pathways between the thalamus and the cortex might be what is responsible for the conscious collaboration between various parts of the brain. The thalamus is responsible for letting various parts of the cortex broadcast and influence the rest of the cortex. This meshes well with the global workspace theory, and the idea of a single subsystem broadcasting its contents to the rest of the whole.

There has also been discovered a link between the switching between coherent and decohoerent EEG-activity that seems to indicate a switching between states of competition for access to the global workspace. Decoherent electric activity seems to indicate a competitive process, while coherent activity indicates a passive "listening".[21]

The periods between these states is compatible with the widely accepted figures for the time it takes for stimulus to become conscious.[29]

Several studies support the theory that consciousness is what enables global activation. They usually compare conscious and unconscious conditions in conscious subjects, either by sensory stimulation or overpractice of automatic habits[2]. All results show that conscious processes lead to widespread cortical activation, while unconscious ones usually only activates local regions.[4]

There has also been experiments done where subjects have been in various states of unconsciousness (sleep, general anasthesia, epileptic loss of consciousness and vegetative states), and it has been found that sensory stimulation in all states only lead to local activation in the cortex, which seems to

[2]Consciously doing tasks that usually are done unconsciously.

indicate that conscousness leads to spreading activation, and collaboration between parts of the cortex.[29]

Implementations

There has been implementations of several architectures incorporating the global workspace theory. Here follows some of the most significant ones.

IDA In the mid-nineties an artificial agent was designed for the U. S. Navy that would replace so-called "detailers", which are specialists that allocate personell. They take into account personal preferences, moving costs, requirements from the Navy (number of personell on each boat etc.) and various regulations from the Navy. Preferences from sailors are taken in from natural language e-mails, and these are processed using a slipnet[3] that processes the natural language into something that has usable semantic meaning for the system. The other inputs, regulations and requirements from the Navy, are stored in databases, and are pre-processed by similar slipnets before being pushed into the system. This system was able to replace the 300 or so detailers that the Navy employs. The system is up and running, and is matching the performance of the human detailers. [5][15]

It was based on the Virtual Mattie agent, which is a virtual clerical agent, responsible for interacting with humans through natural language over e-mail.[20]

LIDA One problem with IDA was that learning wasn't really implemented at all, and therefore they started re-implementing from scratch, in a project called LIDA; the Learning IDA system. The idea was to add learning from experience; learning newly perceived objects and their relationships to already known objects, relationships between objects, categories, relationships between objects and actions, effects of actions, and improved perception/-tagging of sensory data with learned memories.[17] Over time, however, the LIDA framework evolved to become a more generic Java-framework for cognitive architectures.[30]

CERA-CRANIUM This is a two-fold architecture, as reflected in the name. *CRANIUM* is a more generic tool for creation of a high amount

[3] A slipnet is a semantic network with activation.

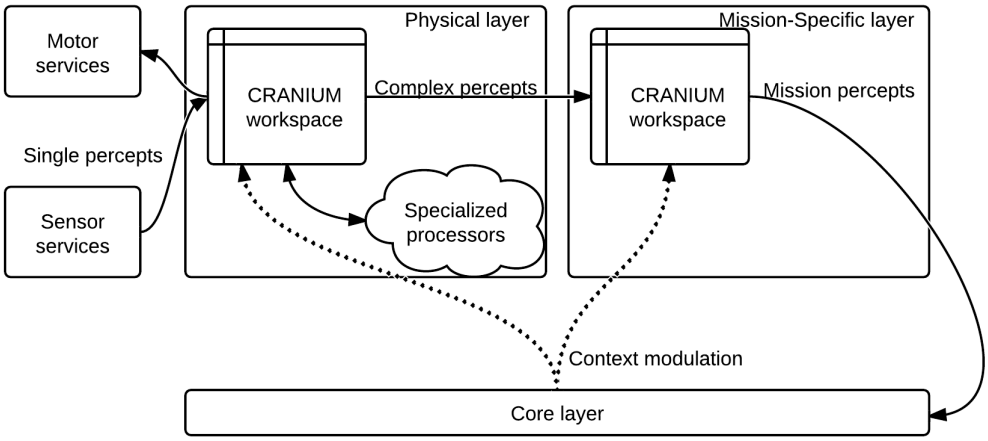

Figure 2.6: Overview of the CERA-CRANIUM architecture.

of parallel processes operating with shared workspaces. *CERA* uses these services for creating a dynamic and adaptable system which operates on perceptions and generates actions, based on a cognitive model.[2] It has been used for making agents that act in several different environments, both virtual environments like the computer game Unreal Tournament, as well as real environments, where it has been embodied in small robots that map out unknown environments. [1] A high-level overview of the architecture can be seen in Figure 2.6.

2.2.2 Cognitive Models in Game AIs

There have been several implementations of models of cognition as game-playing agents. Examples of this is CERA-CRANIUM, which was used to implement an agent playing Unreal Tournament [1], and SORTS which implemented an agent for the real-time strategy game ORTS using the symbolic, cognitive architecture Soar.[37] One of the reasons for using computer games for experiments with regards to high-level artificial intelligence is that the characteristics of computer games lend themselves to this, by eliminating noise and uncertainty, and providing a more or less realistic simulated environment.

2.3 LIDA

LIDA is both the name of a cognitive model and a software framework based on the cognitive model. Both are developed at the Cognitive Computing Research Group[4] at the University of Memphis. The LIDA framework is currently in beta, but it is based on the well-tested IDA project.

2.3.1 The LIDA model

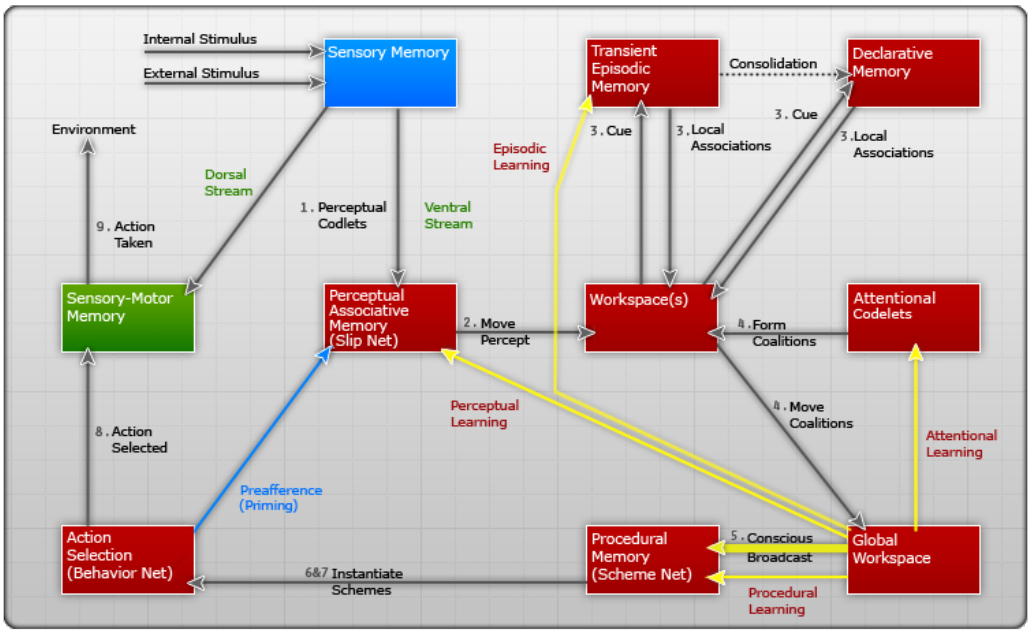

Figure 2.7: Overview of the LIDA cycle.[19]

The LIDA model is a conceptual and computational model that describes cognition. It is based on the General Workspace Theory, which is a theory to describe functional consciousness. It incorporates and accounts for a large number of psychological and neuropsychological theories.[18]

The LIDA cognitive cycle

The computational model of LIDA views the artificial agent's "life" as a series of cognitive cycles. A high-level, simplified view of how this cognitive

[4] `http://ccrg.cs.memphis.edu/`

cycle is modeled in LIDA can be seen in figure 2.7. The cognitive cycle itself is subdivided into three separate phases; the understanding phase, the consciousness phase and the action selection phase.

