University of Sheffield

CITY Liberal Studies

Department of Computer Science

FINAL YEAR PROJECT

StarPlanner

Demonstrating the use of planning in a video game

This report is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by

Panagiotis Peikidis

May, 2010

Approved

Petros Kefalas

Konstantinos Dimopoulos

StarPlanner

Demonstrating the use of planning in a video game

by

Panagiotis Peikidis

Petros Kefalas

Abstract

Many video games today do not use Artificial Intelligent techniques to augment the decision making of Non-Player Characters. One of the main reasons is that these techniques require a significant amount of resources that make them unfeasible. Instead, other techniques that produce similar results are used. However, the demands of modern video games are increasing and many of these techniques do not scale.

One Artificial Intelligence technique that has successfully shown to scale well is planning. Ever since it was first used in First Encounter Assault Recon, the video game industry has been growing an interest in it.

The aim of this dissertation is to demonstrate that the use of planning meets the demands of modern video games. To achieve this goal an Artificial Intelligent system that uses planning will be developed and integrated in the StarCraft video game. This system will then be evaluated according to the demands of modern video games.

Keywords: planning, Goal-Oriented Action Planning, StarCraft, Game AI

DECLERATION

All sentences or passages quoted in this thesis from other people's work have been specifically acknowledged by clear cross referencing to author, work and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this thesis and the degree examination as a whole.

Name: Panagiotis Peikidis

Signed: _____ Date: 5 June 2010

Acknowledgments

First and foremost I would like to thank my parents, for without their unconditional support during my Bachelors degree, none of this would be possible. Special thanks to my brother and close friends as well, for their development tips and feedback.

I would also like to thank my supervisor and mentor Petros Kefalas for having the patience of putting up with me and constantly pushing my limits.

Finally, I would like to thank the community of AiGameDev.com, particularly Alex J. Champandard, Jeff Orkin, William van der Sterren and Ben Weber. They provided excellent insight for my numerous queries regardless of how vague or demanding they were.

Table of Contents

Table of Figures

Table of Code Listings

1 Introduction

1.1 Motivation

Artificial Intelligent techniques are sparsely used in video games to enhance the decision making of Non-Player Characters (NPCs) [1]. The significant amount of resources required for these techniques renders them impractical. To combat this limitation, many video games employ other means of producing the illusion of decision making. However, these techniques are gradually becoming less adequate as the demands of modern video games increase [2]. Demands such as reusability and extensibility are crucial from a company's point of view [3], while demands such as challenging and entertaining are valuable from a consumer's point of view [4].

A company's demands for reusability and extensibility are enforced by two separate groups of people: the people running the company, who aim to reduce development costs and the people creating the video game, who desire a flexible and easy-to-use system.

These demands have been acknowledged from the video game industry and, for the past couple of years, various groups and conferences have been formed to address them. The Artificial Intelligence Interface Standards Committee (which is now part of the Special Interest Group for Artificial Intelligence) is one such group. Their goal is "*to provide and promote interfaces for basic AI functionality, enabling code recycling and outsourcing thereby, and freeing programmers from low-level AI programming as to assign more resources to sophisticated AI [in video games]*" [5].

Among the various Working Groups of the AI Interface Standards Committee is the Working Group on Goal-Oriented Action Planning. Goal-Oriented Action Planning is a decision making architecture that received critical acclaim for the decision making of NPCs in the video game First Encounter Assault Recon (F.E.A.R.) [6]. Numerous video games have implemented this technique since [7]. Furthermore, Hierarchical Task Network Planning has been the subject of many recent conferences, such as the Paris Game AI Conference in 2009 [8].

It is evident therefore, that planning in video games is gaining much attention.

1.2 Aim

The aim of this dissertation is to demonstrate that the use of automated planning in a video game meets the demands of today's video game industry.

1.3 Objectives

To accomplish this aim, the objectives are:

- to understand what planning is and how it works,
- to understand what are the demands of contemporary video games,
- to understand what is unique in video games in terms of Artificial Intelligence,
- to develop a planning system that adheres to these requirements,
- to integrate the system in a video game
- and evaluate

Understanding planning and the requirements of video games will guide the development of a planning system. The system will then be integrated into a video game for evaluation. Understanding the demands of today's video games will facilitate the evaluation process. The outcome of the evaluation will demonstrate whether the use of automated planning meets the demands of contemporary video games and thus achieve the aim.

1.4 Structure

The process of achieving each objective is depicted in the remaining chapters of this document. The structure of the document is such that each chapter contains more specialized knowledge of its predecessor.

More specifically, Chapter 2 contains the general knowledge acquired during research on Artificial Intelligence in Video Games (Section 2.1), Agents (Section 2.2) and Automated Planning (Section 2.3). At the end of the chapter the topics are united and specific technologies are chosen.

Chapter 3 is an analysis of these technologies for the purpose of understanding the inner workings. Section 3.1 explains how Goal-Oriented Action Planning is structured while Section 3.2 explains the basic concepts of StarCraft. Finally, Section 3.3 looks at how one can develop an AI system for the video game StarCraft.

Starting with the design of the overall proposed system, all subsequent sections in Chapter 4 describe the iterations of the solution.

In Chapter 5, using the requirements identified in Section 2.1.2 through 2.1.4, evaluates the solution developed.

Finally, Chapter 6 contains some final notes, difficulties experienced throughout the project and the conclusion.

2 Field Review

At the highest level, this dissertation correlates with Artificial Intelligence and Video Games. As such, Section 2.1 takes a closer look at the distinctive characteristics of Artificial Intelligence in video games in relation to Artificial Intelligence in general. It identifies the constraints and requirements of an Artificial Intelligence system of a video game.

In contrast, Section 2.2 analyzes a common element found in both fields: Agents. Initially, a description of an agent is provided followed by its two primarily properties: the task environment of agents and the types of agents.

Since implementing a planning system is one of the objectives, Section 2.3 shortly describes the vast area of Automated Planning.

Finally, the previous three sections "meet" at Section 2.4 where a discussion of the first planning technique in video games takes place along with a the applicability in a Real-Time Strategy.

2.1 Artificial Intelligence in Video Games

2.1.1 Game AI

Artificial Intelligence (AI) is the field of science that focuses on developing intelligent systems. In *Artificial Intelligence: A Modern Approach*, Russell and Norving [9] classify AI systems into four primary categories depending on the area of focus: systems that **act** like **humans**, systems that act **rationally**, systems that **think** like humans and systems that think rationally. Figure 1 represents graphically the two dimensions of an AI system: Humanly/Rational and Act/Think.

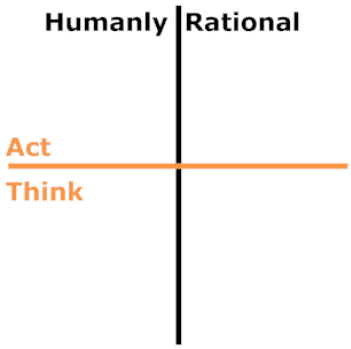

Figure 1 The two dimensions of an Artificial Intelligence System

Systems that act like humans are systems that simulate human behavior while systems that think like humans are systems that focus on simulating the human thought process. Rationality has many definitions; simply put, rationality is "doing the right thing" whatever that may be. Thus, systems that act rationally are systems that behave correctly while systems that think rationally are systems that reason correctly.

Video games include synthetic characters that are known as **Non-Player Characters** (NPCs). An enemy monster, a racing car, an opponent in chess and a pet dog are all examples of NPCs. These characters possess some level of intelligence in order to make decisions. The goal of these decisions is to simulate human behavior rather than behave correctly. A competing car able to behave correctly suggests that it will never crash another car, or get out of the road, or make any other mistakes. Human players cannot play against that level of perfection. Therefore, Game Artificial Intelligence (Game AI) focuses on the development of NPCs that act and think like humans.

From a system's approach, Game AI is a set of techniques used in video games to create the illusion of intelligence [2]. Many of these are based on techniques established by the academic field of Artificial Intelligence, such as planning, neural networks and genetic algorithms. However, in many situations, the demands of such techniques prohibit their use due to a number of constraints imposed by the video game (explained in the next section). Consequently, many video games implement non-AI techniques designed to work under these constraints. Finite State Machines (a technique used to explicitly define the behavior of NPCs) is one such example [10].

Ultimately, the goal of Game AI is to create **challenging** and **entertaining** NPCs for the player [4]. Whether this is achieved using AI techniques or not is irrelevant (from the player's point-of-view).

2.1.2 Challenges of Game AI

The constraints previously mentioned relate to how video games are designed. First, a video game may consist of a number of sub-systems all of which require certain amounts of CPU and Memory and, second, each sub-system is given a small amount of time in which it must complete its execution [11].

A sub-system in a video game, commonly known as an **engine** [12], can be a graphics engine (that renders the graphics to the screen), a physics engine (that calculates forces) and several other engines. Each engine is given a certain amount of CPU and memory with which it must operate. A graphics engine is typically given the majority of these resources, forcing the remaining engines to seek alternative methods to overcome this limitation.

Limited resources are the primary reason why non-AI techniques are preferred [4]. These techniques are designed to require fewer while achieving sufficient results (always from a player's perspective). However, as video games increase in complexity, these techniques fail to scale. Thus, game developers "turn" to the academic field of artificial intelligence in search of more scalable solutions [2].

Aside from the limited CPU and Memory resources, in many situations, the AI system for a video game, hereafter called **Game AI system**, is also restricted in the amount of time it can process. At its very core, a video game is an infinite loop of three phases: **rendering the scene**, **receiving player input** and **updating the game logic**. This is known as the **Game Loop** [12]. Each sub-system of a video game updates in a fixed-rate (depending on the sub-system). All calculations must be performed within each interval otherwise a small decrease in performance may be observed.

By and large, techniques undergo a series of extensive modifications and optimizations to reduce their resource requirements [2].

2.1.3 Additional Requirements

The two previously mentioned constraints are key necessities in any Game AI system. Two additional requirements in such systems are reusability and control.

Many video game companies reuse the source code of a project. This reduces both development time and costs [3]. This is especially the case when the next company's title is a sequel. Furthermore, several companies sell a subset of the source code as an engine to third-parties. Id3, for example, offers the well-known Unreal Engine which is the basis of many titles [12]. There are also companies that specialize in developing game-independent engines, known as **middle-ware** [13]. For example, Havok offers a physics engine by the same name. Hence, it is evident that there is an interest for reusable systems.

Finally, the need for control is imposed by game designers discussed in the next section.

2.1.4 Game Designers

Game designers are responsible for designing a video game's **gameplay** and all elements that produce an experience for the player [14]. Gameplay, as described by Grant Tavinor, is "*[T]he interactive involvement typically associated with videogames, that is, the activities that occur when one plays a videogame*" [15].

One way designers produce the various aspects of gameplay is by using tools created by the game developers [11]. For example, a level editor is a tool that level designers use to create the environment.

A designer can also tweak and define the behaviors of NPCs using tools that can access and modify the Game AI system. This is essential in video games where NPCs need to exhibit specific behaviors enforced by the game's story line.

Therefore, the Game AI system in a video game could provide both autonomous NPCs and tools to control this autonomy wherever needed.

To summarize, a Game AI system must

1. be able to work under limited resources
2. provide NPCs with autonomy as well as the means to control this autonomy
3. be sufficiently reusable for future projects

2.2 Agents

2.2.1 General

An NPC, in the implementation level, is commonly designed as an agent. In Artificial Intelligence: According to Russell and Norvig an agent is "[…] *anything that can be viewed as perceiving its environment though sensors and acting upon that environment through actuators*" [9]. Figure 2 represents graphically this basic architecture.