Understanding phase In the understanding phase sensory input is tagged with semantic meaning. First low-level feature detectors in the Sensory Memory are run on the incoming stimuli. The output from these is passed onto the Perceptual Associative Memory. There higher-level feature detectors are run, which recognise abstracted entities like objects, events, actions, etc. The output of these is sent to the Workspace, from where it is pushed to both the Transient Episodic Memory and the Declarative Memory. Local associations generated from these memory modules are combined with the percept to generate a Current Situational Model, which is the agent's "inner map" of how the world looks.

Attention phase Parts of the Current Situational Model is composited together to coalitions by Attention Codelets. These coalitions are put into the Global Workspace. From the Global Workspace a codelet selects the coalition in most need of conscious attention and broadcasts this.

Action selection phase The Action Selection module takes in possible action schemes (behaviors) from the Procedural Memory, and selects one of these according to a competetive process. The selected action scheme is sent to the Sensory-Motor Memory, where it is executed. This method of action selection is based on Maes' behavior net.[26] Learning is also done in the action selection phase. In the Perceptual Associative Memory new entities and associations are created and old ones are reinforced during this phase. In the Transient Episodic Memory events from the broadcast from the Global Workspace are stored. The Procedural Memory stores new possible action schemes from the broadcast, and reinforces old ones.

The workspace

The workspace is composed of three main modules; the Current Situational Model, Scratchpad and the Conscious Contents Queue.

Current Situational Model This is where the internal "map" of the agent is represented. This contains current events that are linked to associations. Various structure-building codelets running in the Workspace are responsible for building these.

Scratchpad The Scratchpad is used for temporary storage for the codelets while they build up structures before moving into Current Situational Model.

Conscious Contents Queue This contains a list of previous broadcasts, which allows LIDA to take time into account, and reason on, things that happens over time.

2.3.2 LIDA framework

Background

The LIDA framework implements a growing subset of the full computational LIDA model in a re-usable Java-based software framework.

While LIDA seems to have started simply as an extension for the IDA agent with added learning modes[19], it has evolved into a generic AGI framework, which can be used for other models than LIDA, because of the modular nature of the framework.[30]

The reason for using a framework is that software projects implementing cognitive models tend to grow fairly large and complex therefore difficult to both implement in the first place and maintain over time.

Various types of frameworks, like the Qt application framework or Ruby on Rails web application framework, have been successfully used in software development for a long time to simplify and ease the implementation software projects[6].

They do this by providing pre-built functionality, a shared platform that can be improved and worked on by everyone who uses it, and also promotes collaboration between users of the framework. And since the amount of custom code that has to be maintained in the individual projects decrease significantly the burden of maintainence eases as well.[31]

Features

The LIDA software framework contains default implementations of the major modules in the LIDA model as well as abstract classes for the generic parts of the model, which needs to be implemented in an agent.

The main parts of it are the modules, tasks, common internal representations, a task manager, a GUI, an object factory, strategies and also an XML parser for the agent description files.[31]

Modules

The modules are collections of tightly coupled data and processes that operate on them. The coupling between the various modules usually is very loose, however.

Domain independent modules in the LIDA model have full implementations in the framework, while domain-dependent ones only have abstract implementations that has to be provided by developers that use the framework.

Modules are arranged in a hierarchy; for example the Workspace module has a submodule that represents the Current Situational Model.

Communication between modules is done with the Observer design pattern, where objects registers interest in other objects by calling their *addListener* methods, and then get callbacks to pre-defined methods whenever there are updates. An example of this is the GlobalWorkspace-class, which implementations of BroadcastListener can listen to.

The modules with default implementations in the current version of the LIDA framework are as follows:[31]

- Environment (abstract)
- Sensory Memory (abstract)
- Perceptual Associative Memory
- Transient Episodic Memory
- Declarative Memory
- Workspace
- Structure-Building Codelets

- Attention Codelets
- Global Workspace
- Procedural Memory
- Action Selection
- Sensory-Motor Memory

Environment This is an abstract class, since it is highly domain-dependent, and it has to be provided by the user of the framework. The implementations need to implement at least getState and processAction.

Sensory Memory This is also highly domain-dependent, and therefore has no default implementation.

Perceptual Associative Memory Here all the nodes available from the environment are, together with their current activation level.

Transient Episodic Memory This is the short-term episodic memory. This module stores events from the conscious broadcast for a short term, with a quick decay.

Declarative Memory This is the long-term episodic memory, with a slow decay. Entries in the Transient Episodic Memory are moved here if they haven't decayed for a long period of time.

Both the declarative and transient episodic memory are implemented as sparse distributed memory.[24]

Procedural Memory This module contains behavior schemes that are used to instantiate behaviors. Instantiated behaviors from this are sent to the action selection from procedural memory.

Action Selection In the current version of the framework this is implemented as a simple rule-based system, however in version 1.2 this is replaced with a behavioral net, inspired by Maes' nets.[26]

Tasks

These are small processes implemented as short-lived algorithms that reflect the *codelets* and other processes in the LIDA model. The implementation class is called *FrameworkTask*.

Tasks are spawned by a separate class, the TaskSpawner, which classes that need to spawn tasks interact with. Tasks are scheduled for execution at particular "ticks", which are the time units used in the framework. The task manager runs the Task's *runThisFrameworkTask* method each time it is scheduled for execution. Tasks can either be run once, or repeatedly, depending on the type of task it is. For example an ExciteTask, which is used for passing activation, is run once, while feature detector tasks are run repeatedly on input.

When a task has finished executing it is returned to the TaskSpawner which processes the results and decides whether the task should be rescheduled for execution.

Common internal representations

To make modules able to efficiently collaborate on information and data, there are several common internal representations used for data in the framework.

Nodes and *Links* are the two major data structures used for representing information in the LIDA framework.

Nodes are data structures that can represent anything needed in an agent, both concretely or more abstract, like features, objects, events, concepts, feelings, actions, etc. Every Node has a reference to a PamNode from the Perceptual Associative Memory, from which it originates, as well as a unique ID.

Links represents connections between Nodes. The framework differentiates between simple links, which links between two Nodes, and complex links, which links to other links. Links have a LinkCategory that describes the nature of the Link; for example a Link connecting two Nodes "ball" and "red" might have the category "feature", for a representation of a red ball.

Both Links and Nodes implement the *Linkable* interface, and contains a unique ID represented by a variable named *ExtendedId*.

It is also possible to have custom subclasses of both Link and Node with custom properties and functionality.

All instances of Link and Node, both default and custom subclasses, are created by the element factory.

The common internal representation used for sending information between modules is usually a NodeStructure, a graph structure composed of Nodes and Links. The default NodeStructureImpl has a default implementation of methods for managing the elements in it (adding, removing, retrieving, merging, etc.). An important point to note is that when an element is added, a copy is made and added instead, because adding the same Java object to several node structures could have significant, unforeseen circumstances (where operating on data in one structure affects a completely unrelated one).

Task manager

The task manager is responsible for the dispatching (when called from the TaskSpawner) and execution of the tasks, usually parallelized. This is similar to the CRANIUM software in the CERA-CRANIUM architecture described in Section 2.2. It is also responsible for keeping track of the internal time representation, the "ticks".

Tasks are organized in a queue, sorted according to the tick they are scheduled for. The task manager main loop consists of four steps; First it decays the activation level of all the modules, then it executes all the tasks scheduled for the current tick, thirdly it updates the GUI and then increments the tick. This loop is controlled by the public start/pause methods, which are also exposed in the toolbars in the GUI.

The task manager also has a *tick duration*, which is the minimum amount of time a tick takes. This can be used to be able to slow down execution if one wants to inspect what goes on. This only is used if the minimum tick duration is longer than the actual time used by the tasks. If the thread pool available to the task manager is larger than one, then several tasks might run in parallel, but still in one tick. Both the tick duration and the thread pool size is set in the configuration file, and the tick duration can also be controlled interactively from within the GUI.

Activation

The *activation level* describes the *salience* (their relative value) of the various elements in the framework that implements the Activatible interface. This includes the nodes, links, coalitions, codelets, schemes, behaviors and more.

The activation level is represented by a decimal value between 0.0 and 1.0. The Activatible interface has two methods for controlling the activation level; *excite* and *decay*, and elements are usually removed if their activation level drops beneath a certain threshold (referred to as a *removal threshold* in the framework). These are controlled by separate elements, Strategies (described below), that are easily replacable and interchangeable.

Learning is also done as an extension to the Activation interface, by providing a *base-level activation*, which can for example represent the usefulness of a node in the past. We won't go into detail about the Learnable interface here, as the current version of the LIDA framework doesn't ship with any learning algorithms that use this interface.

GUI

The framework contains a highly customizable GUI that can be used to inspect every aspect of the implemented model at runtime, as well as provides a way to control the execution.

The appearance of the GUI is controlled by a "GUI Panels" property file which describes which custom and default panels are to be displayed in the GUI.

Strategies

Strategies encapsulate various algorithms that are shared between modules. Examples of Strategies are the decay and excitation strategies for the activation level; sigmoid and linear excitation and decay are the ones that are provided by default.

XML parser

Since the agent description is done in an XML file, the framework provides a complete XML parser that loads and creates an in-memory representation

of the agent.

Factory

This object factory provides a way to easily generate objects for common data structures and strategies. It is highly recommended that elements aren't created directly, but requested from the ElementFactory. This allows for easy addition of new subclasses of for example Node and Link or strategies. It also allows to easily define activation levels, strategies and parameters to be set in the created objects. An example given in the documentation[31] is that of several Nodes with the same NodeImpl class, but with varying excite and decay strategies, and different initial activations.

The element type definitions are defined in the factory data XML file which is parsed with the included XML parser, and each contains the name which is used when requesting a new object instance from the factory, the class and parameter values.

Framework initialization

The classes that does the initialization of the agent are in the *initialization* Java package. The main class for starting the initialization is called *AgentStarter*, and contains a main method to take in the location of the main configuration file.

1. First the main configuration file is loaded, which contains (among other things) the names of the other configuration files.

2. From there it gets the name of the factory data file, which contains the definitions for the element factory.

3. Then an instance of the Agent class is created based on the contents of the agent declaration file.

4. Then the GUI properties file is loaded and all the GUI objects (including custom panels) are created.

5. Finally the agent and GUI is loaded and displayed.

Parameter tweaking

Like any other (partially[5]) sub-symbolic artificial intelligence system, parameter tweaking is very important and small changes in the parameters can completely change how the agent behaves and acts. In LIDA there are a lot of activation values and thresholds for when something is counted as activated.

The feature detectors are the first instances where activation is important. When they discover something from the input they send activation to one or more nodes in PAM, and this activation level will decide how important the model thinks this piece of information is. In addition to this, it will also have cascading effects on the system further down the line. When attention codelets compete about access to the global workspace and consciousness broadcast the activation of the nodes contained in the codelets context together with the initial activation value of the codelet itself makes up the total activation of the codelet. So if some of the nodes have low activation from the feature detectors the codelet that contains that node will also be judged as unimportant.

Learning base level activations are part of the LIDA model but are not yet implemented in the current version of the framework, so we haven't been able to utilize it. What it implies is that when some nodes have been important in the past this will make them more likely to be important in the future, and this should have an impact on the performance of the agent.

Both feature detectors and codelets in the framework are implemented as processes that run at specified intervals, defined by the number of ticks between each time they are run. Changing how often a process is run will also impact the agent in not always predictable ways. If you do not detect the presence of some piece of information in time, you might decide on an action that is not the optimal way to handle the situation, which you might have done if you had all the information available to you. If the period between the execution of a feature detector is to long, you can end up with the related PAM node decaying below the threshold for activation between each time the detector is run, but if the detector was run at more regular intervals it would stay activated constantly because the node would receive excitation after each run of the detector.

All the elements in the LIDA framework that has activation have different parameters for how this activation is received and decayed. Decay of elements are configured differently for different modules. In Declarative Memory ele-

[5]LIDA is generally viewed as hybrid symbolic/sub-symbolic architecture.[14]

ments decay at a very slow rate, if at all, while in PAM they decay during a very short period of time, usually just a few ticks long. The same options for configuration of excitation exists in the framework.

To facilitate easy tweaking of parameters in LIDA most of them can and should be configurable from the XML definition of the agent. This makes changing and testing new values a lot easier then having to rewrite lines of code between runs.

Chapter 3

Implementation

In this chapter we describe how we implemented the project. In Section 3.1 we explain why we did what we did. In Section 3.2 we explain how we integrated BWAPI with LIDA. Section 3.3 explains the feature detectors we made. In Section 3.4 we explain how we manage attention in our agent. In Section 3.5 we explain how action are executed.

3.1 Overview

We chose to use the LIDA framework because it is built on a strong theoretical foundation of core cognitive principals. Even though the framework is currently, at the time of this writing, only in a beta version it is well-documented and has an active community working on improving it. Because LIDA is written in Java we had to use JNI-BWAPI since it provides Java bindings for BWAPI.

We made the decision to allow our agent access to perfect information about the environment in StarCraft, this means that it can see the whole map and all the units on the map without having to scout first. We did this because it simplified our implementation and made it easier to begin with to create an agent that functioned decently. With the same reasoning in mind, we decided that we would go for a simple strategy in our games, and limit our self to only use one race. So we decided to use the Protoss race, and our strategy involved building gateways, that are the the basic unit production building for Protoss, and then producing an army consisting of Zealots, a basic close-range melee unit.

3.2 Integration

BWAPI itself provides only a basic C++ API, so JNIBWAPI provides, as explained in Chapter 2.1, a custom Java API using the Java Native Interface, JNI.[23] Because we can't load an entire Java VM into the StarCraft process, we use shared memory to connect the StarCraft process with the process of our agent. Because the processes manipulates the same shared memory area both processes needs to be ran with Administrator privileges in Windows.

We also updated JNIBWAPI to adapt to some minor recent changes in BWAPI, and added in some missing functionality that we needed for controlling the game, like starting, pausing and restarting the on-going game.

In order to use the LIDA framework we had to integrate LIDA with JNIBWAPI. To accomplish this we implement the domain specific modules of the LIDA framework to make calls to JNIBWAPI.

Figure 3.1 shows a general overview of the Jantu bot's architecture, which consists of 3 main parts. In the StarCraft process the game itself runs together with BWAPI injected into the game client. Because our code needs to be run in the Java virtual machine, our code is run in a separate process that communicates with the Starcraft process using a shared memory bridge. This enables JNI-BWAPI to make calls to BWAPI and retrieve information back across the bridge.

In the Jantu process JNI-BWAPI is running together with the LIDA framework. In order to integrate the LIDA framework with JNI-BWAPI we setup the framework first with the configurations we needed to get it up and running. This involves describing and structuring the modules you need in different XML configuration files. Also in these files we configure what kind of information that will be possible to use and transmit internally in the LIDA framework.

JNI-BWAPI consists of different models and types that represent Starcraft information like units and buildings together with a lot of native functions that can be called to communicate with BWAPI. We integrated this with LIDA by using a custom implementation of the Environment class in LIDA. This class becomes the interface between the domain specific modules of LIDA, the sensory memory and sensory-action memory, and JNI-BWAPI.