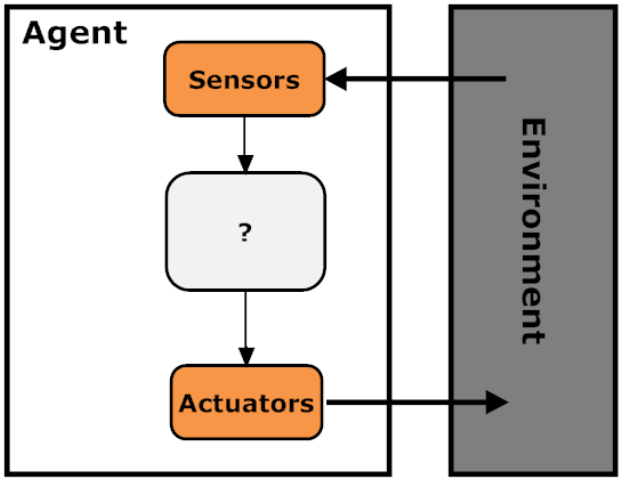

Figure 2 A simplified Agent Architecture

An agent is composed of two elements: the **agent architecture** and the **agent program**. The architecture is essentially the system that provides the sensors as inputs and the actuators as outputs to the agent program. At any given time, the sum of all perceptual inputs (from the sensors) to the agent program is known as the agent's **percept**. Additionally, the sum of all percepts of an agent from its realization to the current state is known as the **percept history**. The output of the agent program is a sequence of actions that depend on the percepts it receives. These actions are executed by the actuators which change the state of the world. A **performance measure** indicates whether the new world state was desired.

Returning to the relation between an NPC and an agent, an NPC perceives the environment (the virtual game world) through sensors (software components that gather information from the virtual world) and acts upon it through actuators (software components that affect the virtual world).

2.2.2 Task Environments

The performance measure, the environment, the actuators and the sensors constitute the **task environment**, which in effect defines the agent architecture. They can be categorized by each of the following properties [9]:

- If the entire state of the environment can be accessed by the agent, then the task environment is **fully observable** otherwise it is **partially observable**.
- A **deterministic** task environment is one that can be determined by the current state and the action performed by the agent. If this is not possible, then it is **stochastic**.
- **Episodic** task environments divide the agent's decision making into episodes. In other words, the process of perceiving and performing an action consists of one episode. In these environments, the selection of an action and its effects depend only on the current percepts of the episode. On the other hand, if the selection of an action affects future selections, then the task environment is **sequential**.
- If the environment changes while the agent performs its decision making, then the task environment is known as **dynamic**, otherwise it is **static**.
- If the state of the environment, the notion of time, the percepts and actions of an agent can be described discretely, then the task environment is **discrete**, otherwise it is **continuous**.
- Finally, a task environment can be **multi-agent** or **single-agent** depending on the number of agents in the environment.

2.2.3 Agent Program Types

Furthermore, there are four basic types of agent programs: the **simple reflex**, the **model-based reflex**, the **goal-based** and the **utility-based** [9].

The simple reflex agent program derives its decisions solely on the basis of its current percepts as opposed to the model-based which uses the percept history. The percept history is used to maintain an internal state of the world which is updated by the sensors. A **world model** provides the knowledge of how to update the world state.

Goal-based agents maintain a set of **goals** which are used to generate a sequence of actions that, when followed, will reach from the current world state to the goal world state. Once the goal has been reach, another is selected and the process starts over. Figure 3 illustrates the architecture for a goal-based agent.

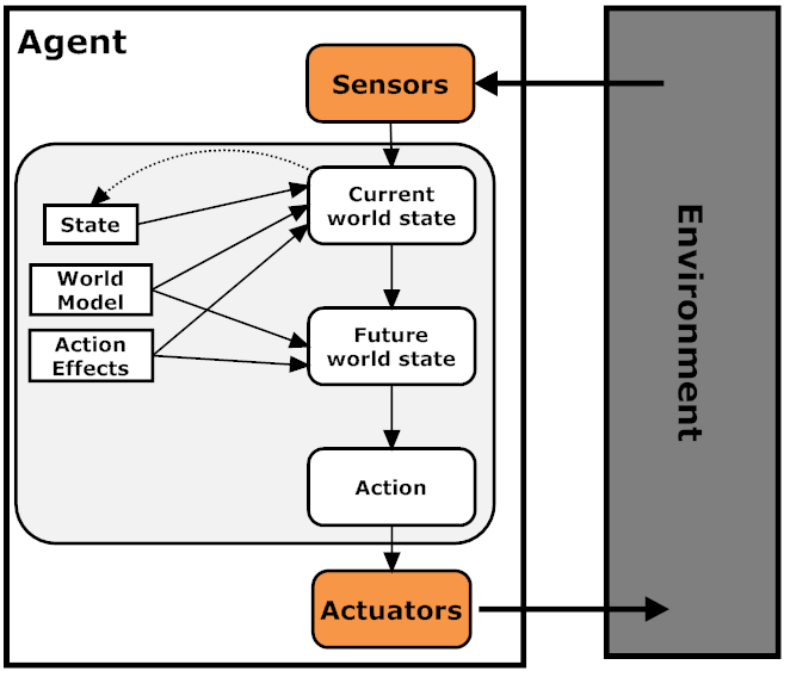

Figure 3 The architecture of a goal-based agent

Finally, a utility-based agent program ranks goal states based on a utility function which affect the decision making process in two ways: it specifies the tradeoffs between two conflicting goals and it sorts the set of goals by importance.

2.3 Automated Planning

2.3.1 General

The generation of a sequence of actions in a goal-based agent can be achieved by an **automated planning** technique. Automated Planning is an artificial intelligence technique that generates a plan that, when followed, solves a problem. This resembles how humans plan and thus falls under the category Think/Humanly [9].

2.3.2 Basic Concepts

A planning problem consists of a **problem definition** and a **planning algorithm**. The problem definition describes the **initial state** of the problem, a **set of operators** and a **goal state**. The algorithm uses the problem definition to construct a sequence of actions that, when followed, will reach the goal state.

A state consists of a set of **facts that are true**. The initial state contains all the facts that are true before planning begins while the goal state contains the set of facts that need to be true for a problem to be considered as solved. The goal state needs to define only the facts that are relevant.

An operator consists of a **set of preconditions** and a **set of post-conditions** (or effects). The set of preconditions are facts that need to be true in a given state in order to apply the operator while the set of effects are facts that become true in a given state after the operator has been applied.

Operators can also contain **variables**. In this case, the operator is called an **action schema** and can only be executed once it has been **instantiated** (all variables have been set to a value).

For example, let us assume that we are currently in our living room and we are hungry. The set of actions that we can perform in this example are to *go in a room*, *cook spaghetti* and *eat it*. Our goal is not to be *hungry*.

We will use the Planning Domain Definition Language (PDDL), a standardized syntax that is used to represent planning problems [4]. PDDL separates the problem definition into two sub-definitions: the **domain definition**, which consists of the set of operators, and the **problem definition**, which consists of the initial state, the goal state, and a set of possible values for the variables.

Code Listing 1 defines the example's domain while Code Listing 2 defines the example problem.

```
(define (domain example)
      (:action cook
            :precondition (and (at kitchen) (meal no_meal))
            :effect (meal spaghetti))

      (:action go
            :parameters (?From ?To)
            :precondition (at ?From)
            :effect (at ?To))

      (:action eat
            :precondition (and (meal spaghetti) (hungry true))
            :effect (and (hungry false) (mean no_meal))))
```

Code Listing 1 Example Domain

```
(define (problem exampleproblem)
      (:domain example)

      (:objects living_room kitchen)

      (:init (hungry true) (at living_room) (meal no_meal))

      (:goal (hungry false)))
```

Code Listing 2 Example Problem

A typical planning algorithm will use these definitions and produce the sequence of actions shown in Code Listing 3.

```
1: go(living_room, kitchen)
2: cook
3: eat
```

Code Listing 3 Example Solution

During planning, the changes made to a world state are only temporal and do not actually change the environment. Changes to the environment are made when a plan has been constructed and given to the agent for execution. We will separate the concept of manipulating world state symbols during plan construction and affecting the environment during plan execution as **applying an operator** and **executing an action** respectively [16].

2.3.3 Planning Process

Code Listing 4 describes a simplified planning algorithm in pseudo-code.

```
create_plan(operators, initial_state, goal)
    current_state = initial_state
    plan = < >
    loop
        if current_state satisfies goal then return plan
        applicable_operators = find all operators that have valid
preconditions
        if applicable_operators = < > return failure
        operator = select an operator from applicable_operators
        apply effects of operator to current_state
        plan = plan + operator
```

Code Listing 4 Simplified planning process

Using our example, the algorithm will start looking for operators that can be applied in the initial state. `Cook` and `Eat` are not applicable because we are not in the kitchen and there is no meal ready. Hence, the algorithm will choose the `Go` operator.

The `Go` operator contains two variables: `From` and `To`. In our problem definition we have stated that there are two objects that can be set to variables. Unless there is a distinction between these objects, the planning algorithm will try instantiating the `Go` operator with every possible combination. However, setting the values to anything other than `From = living_room` and `To = kitchen`, the operator `Go` will not be applicable. Once the algorithm sets the correct values, it will apply its effects.

This will change `(at living_room)` to `(at kitchen)`. The algorithm then will compare the current state with the goal state and realize that it did not reach the goal, so more planning needs to be performed.

Now cook is applicable since we are at the kitchen and we do not have any meal ready. Applying this operator will change the `(meal no_meal)` to `(meal spaghetti)`. Finally, `Eat` is now applicable and applying its effects will set `(hungry true)` to `(hungry false)` and `(meal spaghetti)` to `(meal no_meal)`. Now the planning has ended since `hungry` is set to `false` as stated in our goal.

2.3.4 Searching

Looking at the previous planning process one might wonder why the algorithm does not choose the `Go` operator indefinitely, seeing as it is always applicable. The answer lies to an algorithm known as **searching** [9].

To best describe a searching algorithm, we will first consider a tree-like structure as illustrated in Figure 4.

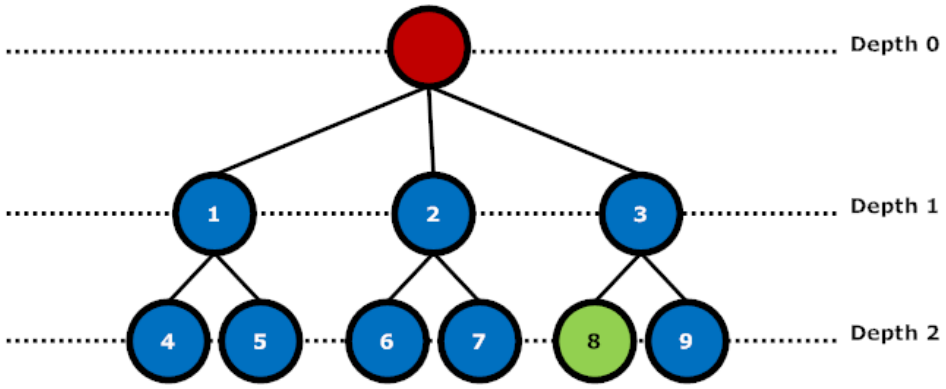

Figure 4 Tree Structure

We will follow Nilsson's graph notation of **directed trees** [17]. An arc between two nodes denotes a parent-child relationship; the upper node is the parent while the lower is the child. A node can have only one parent and any number of children. The top-most node is called the **root** node and has a **depth** of 0. Every other node has a depth of its parent plus one. The number of child nodes of a given node is known as its **branching factor** while the depth of the lowest node denotes the tree's **height**. To reach a certain node one must first "expand" its parent.

Given the red node as a starting point, the goal is to reach the green node. A searching algorithm will traverse the tree in some fashion (depending on the algorithm) and will return the sequence of steps taken to reach the goal, or nothing if the goal was not reached.

A breadth-first search for example, will traverse all nodes in a given depth before expanding their children. In this example, the depth-first search algorithm will return:

$$\langle 1,2,3,4,5,6,7,8 \rangle$$

In state-space planning algorithms, a node represents a world state while an arc represents an instantiated action. In other words, expanding a node means creating a new world state for each applicable operator and applying its effects. The set of all world states and operators is called the **state space**. Figure 5 illustrates the state space for the previous example.