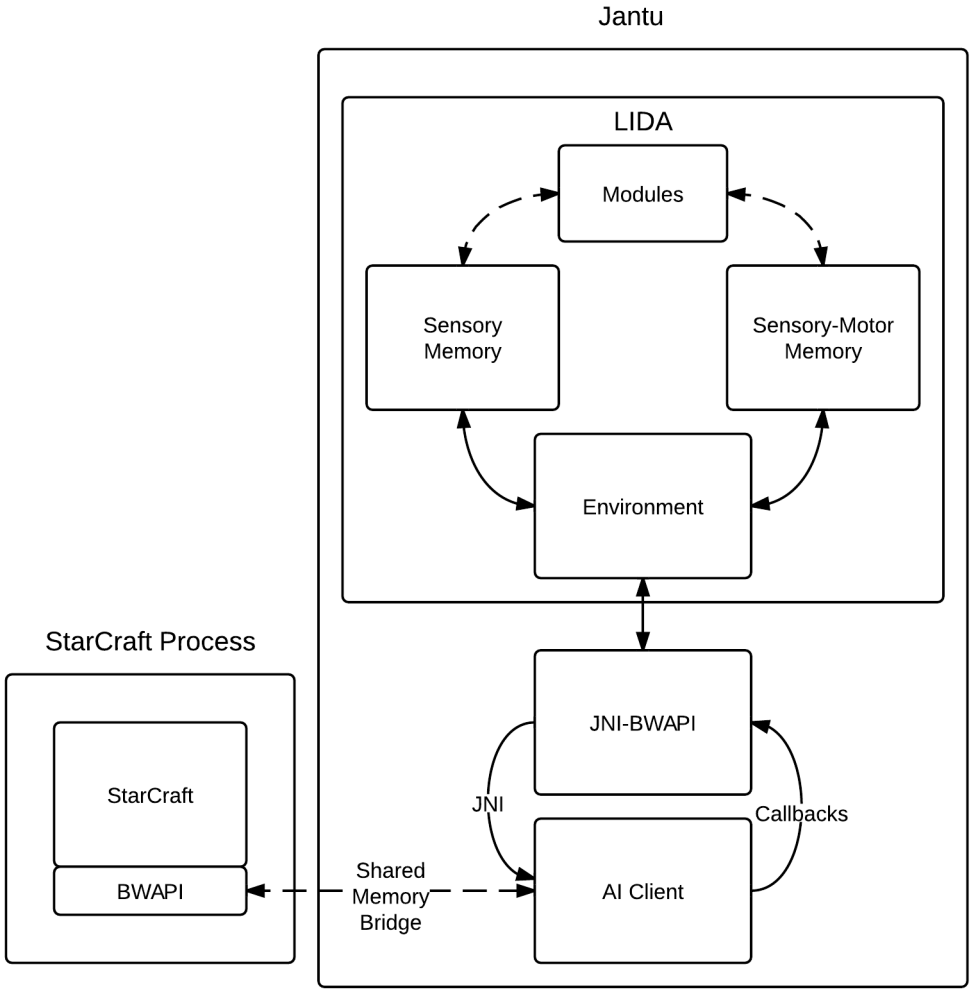

Figure 3.1: A general architecture overview of Jantu

Environment module

The Environment module is the interface between JNI-BWAPI and LIDA. It is responsible for abstracting away the JNI-BWAPI, and making sure that the game runs when it should. It allows LIDA to reset the state of the environment, by restarting the game.

Timing When working with a cognitive architecture it is important to be able to look at the internal structures of the model during runtime, like the current situation model and what node structures have what activations.

This is useful for debugging and performance tweaking.

To achieve this we have to be able to pause and resume the game at any time, so to get the run/pause and timing functionality of LIDA inside StarCraft, we have a semaphore[1] that is released by a LIDA codelet each tick, and then the callback from StarCraft/BWAPI that we get each frame waits for this to be released. This allows us to easily set the amount of StarCraft frames that are processed for each LIDA cycle, by increasing the amount of permits available in the semaphore. In our implementation we let one cognitive cycle equate to one in-game frame.

GUI panel We also provide a custom GUI panel to represent the environment, see Figure 3.2. Different regions from the Brood War Terrain Analyzer are separated by gray lines. Enemy entities are displayed as red dots, entities owned by our agent are blue dots. Neutral entities (vespene geysers and mineral fields) are green. Choke points are marked by yellow circles.

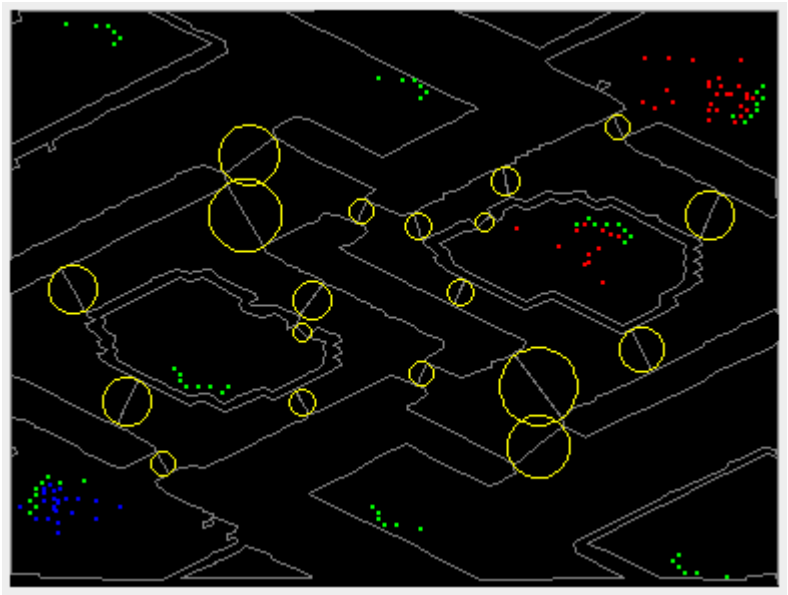

Figure 3.2: Our custom representation of the state of the environment.

[1] A semaphore is a special data structure that is used for synchronization of threads. It has a number of *permits*, which a thread can request. A thread will pause while waiting for an available permit. Another thread can then release permits in the semaphore to let the waiting threads continue.

3.3 Detectors

Feature detectors are how LIDA perceives it's environment and identifies important aspects of the current game state. They are tasks that are run repeatedly, at specific intervals and parses the game state at that time in order to identify a given feature that can later be used in different modules in LIDA, to recognize thoughts and concepts. Each detector usually only identifies one specific feature, but it is possible for one detector to identify several features if they are of the same type.

Feature detectors can be created that detect almost every aspect of the game, and they can be everything from simply detecting the existence of specific units or game elements to more complex detectors that detect army compositions or enemy tactics and strategies. The more of them you implement the more advanced concepts you can identify and that opens up more advanced strategies you can perform yourself.

3.3.1 Implemented feature detectors

Following are brief descriptions of the detectors we have implemented.

IdleWorkerFeatureDetector

Identifies worker units that don't have a job. A worker could be gathering resources, constructing buildings or scouting, and for efficiency it should be doing something at all times.

ResourceFeatureDetector

Identifies what type of units and buildings that we currently have enough resources available to create. This can be buildings we can construct, units we can morph or upgrades that we can research.

SupplyBlockFeatureDetector

Identifies when we are getting close to being supply blocked, that means that we can't build any more units until another supply-granting building/unit is created.

UnsaturatedResourcesFeatureDetector

Identifies whether or not our available resource nodes have are saturated with enough workers that are gathering them. This can be used to decide if we need to build more workers or not.

BuildOrderFeatureDetector

Identifies what type of building we need to construct for our next step in the planned build order. For our simple implementation it detects if we have more resource generation then unit production buildings, so we need to expand our capacity by adding new ones.