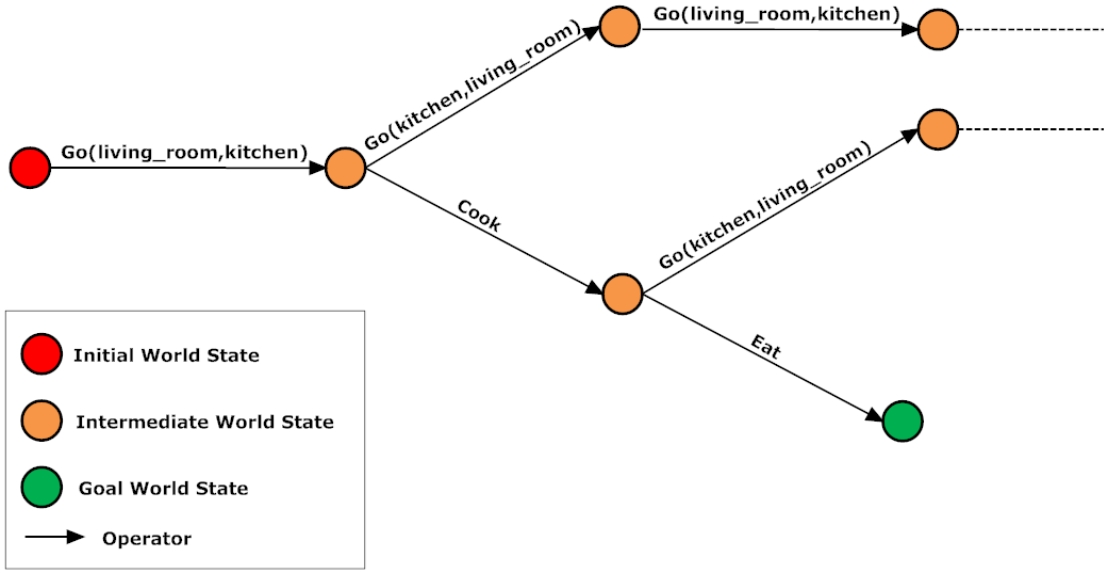

Figure 5 Example State Space

Let us assume that we use the breadth-first search algorithm. In the initial state of the problem, only the `Go` operator is applicable so only one new world state is created and the effects of the operator are applied to it. The new world state is compared with the definition of the goal state; not all facts of the goal state are equal to the new world state (`hungry` is not set to `false`), so more searching needs to be performed.

In this stage, two operators are applicable, `Cook` and `Go`, so two new world states are created, one for each operator. Again, for each new world state, the effects of the respected operators are applied and compared with the goal state. Now, each of the two new world states will create a new one for their applicable operators.

In this case, 3 new world state will be created: two for the world state created by the `Cook` operator (one for the `Eat` operator and one for the `Go` operator) and one for the new world state created by the `Go` operator. The algorithm will first compare the new world state created by the `Eat` operator with the goal state and find that all facts in the goal state are equal to the new world state (`hungry` is now set to `true`). Thus, the plan generated will be:

$$\langle Go(living\_room, kitchen), Cook, Eat \rangle$$

Note that a state-space can be infinite. Many searching algorithms can get stuck in an infinite loop forever expanding nodes. This rapid growth of nodes is called a **combinatorial explosion** [9] and great care must be taken to avoid such a situation.

The breadth-first search algorithm is the only algorithm so far mentioned. Many searching algorithms exist today and they are all divided into two categories: **uninformed** and **informed** searching algorithms [9].

An uninformed algorithm has no information about the problem other than what is the goal state. In other words, it has no means of evaluating whether using an operator will get closer to the goal or not. Breadth-first search is one such algorithm.

Informed search algorithms use what is known as an **evaluation function** in order to evaluate a node. The evaluation function depends on the domain of the problem. Nodes with lower values are preferred over higher ones. Many informed searching algorithms include a **heuristic function** as well, which is an estimation of how far a node is from a given state to the goal state [9].

Depending on the problem, informed and uninformed algorithms produce different results. There is no algorithm that fits all problems. While uninformed algorithms can stick in infinite loops, informed algorithms typically require processing power for the evaluation function.

The most widely-known informed search algorithm is **A* search** (pronounced "A star search") [9]. Each edge in A* has a cost. Given two nodes, the total cost of the path from one node to the other is the sum of all edge costs in that path. For example, if our path from `NodeA` to `NodeD` is $NodeA \dashrightarrow NodeB \dashrightarrow NodeC \dashrightarrow NodeD$ and each edge costs 1, the total cost of the path is 3. Additionally, each node in A* has a heuristic value. The total cost of a node in A* is the sum of the path cost from the initial node plus the heuristic value:

$$f(x) = g(x) + h(x)$$

Where *x* is the node, *g(x)* is the path cost of the node from the initial node and *h(x)* is the heuristic value of the node. **Error! Reference source not found.** describes how A* works in pseudo code.

An example of a heuristic function in planning can be the number of facts that are different between a given state and the goal state. Figure 6 illustrates the state space of A* assuming that each edge costs 1.

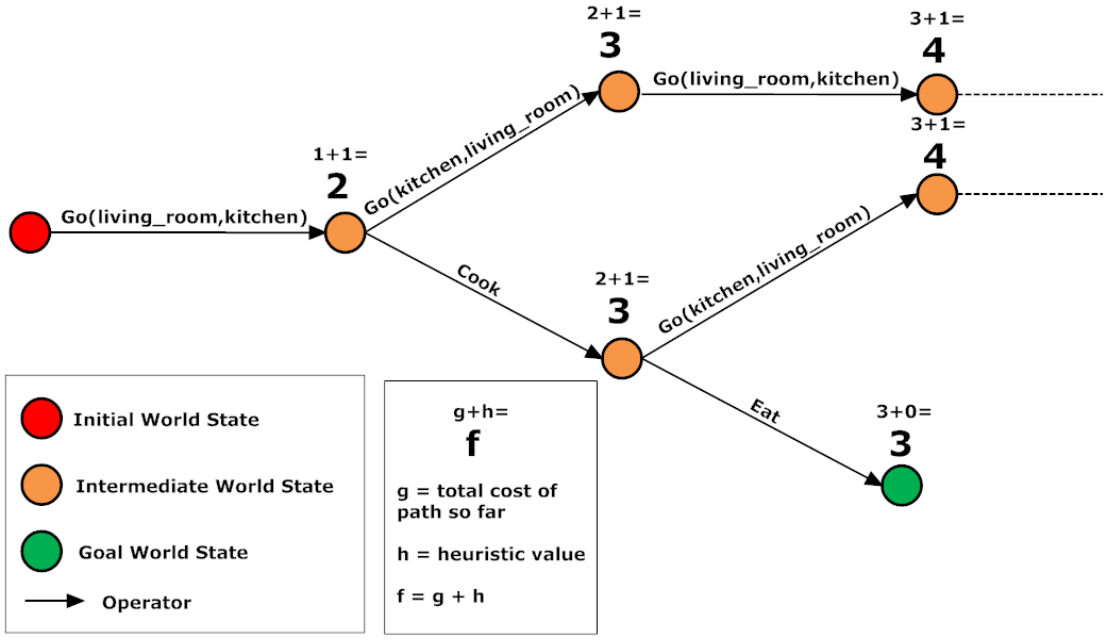

Figure 6 A* State Space Example

```
createPlan(startNode,goal)
     closedset = <>
     openset = <startNode>
     start.g = 0
     start.h = get_heuristic_value(startNode, goal)
     start.f = start.h
     while openset != <>
         node = getCheapestNode()
         if node == goal
             return create_plan(node)
         remove node from openset
         add node to closedset
         neighbors = node.getNeighbors()
         foreach nNode in neighbors
             if nNode in closedset
                 continue

             h = get_heuristic_value(node, nNode)
             g = node.g + h

             if nNode not in openset
                 add nNode to openset
                 isBetter = true
             elseif g < nNode.g
                 isBetter = true
             else
                 isBetter = false
             if isBetter = true
                 nNode.parent = node
                 nNode.g = g
                 nNode.h = get_heuristic_value(nNode, goal)
                 nNode.f = nNode.g + nNode.h
     return failure

create_plan(node)
     if node.parent is not empty
         plan = create_plan(node.parent)
         return (plan + node)
     else
         return node
```

Code Listing 5 The A* algorithm in pseudo code

To further guide the search, we can set different costs to each operator. For example, if `Go` has a cost of 2, while `Cook` and `Eat` have a cost of 1, then the algorithm will prefer `Cook` or `Eat` over `Go` since it is cheaper. Figure 7 shows the total cost of each node in this example.

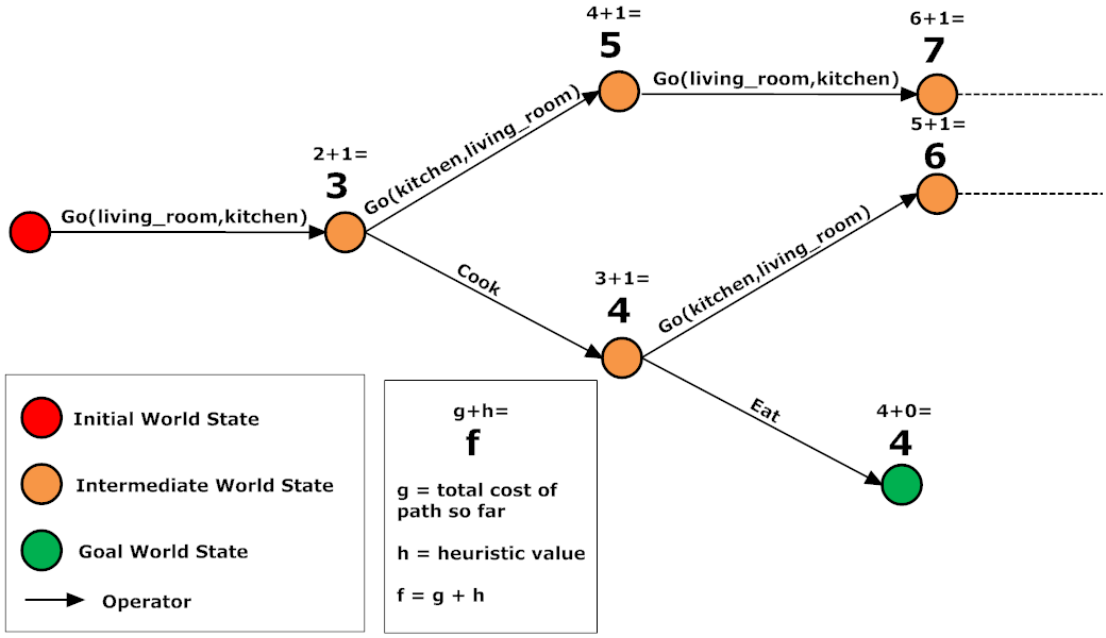

Figure 7 A* State Space Example with varying costs

To return to the question in the beginning of this section, "*why the algorithm does not choose the `Go` operator indefinitely*", a state-search planning algorithm uses search to choose an operator in the set of applicable operators. Of course, as mentioned earlier, depending on the searching algorithm in use, other problems can arise.

2.3.5 Regressive Search

Searching algorithms can also search **regressively** (as opposed to progressively) [9]. A regressive search algorithm starts from the goal state and searches "backwards" until it reaches the initial state. In regressive search planning, operators are applicable if all of their effects are true in the current state and applying an operator will set all its preconditions to the world state.

Whether regressive or progressive search is better depends on the amount of preconditions and effects of each operator. Generally, searching regressively is more direct since operators are chosen based on their effects, thus avoiding irrelevant actions. However, a good heuristic function can equally be as efficient in progressive search.

2.3.6 Partial Ordered Planning

Planning with regressive or progressive search is a form of **totally ordered** plan search. That is, the plan generated is a **linear** sequence of actions from start to finish. Planners that can generate **non-linear** plans are called **partial-ordered planners** [18].

To demonstrate a plan generated by a partial ordered planner, let us look at an example problem. Code Listing 6 and Code Listing 7 define the shoe and sock problem. The goal is to go to school with both shoes on.

```
(define (domain example)
      (:action go_to_school
            :precondition (and (left_shoe on) (right_shoe on))
            :effect (at school))

      (:action right_sock
            :effect (right_sock on))

      (:action left_sock
            :effect (left_sock on))

      (:action right_shoe
            :precondition (right_sock on)
            :effect (right_shoe on))

      (:action left_shoe
            :precondition (left_sock on)
            :effect (left_shoe on)))
```

Code Listing 6 Sock & Shoe Domain

```
(define (problem exampleproblem)
      (:domain example)

      (:init      (left_shoe off)
            (right_shoe off)
            (left_sock off)
            (right_sock off)
            (at home))

      (:goal (at school)))
```

Code Listing 7 Sock & Shoe Problem

The plan generated for the Sock & Shoe problem by a partial-ordered planner is illustrated in Figure 8.