IdleProductionFeatureDetector

Identifies when a production building is currently not training any units. In StarCraft being effective with the resources you have available is important, therefore not queuing units on a production building is important, but instead only building new units when production buildings are idle.

StructureFeatureDetector

Identifies the buildings we have. This is important to know because we might not want to build another of the same building, and for unit production it is important to know what kind of production buildings we have available.

TimingAttackFeatureDetector

Identifies when it is time to attack the enemy. In our current implementation is a very simple detector that just waits until we have collected a substantial army of units and then activates to "detect" that it is now time to attack.

3.4 Attention Module

One of the main features of a cognitive system, and even of the human brain, is the ability to focus consciousness or give attention to a subsection of world

as it perceives it. This is what the *attention module* is responsible for. It achieves this by running attention codelets that look for specific workspace content from the current situational model and them creates coalitions from this that are added to the global workspace, where they will compete with other coalitions for conscious focus.

The activation for a coalition is based both on the activation for the PAM nodes that it contains and the activation for the codelet itself that created it.

3.4.1 Implemented attention codelets

Idle worker codelet

A simple codelet that makes a coalition with idle worker node. This codelet has a high activation in order to make sure has a big change of winning the competition for consciousness since it is important for workers to be doing something at all times.

Build worker codelet

This codelet looks for three different situations that decides if a worker should be created. These are if we can afford to create the worker, if our resources are not saturated already and the base building is not currently training any units.

Build supply codelet

This is another simple but important codelet that has a high activation value. It looks for the situation where we are supply blocked, or getting close to being supply blocked, and we can afford to build a supply structure. It has a high activation level because getting supply blocked means you can't create any more units until another supply structure is finished building, that will be a lot of valuable time lost where no units is trained.

Build Order Codelet

This is one of the more complex codelets, it looks for all the buildings that we can currently afford as well as all the buildings that we need to create according to the build order nodes. Combining these in a coalition allows the action selector later to decide what building should be created next.

Train Units Codelet

This codelet is similar to the Build Order Codelet, except it looks for the units we can afford to train in addition to what unit production buildings we currently have available to use.

Strategy Codelet

This codelet looks for any node that is a child from the strategy node. These are defensive or offensive strategies during a game. In our implementation we only have a simple strategy that is to attack with every army unit we have at the same time, when we feel we have a sufficient army size.

3.5 Action Execution

After LIDA has selected a focus of consciousness and an appropriate action has been selected from the behavior net, the selected action must be executed. This gets executed by having the sensory-motor memory make function calls to the environment class, which functions as a bridge between LIDA and JNI-BWAPI.

How the action is implemented in the sensory-memory is up to the developer, as long as it performs its given function. It can be really simple implementation or you could do something more complex, LIDA is there to decide what parts or game state to focus on, and to make the decision on what action to perform, not how that action is executed.

3.5.1 Implemented actions

Mine minerals

This actions takes a worker that is not currently performing a task and orders it to start mining minerals from an unsaturated mineral deposit.

Build worker

This action starts training a worker unit at the main building in the home base, the Nexus for Protoss.

Build supply

This action constructs a building or unit that grants increased supply. This can be a building or a unit depending on the race. For Protoss it creates a structure called a pylon that grants more available maximum supply.

Build gateway

This action works in much the same way as *Build supply* except that it creates a production building called gateway. This building is used to warp in (train) army units.

Train zealot

This action sets a zealot for training at an available gateway that is currently not training any other units. It is important that it doesn't queue it at a gateway that is already training another unit since it locks up resources and is inefficient.

Attack

This action executes an attack on enemy units. It can be a very simple action like in our implementation where it just used every available army unit and attacks the enemy, or it could be more complex like using groups of units for different attacks. Also worth noting that if you are not using perfect

information then you can only attack units that you have seen by scouting, you are being invaded by or otherwise manage to see.

Chapter 4

Results

In this chapter we present the results of what we have created. In Section 4.1 we look at how we performed against the built-in AI in StarCraft: Brood War, and in Section 4.2 we review how a cognitive cycle was evaluated in our finished agent.

4.1 Agent performance

In this section we describe the performance test we ran and the results that it yielded.

4.1.1 The test

To measure how well the agent performed when it played we ran 50 games against the built-in AI in StarCraft Brood War. We only ran 50 games because the performance of the agent is very dependent on how how many feature detectors are implemented and how much of the game that the agent is capable of understanding. Because we only implemented a very basic agent as proof of concept it will not be smart enough to handle all the situations that it will get into, and therefore the performance is not any good measurement of how well the method will perform if expanded upon with more complex detectors and schemes. Therefore we did not deem it relevant to spend significant time measuring this performance.

All the games where played on the same map, Astral Balance, to avoid any

Victories	Defeats
17	33

Table 4.1: Results from performance test.

map specific problems that could mess with the agents performance. As explained in section 3.1 our agent only played as the Protoss race, and the enemy was set to random so they could be any of the three available races. We gave the agent access to perfect information in the game, because the built-in AI also utilizes perfect information, and that means it does not have to scout to know where the enemy is located.

4.1.2 The result

Table 4.1 shows the results from the performance test. After the 50 games we ended up with 17 victories and 33 defeats, which is a win percentage of 34%. In a few situations the agent got stuck after it had destroyed all the enemy buildings and unit except for one or two in an expansion or hiding in a corner. These games we judged as a win because the enemy had no chance of rebuilding no matter how long it was left alone in this state.

4.2 A cognitive cycle

A cognitive cycle is the sequence of processes that the cognitive model has to go through from sensing to execution of an action. In LIDA the sequence starts with sensing of the environment, next is understanding, then attention and last is the selection of an action and its execution. These cycles run constantly and repeatedly for the "lifetime" of the agent.

Following we describe the complete cognitive cycle that ends with the creation of a new worker unit.

4.2.1 Sensing

The first step in the cognitive cycle is sensing. Here sensory-memory samples the environment for new input to the model before low level feature detectors analyzes the input looking for specific features, ideas and meanings. If the feature detector finds something it sends excitation to a corresponding feature

node in the perceptual associative memory (PAM). The amount of excitation can be anything from zero to one, depending on how important or present the given feature is currently. Figure 4.1 shows the table of nodes that we implemented in the project, this is before StarCraft starts running so all the nodes have zero activation yet. See section 3.3 for a list of what detectors we implemented in our agent that will feed activation to these nodes.

Node	ID	Current Acti...	Base-Level ...	Threshold
supplyBlocked	22	0,0000	0,1000	0,5000
resources	12	0,0000	0,1000	0,5000
affordZealot	20	0,0000	0,1000	0,5000
idleNexus	6	0,0000	0,1000	0,5000
strategy	34	0,0000	0,1000	0,5000
gateway	28	0,0000	0,1000	0,5000
affordGateway	18	0,0000	0,1000	0,5000
None	0	0,0000	0,0000	0,5000
timingAttack	36	0,0000	0,1000	0,5000
nexus	26	0,0000	0,1000	0,5000
idleGateway	8	0,0000	0,1000	0,5000
Parent	2	0,0000	0,0000	0,5000
needGateway	30	0,0000	0,1000	0,5000
unsaturatedResources	10	0,0000	0,1000	0,5000
affordSupply	16	0,0000	0,1000	0,5000
Lateral	1	0,0000	0,0000	0,5000
structures	24	0,0000	0,1000	0,5000
affordWorker	14	0,0000	0,1000	0,5000
Feature	3	0,0000	0,0000	0,5000
buildOrder	32	0,0000	0,1000	0,5000
idleWorker	4	0,0000	0,1000	0,5000

Figure 4.1: The PAM table with current activation values.

When a node receives excitation it also sends a portion of this excitation upstream to any node that it has as a parent. This way meta-nodes like *resources* and *structures* can receive activation even though there are no feature detectors that directly affects this node. Figure 4.2 shows our current PAM graph with the relationships between the nodes.

4.2.2 Understanding

After the sensing phase and nodes in PAM have received excitation, any node that has an activation that is higher then the threshold will be moved to the Perceptual Buffer that is a sub module of the Workspace module. The content of this buffer and any relevant information from the Transient

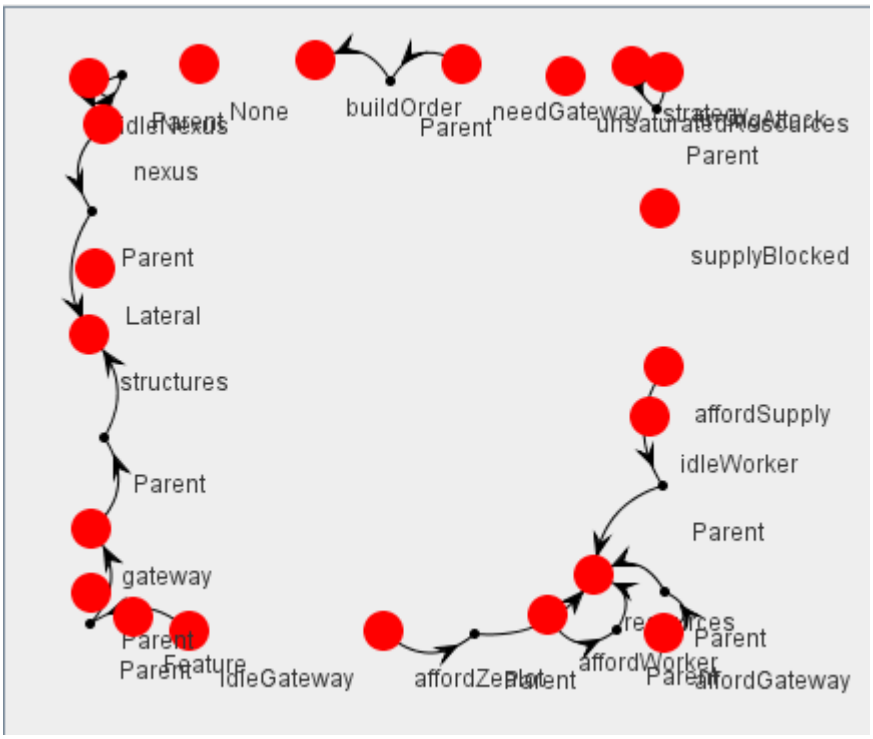

Figure 4.2: Representation of the PAM graph.

Episodic Memory and Declarative Memory will make up the Current Situational Model. The CSM has the structures that represents the agents current understanding of the world. Since we do not utilize the TEM and DM modules CSM contains only data from the Perceptual Buffer. Figure 4.3 shows the content of CSM for our current cycle.

We can see that the agent knows that we can afford to make a worker and that our resources, our mineral line, is not saturated. It also knows that the nexus is idle, not producing any units at the moment. These are the relevant pieces of information for creating a worker. But the CSM also contains information about the need for a gateway, but because we can't afford it it will not be selected as a focus for consciousness in the next phase.

4.2.3 Attention

After the CSM is updated the competition for consciousness begins. It is the job of attention codelets to bring important or urgent content to consciousness. The attention codelet creates a coalition of content from the CSM and

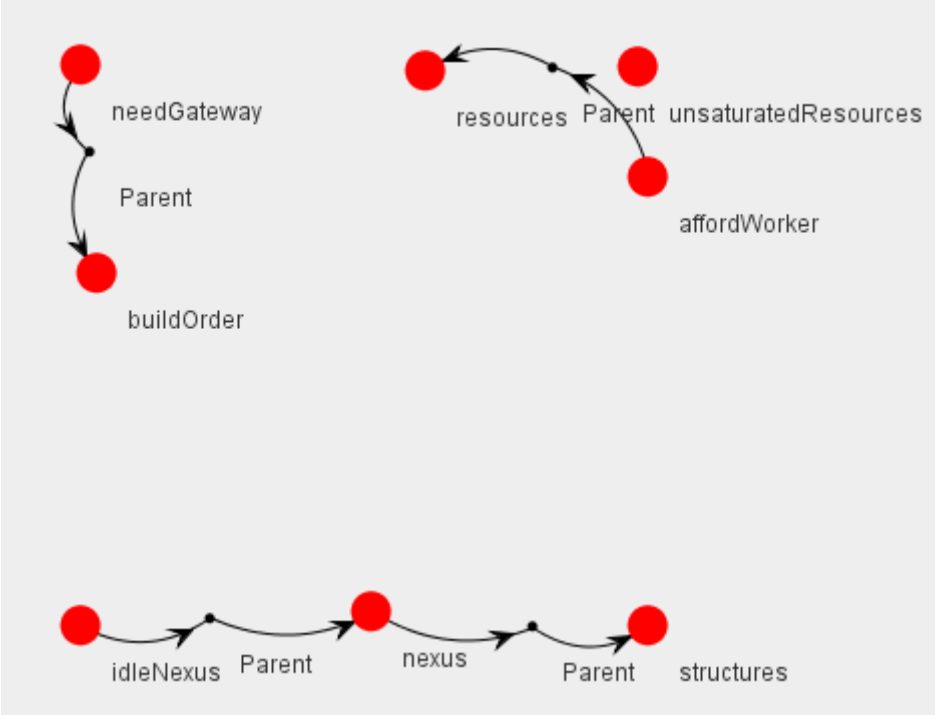

Figure 4.3: The node structure in the CSM.

gains activation based on the activation of both the nodes it gathers and the activation level of the codelet itself. All the codelets that are active, can be both new codelets and codelets from a previous cycles that has not yet decayed, then compete for the chance to be broadcast from the global workspace. In Figure 4.4 we see the codelets competing for consciousness in this cycle. We see that the codelet with a coalition of idleNexus, unsaturatedResources and affordWorker has the highest activated and will win the competition. It will therefore be the focus of consciousness when it is broadcast from the Global Workspace.

Activation	Coalition NodeStructure
0,4167	Nodes (resources[12],needGateway[30],affordWorker[14],buildOrder[32]) Links (Parent[2,1...
0,4217	Nodes (resources[12],nexus[26],structures[24],affordWorker[14]) Links (Parent[2,26,-2147...
0,5820	Nodes (resources[12],idleNexus[6],nexus[26],unsaturatedResources[10],affordWorker[14])...

Figure 4.4: Contents of the global workspace.

The coalition that gets broadcast is used for learning in several of the LIDA modules. But because learning is not implemented in the current version of the LIDA framework we could not utilize this.

4.2.4 Action selection

The last phase of the LIDA cognitive cycle is the action selection. The first step of this phase is the Procedural Memory receiving the conscious broadcast from the Global Workspace. The Procedural Memory contains the scheme net and using the information from the broadcast some of the schemes will increase their activation if their context is contained in the broadcast. The schemes are put above their threshold are instantiated as behaviors and then sent to the Action Selection module. Figure 4.5 shows all the available schemes in our implementation, and the current activated scheme is the one with the action to create a worker. The build worker scheme is activated because all of the nodes in its context was received with the conscious broadcast from last phase.

Scheme Labe	Current	Context	Action
Build supply	0,0000	Nodes (supplyBlocked[22],affordSupply[16]) Links ()	action.buildSupply
Attack	0,0000	Nodes (timingAttack[36]) Links ()	action.timingAttack
Build worker	0,6933	Nodes (idleNexus[6],unsaturatedResources[10],affordWorker[14])	action.buildWorker
Build gateway	0,0000	Nodes (affordGateway[18],needGateway[30]) Links ()	action.buildGateway
Mine minerals	0,0000	Nodes (idleWorker[4]) Links ()	action.mineMinerals
Train zealot	0,0000	Nodes (affordZealot[20],gateway[28],idleGateway[8]) Links ()	action.trainZealot

Figure 4.5: Contents of the procedural memory.