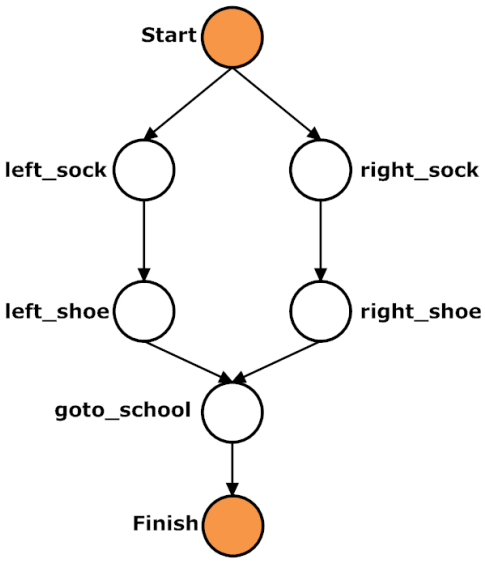

Figure 8 Partial Ordered Plan

Start and Finish are “dummy” actions; Start does not have any preconditions but has as its effects all the facts of the initial state, while Finish has no effects and has as its preconditions all the facts of the goal state.

Note that a partially-ordered plan given to a system that cannot perform actions sequentially can be executed. However, the results will be a linear execution of these actions defeating the purpose of using a partially-ordered planner in the first place.

2.3.7 Plan-Space Searching

Partial-ordered planners can search in plan-space as well as in state-space [9]. In plan-space, planning begins with an empty plan and is refined to a partially-ordered plan step by step. The operators, in this case, are operators that affect the plan instead of the world. These operators can be adding an action to the plan, enforce ordering between two actions etc.

2.3.8 Hierarchical Planning

All previously mentioned planning algorithms consider each action as being relevant to the current problem. In a **hierarchical planner** on the other hand each action can be either a **composite action** (composed by other, lower-level actions) or a **primitive action** (not composed by other actions) creating a hierarchy [9]. Planning is done recursively at each level of the hierarchy by decomposing composite actions and creating a plan. The plan is done when all actions of the plan contains only primitive actions.

Figure 9 illustrates a plan generated by a hierarchical planner for building a house. Note that the plan generated is a partial-ordered plan. This is not always the case; it depends on the hierarchical planning algorithm. In this case, the planner used was a Hierarchical Task Network Planner (HTN Planner).

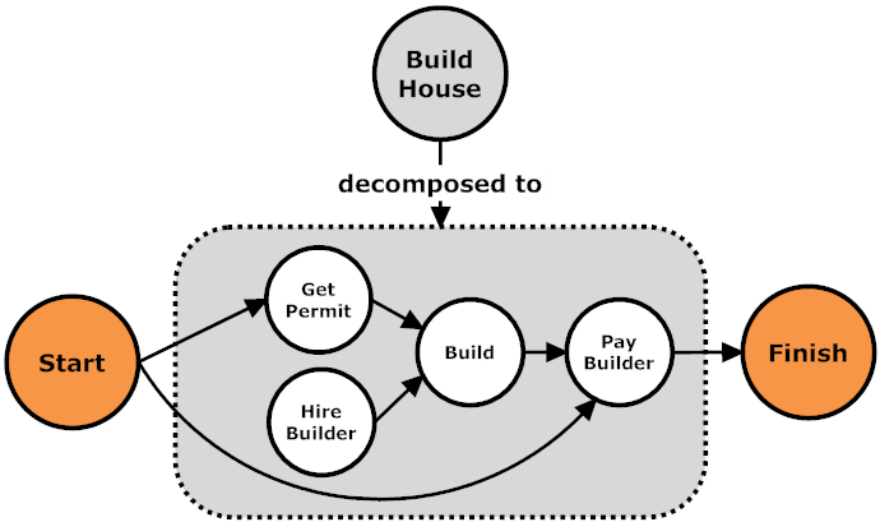

Figure 9 Hierarchical Plan

2.4 Automated Planning and Video Games

2.4.1 Goal-Oriented Action Planning

We know what planning is and how it works. We also know what are the constraints and requirements of a Game AI system. But has planning been applied successfully in a video game?

It has; the first commercial AAA video game to use planning is First Encounter Assault Recon (F.E.A.R.) in 2006. It used Goal-Oriented Action Planning (GOAP), a decision-making architecture that uses a regressive real-time planner developed by Jeff Orkin [19].

The use of planning in F.E.A.R. had a tremendous success [6]. It spawned a Working Group on GOAP with the goal of creating a standard GOAP architecture for video games [5]. Furthermore, many video games since have implemented GOAP including Condemned: Criminal Origins, Silent Hill: Homecoming and Empire: Total War [7].

2.4.2 Benefits of GOAP

In Applying Goal-Oriented Action Planning in Games, Jeff Orkin states that "*A character that formulates his own plan to satisfy his goals exhibits less repetitive, predictable behavior, and can adapt his actions to custom fit his current situation*" [19]. According to Orkin, there are benefits both while the video game is being executed and during its development.

During execution, NPCs generate plans that are relevant at the current world state resulting in a more dynamic behavior. For example, if an NPC's goal is to kill an enemy but there is no weapon in his possession, he will formulate a plan that will include an action to find a weapon. If the same NPC happened at the time to have a weapon, there would be no need to find one, thus the plan generated would not include the action.

However, the greatest benefit of GOAP is during development. While the previous example could have been accomplished in a pre-determined fashion, adding more behavior quickly becomes difficult. In planning, adding more behavior is as simple as adding an action that satisfies a new symbol and adding that symbol as a precondition to another relevant action. There is less need to revisit code to accommodate these changes because the behaviors are encapsulated and, in a degree, isolated from other behaviors.

Furthermore, a pre-determined plan could contain mistakes that would not have been possible with planning. A simplified example would be a pre-determined plan that would have an NPC firing a weapon without having ammo. This mistake can be seen only during execution since during development there is no notion of a valid plan. On the other hand, the preconditions of firing a weapon in planning, forces the NPC to have ammo, otherwise a plan cannot be formulated (or an alternative plan could be found).

Finally, two actions can solve the same symbol. This introduces variety in two levels: the same NPC can behave in two different ways depending on current circumstances and two different NPCs, with different actions that solve the same symbol, will behave differently.

For the reasons outlined in this and the previous section, it was decided that the Game AI solution would use GOAP. The next step is to choose in which video game to integrate it in.

2.4.3 Real-time Strategy Games and Planning

Initially the idea was to develop a planning system and a simple two dimensional game that would be used for demonstration. However, after spending almost a month developing a very basic video game, it was evident that the system would take a lot more time than expected and with no benefits for the dissertation. Thus, it was decided to search for an already made video game that would provide the means to develop a Game AI system.

The first video game that came across was StarCraft, a Real-Time Strategy (RTS) game. A library named Brood War API (explained in Section 3.3) enabled the development of a Game AI system in StarCraft. Further research revealed that RTS is a type of game where planning has many applications [1].

In an RTS the player takes the role of an overseeing General. The goal is to produce an army of units to defeat the opponent. This involves gathering required resources, constructing the army with said resources and strategically planning and performing an attack. The player must be able to maintain a stable economy of resources and make strategic decisions both in the micro-level (tactical decisions) and the macro-level (strategic decisions) [11].

2.4.4 StarCraft

With 11 million copies sold worldwide (as of February 2009 [20]), StarCraft is considered the most popular RTS game today. Developed by Blizzard Entertainment in 1998, StarCraft is still active within the gaming community. So much so that there is an annual televised tournament in South Korea where thousands of fans from all around gather to watch professional teams compete against each other.

In addition to South Korea's annual tournament, this year's Artificial Intelligence and Interactive Digital Entertainment (AIIDE) Conference will include the first StarCraft AI Competition [21].

As if these reasons weren't enough of a motivation, the StarCraft AI Competition encourages submissions that include planning (which amplifies the importance of planning in RTS games). Thus, StarCraft was chosen as the video game in which the Game AI system will be integrated.

3 Analysis

The previous chapter introduced the areas of automated planning, agents and Game AI in general. It was also decided that a planning system will be developed using GOAP in StarCraft.

This chapter looks into GOAP and StarCraft to gather knowledge about how each work in order to develop the solution.

3.1 Goal-Oriented Action Planning

3.1.1 General

Goal-Oriented Action Planning defines both a goal based agent architecture and a regressive real-time planner [22]. The agent architecture is based on MIT Media Lab's C4 while the planner is based on STRIPS. The agent architecture has been modified to include planning instead of "Action Tupples" for the decision-making mechanism while DELETE and ADD lists of STRIPS have been replaced with world state properties.

3.1.2 World State Representation

Goals, actions and the planner all use a world state, each for different reasons. A world state in GOAP is a set of world state properties that are symbolically represented. Each property has a key that uniquely identifies it and a value that represents the state of the property. In F.E.A.R. the value can be a number of different data types (integer, boolean etc) [23].

Each NPC preserves a world state that represents the internal state of the world. Figure 10 illustrates an excerpt of an NPC's world state.

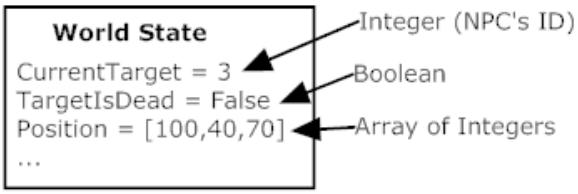

Figure 10 An excerpt of an NPC's world state

3.1.3 Goals and Actions

A goal is essentially a world state that contains a subset of all world state properties. The values of each property in a goal indicate its satisfaction state. Figure 11 contains an example goal.

Figure 11 An NPC goal

Similarly, the preconditions and effects of an action are each a world state that contain a subset of the NPC's world state properties. Each value in the preconditions indicate the state that must be true in the NPC's world state while each value in the effects indicate the value that will be set once the action has been applied. Figure 12 illustrates two example actions.

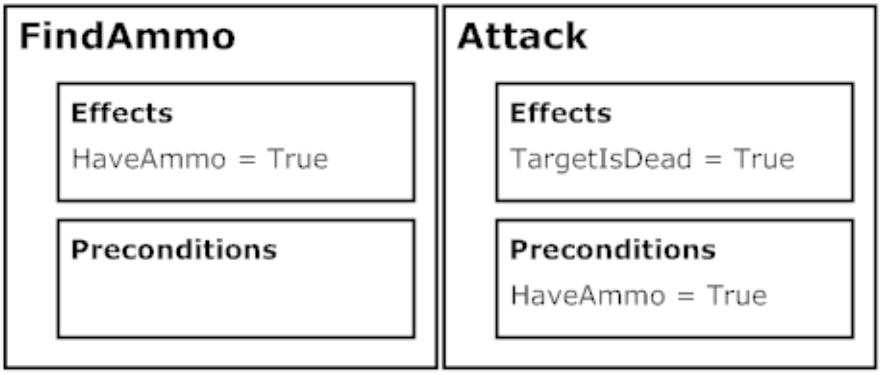

Figure 12 Two example actions of an NPC

According to Orkin [22], preconditions and effects have a direct influence to the actions that will be chosen by the planner: if a precondition is not satisfied, an action must be found that will satisfy it (otherwise a plan cannot be formulated). However, there are many situations when an effect or precondition cannot solve or be solved directly by an action. For these situations, actions specify **Context Effects** and **Context Preconditions** respectively. An example of such a situation is when a visibility test is needed: if a point in space is not visible to the NPC there is no action that will make it visible [22]. Thus, an action that involves an NPC shooting at a target can have a context precondition of the target being visible. Context preconditions and effects allow code to be executed at planning time in order to make additional calculations.

In addition, actions in GOAP implement a method that indicates whether the action is valid. This is used during plan execution. If the method returns false, the plan has been invalidated and thus a new plan must be formulated.

Finally, along with maintaining a local view of the world state, NPCs also maintain a set of goals and actions that are relevant to them. All actions and goals are created by developers but the assignment to a particular type of NPC can be done by the designers.