After some schemes are activated into behaviors they are received by the Action Selection module, where the final decision about what action is going to be performed in this cycle is taken. The LIDA module describes this module as a behavior network[26]. But in the current version of the LIDA framework this has not been implemented yet. So in the current version the Action Selection module is very simplified, which makes it harder to do complex action selections. The current action selection mainly just selects the behavior with the highest activation. So the real selection actually happens in the Procedural Memory. But with the release of the next version of the framework that should change(May 2012). Figure 4.6 shows the content of the Action Selection module for this cognitive cycle. Our one behavior doesn't have any competition so it is selected as the action to perform this cycle. Below are a list of the five last performed actions in past cycles.

When the action is selected it is passed on to the Sensory-Motor Memory that will use its actuators on the environment to execute the action. In our cycle this will involve starting to build a worker unit from the Nexus building.

Behavior Label	Activation	Context	Action
Build worker	0,5600	Nodes (idleNexus[6],un...	action.buildWorker

Tick at Selection	Action
2360	action.buildWorker
510	action.mineMinerals
410	action.mineMinerals
310	action.mineMinerals
210	action.mineMinerals

Figure 4.6: Contents of the action selection module

Chapter 5

Evaluation

In this chapter we conclude our work by looking at the goals defined in the introduction, and evaluate the results. In Section 5.1 we evaluate our implementation. In Section 5.2 we evaluate the in-game performance of our implementation. Section 5.3 contains the final conclusion for the project and this report. Section 5.4 contains our thoughts on further improvements that can be made on our project.

5.1 Evaluation of Implementation

Our motivation for this project was to create an agent based on the LIDA model, that could play StarCraft: Brood War. we did this by implementing a simple proof of concept agent using the LIDA framework. We also discovered some of the problems inherent in the use of this cognitive architecture for this domain.

During the integration between LIDA and StarCraft we ran into a few problems. The most important one being that measurement of time in both processes are different. LIDA operates with ticks, where the length of a tick is dynamic, and is decided by the slowest LIDA process that is run in that specific tick. We had to sync these ticks up with the StarCraft measurement of time which is frames. Because we have to pause the StarCraft game for each frame until the LIDA model returns from that tick this will probably disqualify the agent from participating in certain tournaments.

We discovered that because StarCraft is such a complex game with so many different possible states and situations the amount of feature detectors you

have make a very significant impact on the performance of the agent. We only implemented some basic feature detectors to give the agent enough information about the environment to perform some simple tasks that made for some competition for the built-in AI in the game. But we constantly observed the agent getting into situations that for us as human observers was very clear, but the agent didn't see them. One such situation that happened repeatedly was that the game AI would attack us at the same time as we moved out to attack them. And because our agent wasn't capable of detecting this, it would never turned around to defend his own base against this attack.

But for every situation that we observed we could not deal with, there was obvious that simply more detectors would be able to parse this data and understand the situation. We were unable to see anything that would be outside the scope of reasoning with and handling in our architecture, if it had the right set of information to work with. Expanding upon the set of feature detectors would probably be one of the main priorities for improving the performance of the agent.

As mentioned in Section 4.2.4 the Action Selection module in the current implementation of the LIDA framework is not fully implemented according to the definition in the LIDA model. Because of this the action selection in our agent is very limited in comparison to what it could be if the model was fully realized. In our agent the action selection module simply selects the behavior that has the highest activation when it gets instantiated in the Procedural Memory. This turned out to be one of the biggest limitations we discovered during implementation because we didn't have the option for one behavior to inhibit other behaviors. An example of this is the behavior that creates a new gateway, we didn't want to keep building new gateways after we made the first one. But because we were unable to inhibit this behavior the agent kept creating new gateways. There wasn't a good solution for this problem other then moving some of this logic to the feature detectors, and that is why we created the build-order feature detectors. Ideally you would probable want this logic in the action selection module.

LIDA framework version 1.2b released May, 2012 is the first version of the framework to include the Behavior Network that the LIDA model describes for the Action Selection module. We did not have time to include this update in our agent. The Behavior Networks adds a lot of utility to the Action Selection module. In essence it takes the instantiated behaviors from the Procedural Memory as input to a network of behaviors that can effect each other. Each activated behavior can impact other behaviors in the network

by acting as amplifiers or inhibitors to connected behaviors. The behavior with the highest activation in the end is then selected as the action to be executed by the sensory-motor memory. With the updates LIDA framework the agent should be able to do much more complex reasoning in the action selection stage, and one could remove this logic from the feature detectors where it do not really belong.

Another important aspect that the LIDA framework is currently missing that is defined in the model description is learning. When the consciousness broadcast occurs several of the LIDA modules are suppose to receive the broadcast and use this for leaning. PAM is suppose to use this to learn different base activation values, nodes that have proven to be useful will have a grater change of being activated again. Also the Transient Episodic Memory should store the current focus of consciousness as short-term memory, and there is a chance for this to be moved to Declarative Memory for long term storage. These are just some of the modules that are suppose to learn during execution, a full list of the model defined learning types are: Procedural learning, Perceptual learning, Episodic learning and Attention learning. But none of these are currently supported by the framework, but they will be in in a future release.

The work we have done with integrating and implementation for our agent should simplify the work of anyone else intending to do something similar. Our domain-specific classes should provide everything needed for any kind of agent using the LIDA framework for StarCraft: Brood War, and our implemented feature detectors and behavior should be a good starting point, as well as a good example for other feature detectors and schemes one needs to implement to create more complex behaviors for the agent.

5.2 Evaluation of Agent Performance

From the performance tests we ended up with a win ratio of 35%. That is not to high, but it was also never our goal to create the perfect agent. Each time we lost, it was usually because the agent ended up in a situation where it didn't have the correct detectors and behaviors to handle that situation. This should mean that it is only a matter of extending on the implementation to improve the performance of the bot. The underlaying method seems to be working just fine when the environment is simple enough for this implementation to handle, so it should scale well with more complex detectors and schemes.

The problems we encountered that made us lose games was for instance that our agent does not detect the presence of enemy units attacking us, and we therefore rely on the built-in auto attack of enemy units for protection of our base. This often leads us to "base-trades", where we are able to destroy the enemy base, but are unable to protect our own.

We lack any kind of advanced micro behavior, which means we utilize the units produced very badly. This means that early aggression was very ineffective because they requite good micro of a few units, so we ended up removing that behavior from the agent. We also had big problems when the enemy created base defenses, like Protoss canons and Zerg spine crawlers. One observed behavior a long line of zealots simply marching into the line of fire of a canon, so they were cut down one by one. A more effective strategy here would be to group up the units, so some of them would be able to reach the canon and possibly destroy it.

Another issue is that our bot only builds one particular unit. This makes it very easy for a human to counter, for example with air units, which the zealots are unable to attack. When the behavior net is implemented in the agent it should be a lot easier to create more complex multi unit-type army compositions. Our agents also never expands to additional bases. While this made things easier to implement (positioning of units etc. became easier to work with), it means that if we don't win quickly, we will run out of minerals, and we will be at a severe economic disadvantage if and when the opponent expands to several bases.

5.3 Conclusion

Our goals for his project was to integrate a cognitive model with BWAPI so we could use this as a base for building a proof of concept artificial intelligence agent that played StarCraft using cognitive methodologies.

We succeeded in creating this in the form of the Java agent Jantu that runs the LIDA framework on top of Java bindings for BWAPI. We discovered several limitation with the method, most of whom has to do with missing features in the framework that will be implemented in future releases. And while the in-game performance was not excellent, it is a very good starting point for any future work in this area.

We discovered that the domain of StarCraft is big, and that there are a lot of different situations that an AI agent will have to handle if it is going to

perform well over time. For a cognitive implementation this means that it will require a lot of different feature detectors and schemes to handle every situation it might encounter. It is not given, however, that it will scale well when you start adding a lot of detectors and schemes, and it might become hard to keep track of the different states that will occur in Action Selection as the complexity grows.