3.1.4 Planning Process

Before explaining the planning process of GOAP, it is important to understand that, during planning, the planner maintains two world states: the current world state and the goal world state. Any modifications of these states do **not** affect the environment nor do they affect the agent's world state. When the planning process initiates, the states are empty.

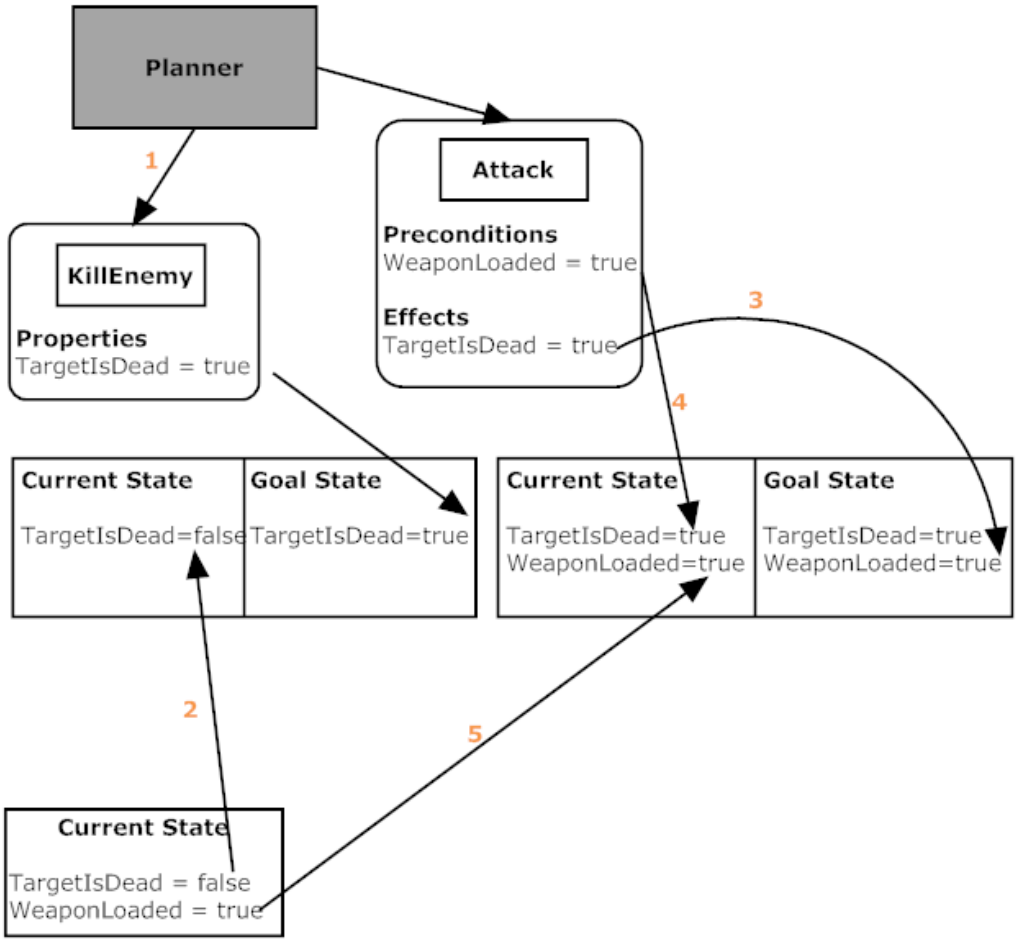

Figure 13 GOAP Planning Process

Let us now describe the planning process using an example (illustrated in Figure 13). Let us assume that our current goal is KillEnemy. The goal contains only one world state property: TargetIsDead = True. The planner will copy all properties that exist in the goal state but not in the current state to the current state, which in this case, it is only TargetIsDead (1). However, the value of this property in the current state is not set to the same value. Instead, the planner applies the same value with the value of the agent's **actual** world state property. In our example, this value is set to False (2). The planner would now look at the current and goal state and observe that not all properties are equal. Thus more planning must be carried out. Note that if the agent's actual world state property value of TargetIsDead was True, the planning would have finished.

The next step is to choose an appropriate action that will solve the TargetIsDead symbol (in other words, set it to True). To choose, the planner will look at the effects of each action and find the property TargetIsDead = True. One such action is AttackEnemy, which has only one precondition: WeaponIsLoaded = True. The planner copies all effects of the action to the goal state, thus, setting TargetIsDead = True (3), and all preconditions to the current state (4). Again, all the properties that are in the goal state but not in the current state are copied to the current state (5). WeaponIsLoaded was set to True in our agent's world state and thus all properties of the current and the goal state are equal. This means that the planning has finished and the plan is a single action: AttackEnemy.

Finally, the searching algorithm used for selecting an action is A*. As such, actions have a heuristic value and a cost. The heuristic value between two nodes is the number of the world state properties that have different values.

3.1.5 Agent Architecture

As mentioned earlier, the agent architecture of GOAP is based on MIT Media Lab's C4. C4 is composed of six components: a blackboard, a working memory store, a sensor system, a decision-making system, a navigation system and a motor system [24]. GOAP replaced the decision-making system with a planner and, for F.E.A.R., a number of other systems where included along site the navigation and motor systems.

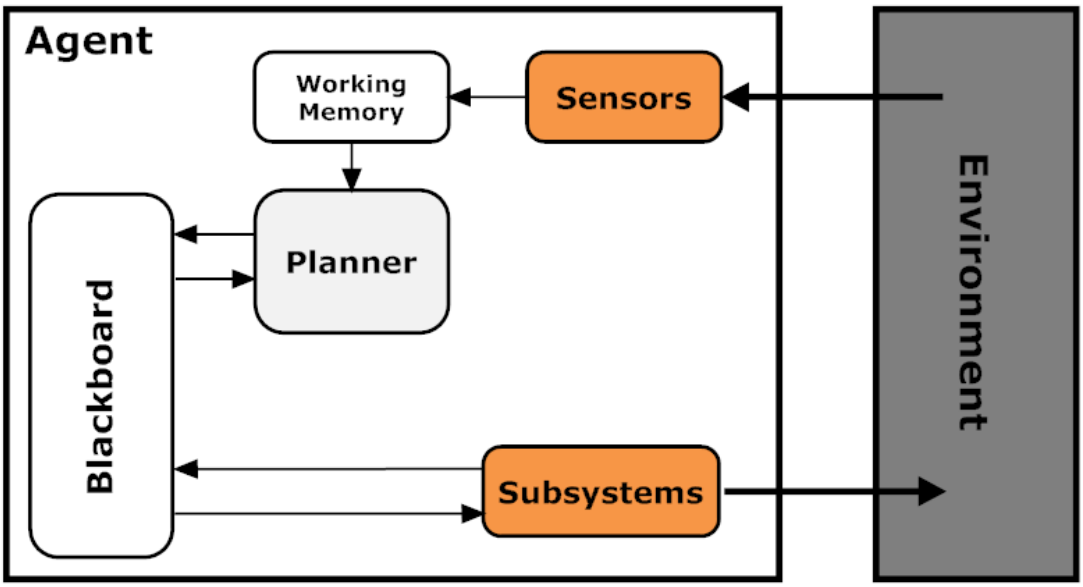

Figure 14 GOAP Agent Architecture

Figure 14 illustrates the general agent architecture of GOAP. **Sensors** store **facts** retrieved by the environment to the **working memory store**. The facts are then use by the planner to drive the planning process. Once a plan has been found, the planner stores **requests** to the **blackboard**. The requests are retrieved by the relevant **subsystems** to perform the actions that will affect the environment [22].

A fact is basically a data type that represents an aspect of the current world much like a world state property. However, unlike a world state property, a fact is not a single value; it is a sequence of values that represent the change over time. Sensors constantly place facts about a property of the world. Thus, the planner can observe a change in a property which, depending on the change, can trigger a formulation of a plan.

For example, a sensor can place facts about the distance of an enemy character. A planner can observe the change in distance and, if that distance is smaller than a certain amount, trigger the formulation of an attack plan.

It is worth noting that subsystems do not directly communicate with each other. Just like a natural blackboard, in GOAP, subsystems place requests that need to be executed by a relevant subsystem on the blackboard. Each subsystem looks up the blackboard for any requests that can be performed by themselves and, if one is found, carries it out. This decoupling of subsystems provides a very flexible system where any number of subsystems can be added or removed without the need of modifying existing ones.

In our previous example of the distance of an enemy, when the planner generates the plan, each step of the plan (each action) places a request to the blackboard. The request is carried out by a subsystem and the plan proceeds to the next step. If a subsystem can perform only part of the request while another subsystem can perform the other part, the communication between the two is done again using the blackboard.

Depending on the video game, a subsystem can be a navigation system which generates a path in which the agent will follow, a weapons system, used in F.E.A.R. or any other type of system that needs to be carried by the agent.

3.2 StarCraft

3.2.1 General

StarCraft includes three distinct races from which the player can choose: the **Zerg**, the **Terran** and the **Protoss**. None of the races are superior to the other, which is one of the reasons it is still played by many today. Generally speaking, the units of the Zerg race are cheap to produce but weak while the units of the Protoss are expensive and powerful. Terran units are in between.

3.2.2 Units and Buildings

There are four types of units which the player can create: **peons**, **fighting units**, **support units** and **buildings**. All four require a certain amount of resources in order to be created. Peons are primary resource gatherers. While they have the ability to attack, they are very weak. Fighting units, on the other hand, have no gathering capabilities and are used to attack an opponent. Units that cannot attack and gather are known as support units which provide special abilities to the player.

Peons, fighting and support units are all produced from a building. Some types of buildings produce specific types of units while others provide upgrades instead. Upgrades can provide special abilities to a type of unit or the player or increase its characteristics (attack and defense). All buildings, excluding the base buildings, depend on the existence of a specific building type. The base buildings are the buildings that accumulate the resources. The dependencies can be visually represented by a hierarchical tree commonly known as the **technology tree**, or tech tree. Figure 15 represents the tech tree of the Terran race. Add-ons are special buildings available only to the Terran race which must be attached on a specific type of building in order to provide upgrades.

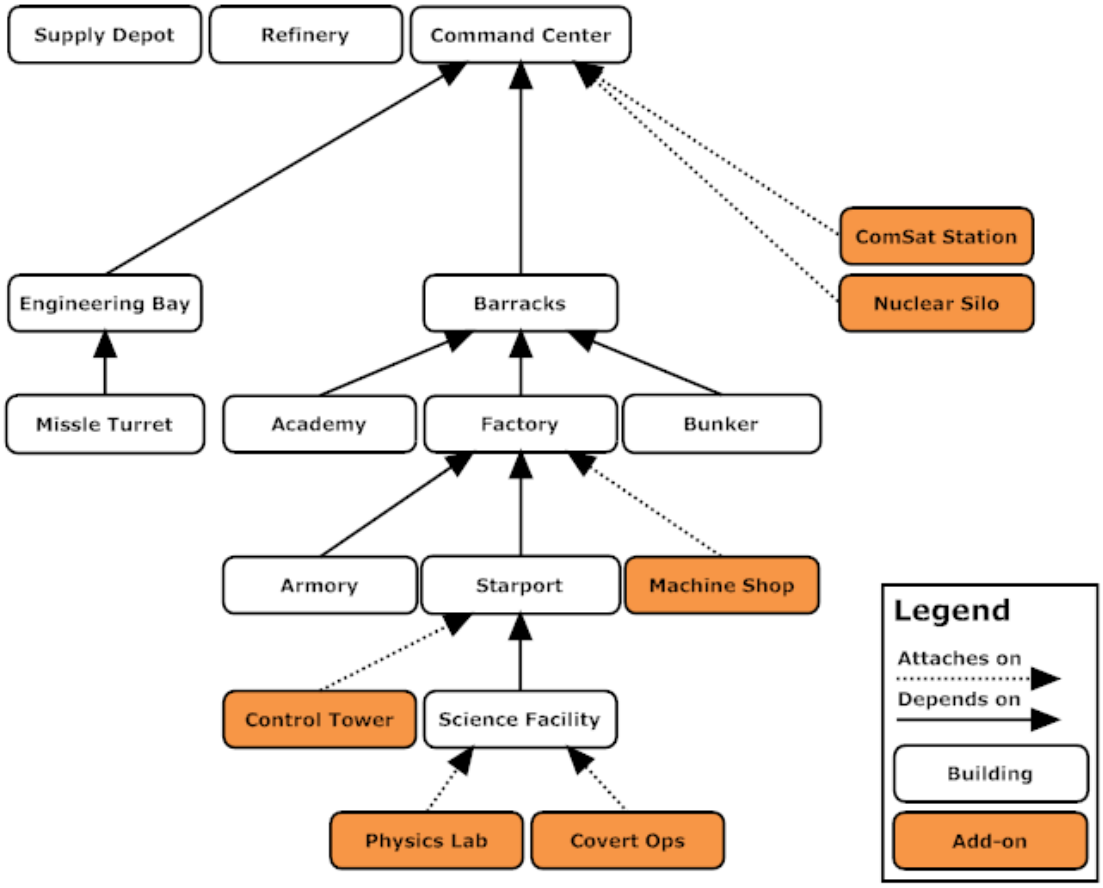

Figure 15 The Terran Tech Tree

3.2.3 Resources

There are three types of resources: **minerals**, **gas** and **supply**. Minerals and gas are gathered by peons and form the economy of the player. They are used to create both the buildings and the units. Supply is the number of units (not buildings) a player can currently posses; the maximum allowed is 200. Each type of unit requires a certain number of supply in order to be created. The player starts with an initial supply of 8 and creates supply buildings to increase the number in intervals of 10.