5.4 Future Work

There are several areas of this project that can be improved upon. Implementing more feature detectors, and more behaviours, might be the easiest and obvious way to go forward. Some core missing behaviors are for example grouping of units and managing the individual units better, expanding to extra bases for additional income, as well as more complex build orders and attack patterns.

A main priority should be to update the LIDA framework to the newest version and replacing the Action Selection Module with the more advanced Behavior network that was implemented in version 1.2 of the LIDA framework. This will be a significant change that will give more power to the decision making process of the agent, and make it more capable of complex reasoning.

Another significant improvement would be to use the various memory modules available more actively.[19]. This should help increase the complexity of the current situational model by letting the bot remember more about previous states of the game. But in order for the memory modules to function properly, learning has to be first implemented into the framework according to the LIDA model specifications. The addition of learning will also help on modules then just episodic memory. When learning of base-level activation is implemented in the LIDA framework this can be used for improving the agents performance, by learning what build orders are good, and learning to prioritize e. g. military strength vs. economic stability.

Bibliography

[1] R. Arrabales, A. Ledezma, and A. Sanchis. "CERA-CRANIUM: A test bed for machine consciousness research". In: (2009).

[2] R. Arrabales, A. Ledezma, and A. Sanchis. "Towards conscious-like behavior in computer game characters". In: *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE. 2009, pp. 217–224.

[3] B. J. Baars. "Global workspace theory of consciousness: toward a cognitive neuroscience of human experience". In: *The Boundaries of Consciousness: Neurobiology and Neuropathology*. Ed. by Steven Laureys. Vol. 150. Progress in Brain Research. Elsevier, 2005, pp. 45 –53. DOI: 10.1016/S0079-6123(05)50004-9. URL: http://www.sciencedirect.com/science/article/pii/S0079612305500049.

[4] B. J. Baars. "How Brain Reveals Mind Neural Studies Support the Fundamental Role of Conscious Experience". In: *Journal of Consciousness Studies, 10* 9.10 (2003), pp. 100–114.

[5] B. J. Baars and S. Franklin. "An architectural model of conscious and unconscious brain functions: Global Workspace Theory and IDA". In: *Neural Networks* 20.9 (2007), pp. 955–961.

[6] M. Bachle and P. Kirchberg. "Ruby on rails". In: *Software, IEEE* 24.6 (2007), pp. 105–108.

[7] *Brood War Application Programming Interface*. BWAPI Team. Dec. 2011. URL: http://code.google.com/p/bwapi/.

[8] *Broodwar Terrain Analyzer*. BWTA Team. Dec. 2011. URL: http://code.google.com/p/bwta/.

[9] M. Buro. "Real-time strategy games: a new AI research challenge". In: *Proceedings of the 18th international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 2003, pp. 1534–1535.

[10] M. Buro and T. Furtak. "RTS games as test-bed for real-time AI research". In: *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)*. 2003, pp. 481–484.

[11] *BWAPI Standard Add-on Library*. BWSAL Team. Dec. 2011. URL: http://code.google.com/p/bwsal/.

[12] M. Campbell, A.J. Hoane, F. Hsu, et al. "Deep blue". In: *Artificial intelligence* 134.1-2 (2002), pp. 57–83.

[13] S. Dehaene and L. Naccache. "Towards a cognitive neuroscience of consciousness: basic evidence and a workspace framework". In: *Cognition* 79.1 (2001), pp. 1–37.

[14] W. Duch, R.J. Oentaryo, and M. Pasquier. "Cognitive Architectures: Where do we go from here?" In: *Proceeding of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*. IOS Press. 2008, pp. 122–136.

[15] S. Franklin, A. Kelemen, and L. McCauley. "IDA: A cognitive agent architecture". In: *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*. Vol. 3. IEEE. 1998, pp. 2646–2651.

[16] S. Franklin and L. McCauley. "Interacting with IDA". In: *Agent autonomy* (2003), pp. 159–186.

[17] S. Franklin and FG Patterson Jr. "The LIDA architecture: Adding new modes of learning to an intelligent, autonomous, software agent". In: *pat* 703 (2006), pp. 764–1004.

[18] S. Franklin et al. "Global Workspace Theory, its LIDA model and the underlying neuroscience". In: *Biologically Inspired Cognitive Architectures* 0 (2012), pp. –. ISSN: 2212-683X. DOI: 10.1016/j.bica.2012.04.001. URL: http://www.sciencedirect.com/science/article/pii/S2212683X12000060.

[19] S. Franklin et al. "LIDA: A computational model of global workspace theory and developmental learning". In: *AAAI Fall Symposium on AI and Consciousness: Theoretical Foundations and Current Approaches*. 2007, pp. 61–66.

[20] S. Franklin et al. "Virtual Mattie-an intelligent clerical agent". In: *AAAI Symposium on Embodied Cognition and Action, Cambridge MA*. 1996.

[21] W.J. Freeman III and L.J. Rogers. "A neurobiological theory of meaning in perception. Part 5. Multicortical patterns of phase modulation in gamma EEG". In: *International Journal of Bifurcation & Chaos* 13 (2003).

[22] *Hindi dictionary entry for jantu*. June 2012. URL: http://hamariweb.com/dictionaries/animal_hindi-meanings.aspx.

[23] *Java Native Interface: Programmer's Guide and Specification*. June 2012. URL: http://java.sun.com/docs/books/jni/.

[24] P. Kanerva. *Sparse distributed memory*. The MIT Press, 1988.

[25] R. Llinás et al. "The neuronal basis for consciousness". In: *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 353.1377 (1998), pp. 1841–1849.

[26] Pattie Maes. "How to do the right thing". In: *Connection Science* 1.3 (1989), pp. 291–323.

[27] *Protoss Logo - TeamLiquid wiki*. June 2012. URL: http://wiki.teamliquid.net/starcraft2/index.php?title=File:ProtossIcon.png&filetimestamp=20100802025319.

[28] O.G. Selfridge. "Pandemonium: a paradigm for learning in mechanisation of thought processes". In: *Proceedings of a Symposium held at the National Physical Laboratory*. HMSO. 1958, pp. 513–526. URL: http://webdocs.cs.ualberta.ca/~sutton/pandemonium.pdf.

[29] M. Shanahan and B. J. Baars. "Applying global workspace theory to the frame problem". In: *Cognition* 98.2 (2005), pp. 157–176.

[30] J. Snaider, R. McCall, and S. Franklin. "The LIDA framework as a general tool for AGI". In: *Artificial General Intelligence* (2011), pp. 133–142.

[31] J. Snaider et al. "The LIDA Tutorial". In: (2012).

[32] *Student StarCraft AI Tournament*. June 2012. URL: http://www.sscaitournament.com/.

[33] *Terran Logo - TeamLiquid wiki*. June 2012. URL: http://wiki.teamliquid.net/starcraft2/index.php?title=File:TerranIcon.png&filetimestamp=20101020145937.

[34] *The homepage of the JNIBWAPI project*. June 2012. URL: https://code.google.com/p/jnibwapi/.

[35] *Wargus*. The Wargus Team. Dec. 2011. URL: http://wargus.sourceforge.net/index.shtml.

[36] B. Weber. *EISBot Plays Dennis "Thresh" Fong*. Dec. 2011. URL: http://eis-blog.ucsc.edu/2011/05/eisbot-plays-dennis-thresh-fong/.

[37] S. Wintermute, J. Xu, and J.E. Laird. "SORTS: A human-level approach to real-time strategy AI". In: *Ann Arbor* 1001 (2007), pp. 48109–2121.

[38] *Zerg Logo - TeamLiquid wiki*. June 2012. URL: http://wiki.teamliquid.net/starcraft2/index.php?title=File:ZergIcon.png&filetimestamp=20100802021610.