3.3 Developing a bot in StarCraft

3.3.1 Brood War API

While there is no access to the source code of StarCraft as is possible in some open-source RTS games such as Wargus, there is a solution that exposes an Application Programming Interface (API) for StarCraft: Brood Wars called **Brood War API** (BWAPI). The API is exposed from the **Chaos Launcher** application which launches StarCraft and injects BWAPI into the game's process.

BWAPI is written in C++ and the API is provided by extending the class AIModule. Currently the version is 2.7.2 Beta. A number of libraries have been developed in order to bring BWAPI in other programming languages as well. Java, Haskell, PHP, Ruby, Python, Lua and C# are supported with various libraries as of this writing [25].

These libraries come in two "flavors": a "wrapper" library and a proxy library. The proxy library extends the AIModule and opens a socket that can be used to communicate with the StarCraft application process while the wrapper library simply extends the AIModule in a different programming language. Figure 16 illustrates this process.

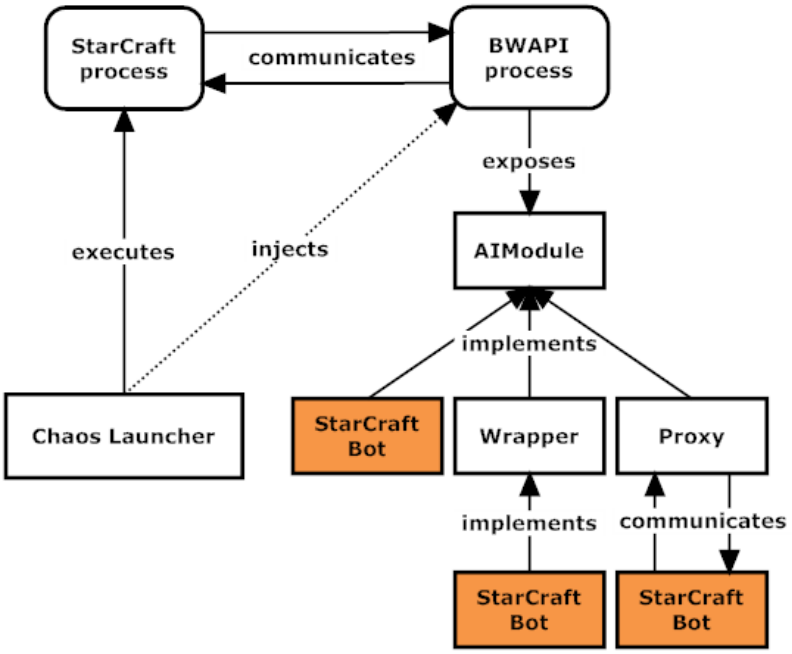

Figure 16 How BWAPI works

3.3.2 BWAPI for Java

Due to limited knowledge of C++ and timing constraints, the implementation of the bot was developed in Java. For Java, there are two solutions: a wrapper named **JBridge** and a proxy named **BWAPI-Proxy**.

Neither solution provides an implementation of the latest version of BWAPI. JBridge is currently compatible with BWAPI 2.4 while BWAPI-Proxy with 2.3.1. However, for the purposes of demonstrating planning, version 2.4 is sufficient.

JBridge was selected for two reasons: it provides a more recent implementation of the API and, more importantly, it provides a superior library of classes. The only drawback compared to BWAPI-Proxy is the inability to provide real-time debugging. Being a socket implementation, with BWAPI-Proxy the bot could be executed before the StarCraft process, thus one would start the bot in a Java console, launch StarCraft and observe any error that caused the bot to terminate abruptly. With JBridge, if the bot could not be executed when the StarCraft process initiates, there is no easy way of discovering the cause. However, this problem occurred only during initiation; during runtime, exceptions could be caught normally.

3.3.3 JBridge

The current version of JBridge is 0.2 Alpha which, as stated earlier, is compatible with BWAPI 2.4. Appendix A describes the process of setting up the environment. To create a bot using JBridge one must implement two interfaces: **BridgeBotFactory** and **BridgeBot**.

BridgeBot is the interface that contains all calls coming from StarCraft. The most important method is *update* which is called in every update of the game.

BridgeBotFactory defines a single method named getBridgeBot. This method takes a Game as an input and returns a BridgeBot. When JBridge is initiated by Chaos Launcher, it looks into the AI folder of StarCraft for any classes that implement BridgeBotFactory (only the first found is loaded). Once a class is found, the method getBridgeBot is called and the bot returned is loaded into the game.

Access to the game's information is provided by the class Game. This class is a singleton (only one instance exists) and can be accessed from anywhere.

Finally, JBridge comes by default with the Brood War Terrain Analyzer (BWTA). This library scans the entire maps and produces three sets: a set of **base locations**, a set of **choke points**, and a set of **regions**. A base location is a location where minerals and gas are nearby (suitable for constructing a base). Choke points are narrow paths, such as bridges that connect two regions. Regions are partitions of the map. The knowledge provided by BWTA is of high strategic value.

4 Design & Implementation

4.1 Solution Overview

4.1.1 GOAP + StarCraft = StarPlanner

Players in any RTS do not control a single character; they control the entire population of their society by giving each unit orders. This was also the approach taken for the development of the Game AI system for StarCraft. The system, named **StarPlanner**, is essentially a single agent that makes high-level and mid-level decisions. At the highest level, StarPlanner makes strategic decisions such as attacking an opponent or creating a secondary base. At the mid-level it decides what units and buildings to create.

These decisions generate a plan where each action is a set of commands that are given to available units or buildings. However, to simplify an action's definition and to enable code reuse, a series of low-level managers were created. Instead of commanding units directly, an action "posts" a request to the blackboard which will be carried out by the appropriate manager. This architecture enabled a number of useful features as well as the separation of concerns. We can think of the managers as the agent's actuators.

4.1.2 Methodology

It was decided to take a bottom-up approach: initially, develop a general-purpose A* engine, then a general-purpose GOAP architecture and, finally, a StarCraft AI system that will use the GOAP architecture. This decision was made based on the fact that one of the requirements of a Game AI system is to be reusable. Both the A* engine and the GOAP architecture can be reused. StarPlanner is a proof of this concept. Thus, the software process model used was the **Incremental Process Model**.

4.2 1st Iteration: A* Engine

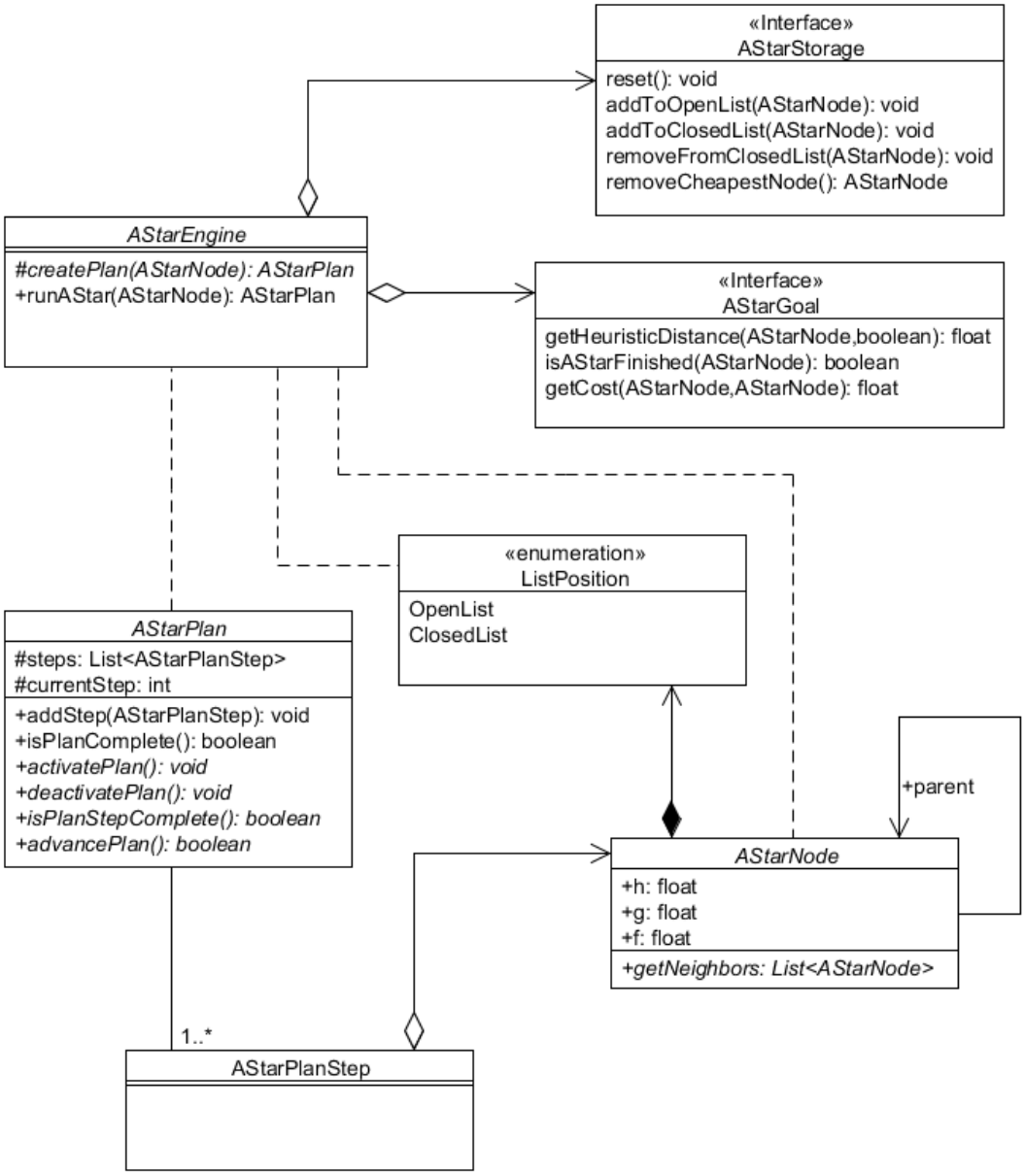

Figure 17 A* Engine UML class diagram

Figure 17 illustrates the UML diagram that contains all classes relevant to the A* engine. The "heart" of the A* engine is the **AStarEngine** class and, in particular, the method runAStar (presented in Code Listing 8).

```
public AStarPlan runAStar(AStarNode endNode) {
	storage.reset();
	storage.addToOpenList(endNode);

	AStarNode currentNode = endNode;
	float h = goal.getHeuristicDistance(currentNode, true);
	currentNode.g = 0.0f;
	currentNode.h = h;
	currentNode.f = h;

	AStarNode tmpNode;
	while (true) {
		tmpNode = storage.removeCheapestOpenNode();

		// If there are no other nodes, break
		if (tmpNode == null)
			break;

		currentNode = tmpNode;
		storage.addToClosedList(currentNode);

		// If search is done, break
		if (goal.isAStarFinished(currentNode))
			break;

		List<AStarNode> neighbors = currentNode.getNeighbors();
		for (AStarNode neighbor : neighbors) {
			if (currentNode.parent == neighbor)
				continue;

			float g = currentNode.g
					+ (goal.getCost(currentNode, neighbor));
			h = goal.getHeuristicDistance(neighbor, false);
			float f = g + h;

			if ( f >= neighbor.f )
				continue;

			neighbor.f = f;
			neighbor.g = g;
			neighbor.h = h;

			if (neighbor.position == ListPosition.Closed)
				storage.removeFromClosedList(neighbor);

			storage.addToOpenList(neighbor);
			neighbor.parent = currentNode;
		}
	}

	return createPlan(currentNode);
}
```

Code Listing 8 A* create plan method

To provide extensibility, almost every class in the A* engine is either an abstract class or an interface. To create a concrete A* implementation one must extend the classes AStarEngine, **AStarNode**, **AStarGoal** and **AStarPlan**.

AStarNode is an abstract class that represents a node in the search space. It contains one abstract method named getNeighbors that must return a list of the node's children (depending on the implementation).

AStarGoal is used during A* search to perform various calculations. First, it contains methods that return the heuristic value and the cost between two nodes. And second, it contains a method that, given a node, it returns whether the goal has been reached.

AStarEngine has one abstract method named createPlan which takes the last node added to the sequence of actions and returns an AStarPlan.

AStarPlan maintains a list of AStarPlanStep that represent the sequence of actions created by the searching algorithm as well as the index that denotes the current active step. The methods provide various operations such as moving to the next step, checking whether the plan is finished etc.

AStarPlanStep was created to provide the separation between a node and a step in the plan. This was used primarily for the GOAP which will be explained in Section 4.3.

Key to debugging an application is the ability to keep a log of what's happening "under the hood". For this reason a simple logging class was created in this iteration. Almost all classes of the entire application use the Logger for printing debug information.

4.3 2nd Iteration: Planning Engine

We will begin by examining how a world state is represented in code. Figure 18 contains the class diagram of classes related to the world state implementation.

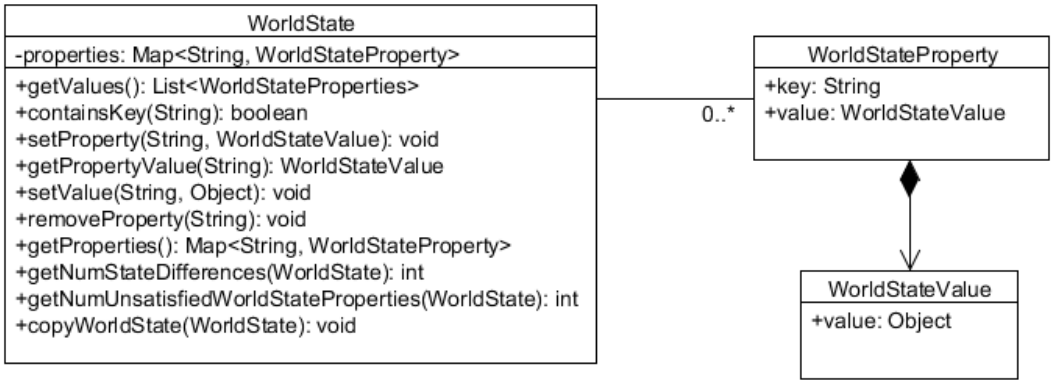

Figure 18 World State UML class diagram

As the class diagram suggests, the world state is represented by a class named **WorldState**. The class maintains a key/value Map of String/WorldStateProperty as well as a plethora of methods that manipulate the list. The String represents the world state property name while the **WorldStateProperty** is a class that contains the value. The value is of type **WorldStateValue** which is a Generic class. This way, the user explicitly defines what type the value holds. This convenience can be seen in Code Listing 9.

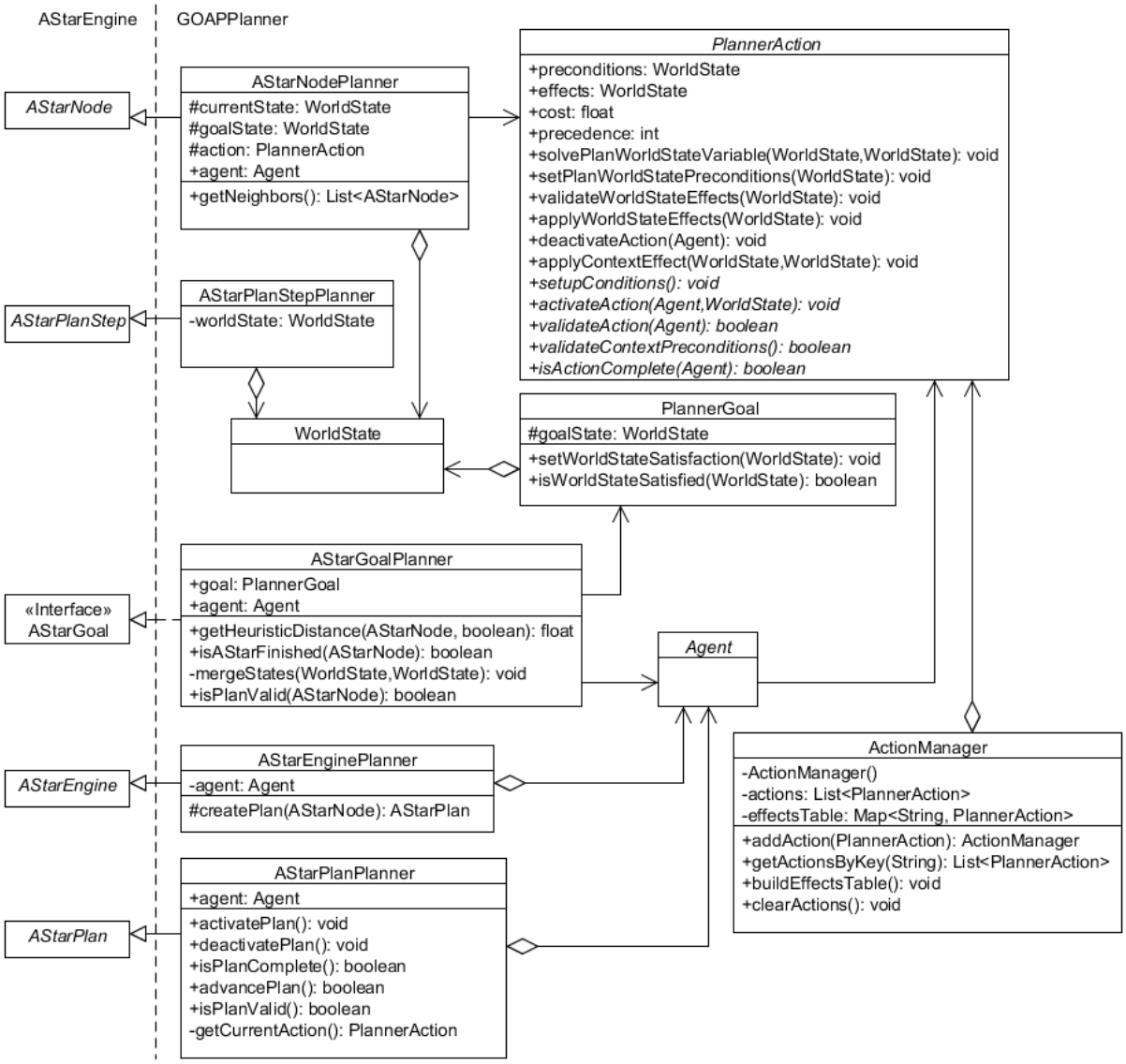

Figure 19 Planning Engine UML class diagram

Figure 19 illustrates the UML diagram that contains all classes relevant to the planning engine. To simplify the diagram, classes that are relevant to the A* engine and the agent architecture display only their names. The **WorldState** will be explained later.

As can be seen in the diagram, the planning engine extends the A* engine by implementing the classes mentioned in the previous section. The naming convention followed was the name of the A* class + "Planner" (i.e. AStarEnginePlanner).

The most important classes of the planning engine are **PlannerAction** and **PlannerGoal**, which represent an action and a goal respectively, **AStarPlanStepPlanner** and **ActionManager**.

AStarPlanStepPlanner represents an action in the sequence of actions that constitutes the plan. The step simply contains the associated action that will be used during plan execution.

To create an action one must extend the PlannerAction class and implement the methods **setupConditions**, **activateAction**, **validateAction** and **isActionComplete**. activateAction is called once when the plan advances to the action. Once the step is activated, the planner constantly calls validateAction and isActionComplete. The plan will be invalidated when the former returns false, while the plan will advance when the latter returns true.

The preconditions, effects and action cost of an action are all set in the method setupConditions. Code Listing 9 provides an example implementation. In addition, an action may extend the method **validateContextPreconditions** and **applyContextEffects** if the action contains Context Preconditions and Effects. If the user does not extend these methods, the former will return true by default while the latter will do nothing.

```
@Override
public void setupConditions() {
    this.precedence = 1;
    this.cost = 2;

    this.preconditions = new WorldState();
    this.preconditions.setProperty("IsWeaponLoaded",
            new WorldStateValue<Boolean>(true));

    this.effects = new WorldState();
    this.effects.setProperty("EnemyIsDead",
            new WorldStateValue<Boolean>(true));
}
```

Code Listing 9 Example setupConditions implementation

To define a goal one must simply set up the world state properties in a PlannerGoal object as seen Code Listing 10.

```
public class ExampleGoal extends PlannerGoal {
    public ExampleGoal() {
        this.goalState.setProperty("EnemyIsDead",
            new WorldStateValue<Boolean>(true));
    }
}
```

Code Listing 10 Example goal class

To choose an action, the planning system has a class named **ActionManager** which maintains a key/value Map of property/actions. In other words, each world state property in the map contains a list of actions that have the same property in their effects. Actually, ActionManager is an enumeration type, denoting that it is a Singleton class.

The planning engine uses ActionManager to retrieve the list of actions that can solve a given world state property. As such, the ActionManager *must* contain all actions of the system *before* any planning takes place. To facilitate easy addition of actions, the method **addAction** returns the instance of the ActionManager. Code Listing 11 contains an example of adding multiple actions.

Code Listing 11 Example of adding actions in ActionManager

4.4 3rd Iteration: Agent Architecture

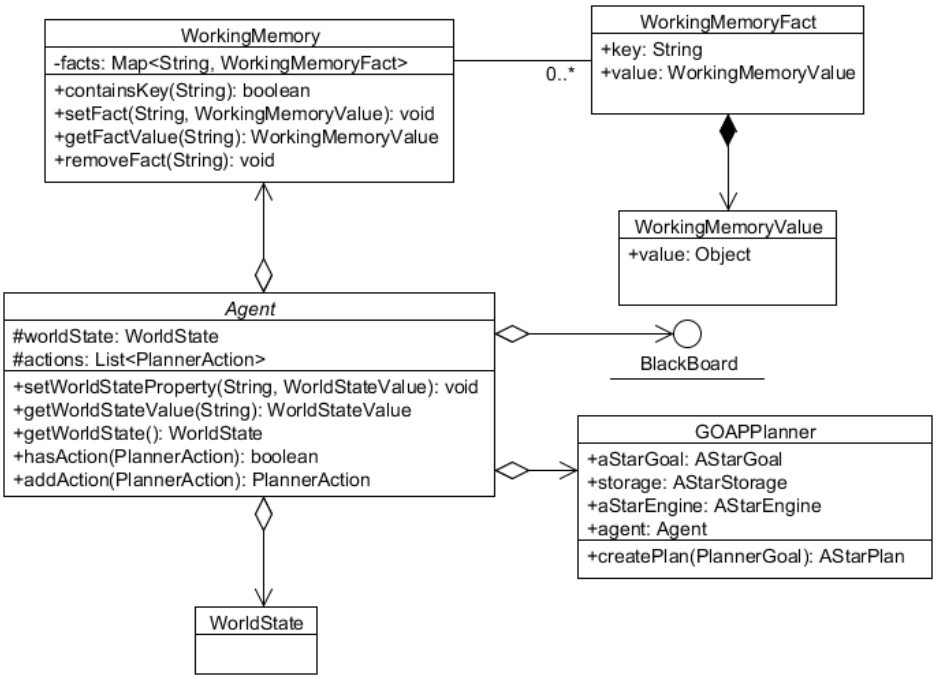

Figure 20 Agent Architecture UML class diagram

Probably the most difficult aspect of creating the agent architecture is thinking about what needs to exist and what not. The integration of the system in a game depends heavily on the game's requirements. However, it was decided to create classes that do not add functionality but instead give hints as to what the integration could have. For example, the agent *should* have a blackboard, and thus contains a field of type **BlackBoard**. However, a blackboard contains information that is relevant to the game. For this reason, the BlackBoard is an empty interface that has no methods.

Another implementation hint is the fact that the **Agent** class is an abstract class with no abstract methods. This denotes that this class **must** be extended because by itself, an agent does nothing.

WorkingMemory, **WorkingMemoryFact** and **WorkingMemoryValue** are implemented similar to WorldState, WorldStateProperty and WorldStateValue respectively. The difference is that WorkingMemory represents the agent's working memory.

4.5 4rth Iteration: StarCraft Integration

4.5.1 Overview

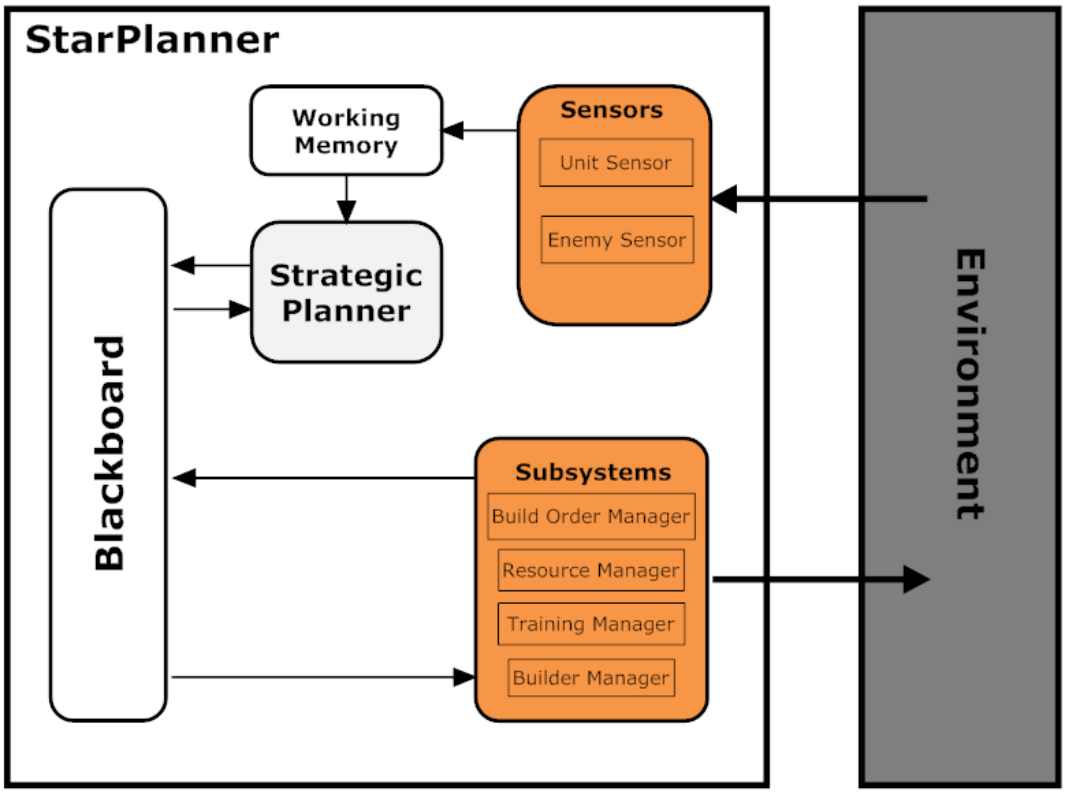

Figure 21 StarPlanner Architecture

Figure 21 illustrates the overall architecture of StarPlanner. Note that the Build Order Manager is an actuator (a subsystem of the agent) even though it uses planning. The following sections describe each component.

4.5.2 JStarPlanner & MainGUI

MainGUI is the GUI in which the user selects whether to run StarPlanner with StarCraft or without. Running StarPlanner without StarCraft provides the ability to test the planning process. Either way, JStarPlanner is displayed.

JStarPlanner is a GUI that contains two tabs, the Main tab and the StarPlanner Manager tab.

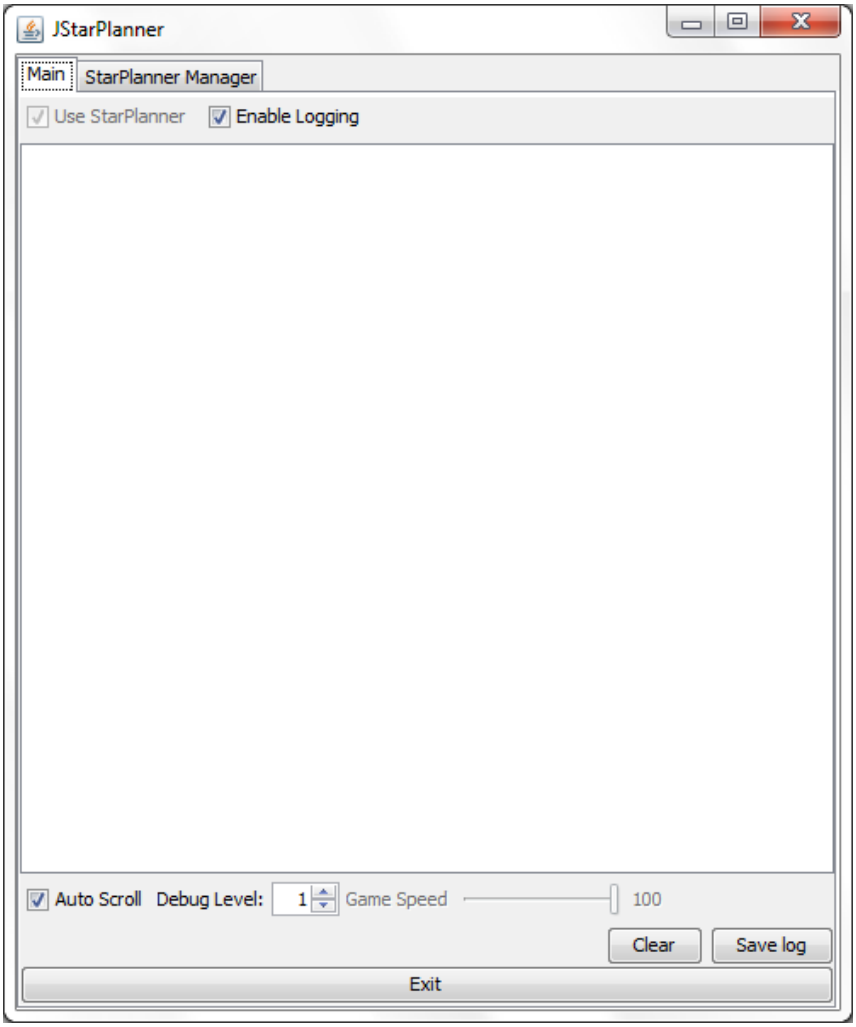

Figure 22 The Main tab of JStarPlanner

The main tab, shown in Figure 22, contains various settings in regards to StarCraft and logging. The user can stop displaying debug information, save this information into a file, clear all information, set text scrolling and set the speed at which StarCraft runs. Most importantly though, the user can enable or disable StarPlanner at run-time. Because StarPlanner controls all units in the game, if a user were to give an order to a worker unit, StarPlanner will "override" that order and set it back to gather minerals. Naturally, this option is unavailable when JStarPlanner is run outside of StarCraft.

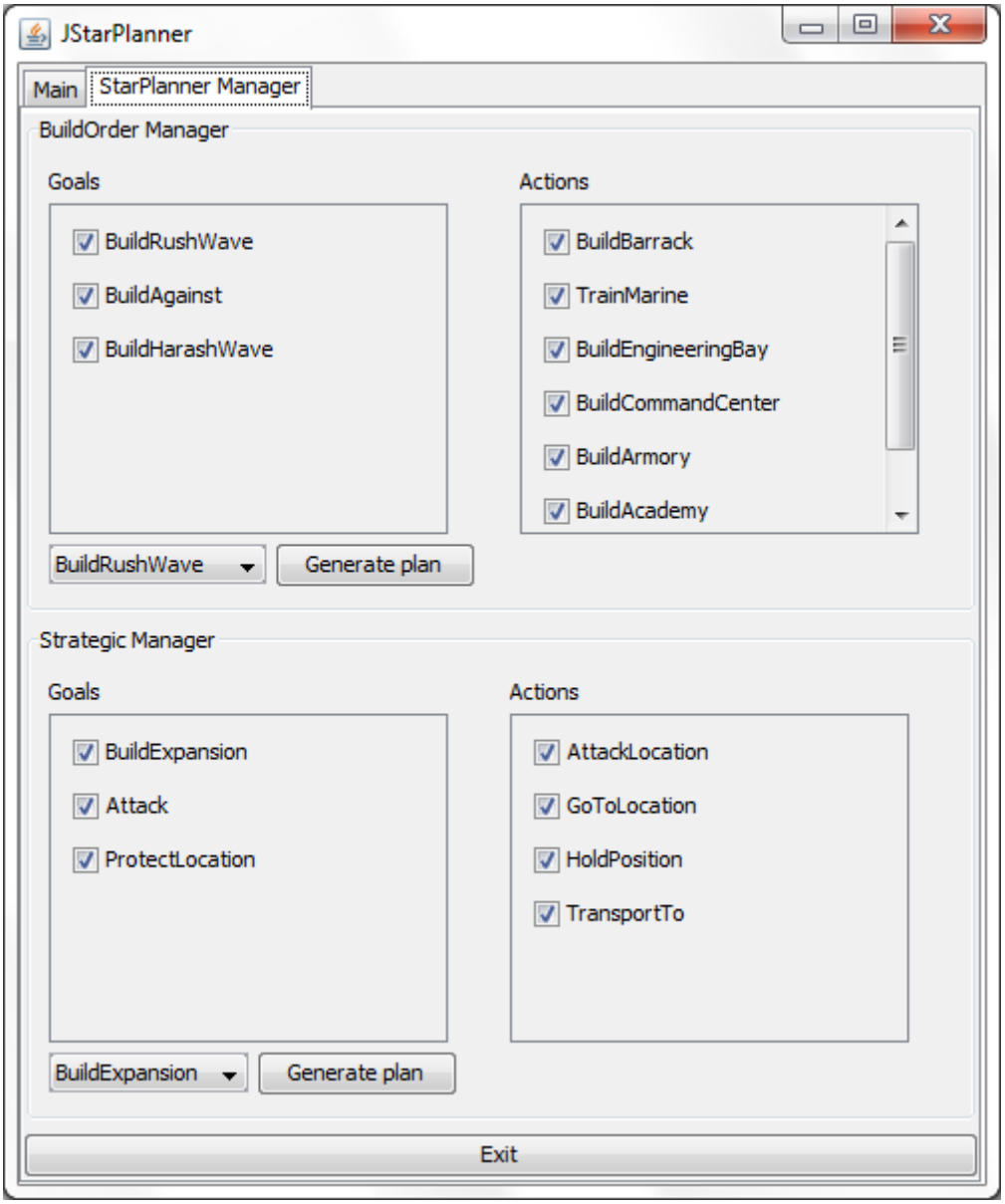

Figure 23 The StarPlanner Manager tab of JStarPlanner

The StarPlanner Manager tab contains the set of goals and actions as well as the ability to generate a plan. A user can enable or disable each goal or action to set the system's behavior. The generation of plans was primarily used for testing. Enabling or disabling an action can sometimes render a plan invalid and thus great care must be taken before using it in StarCraft.

4.5.3 Blackboard & Working Memory

The blackboard is a simple data type that contains public fields used by the different managers. It contains fields such as a build request queue, used by the Building

Manager, and a training request queue, used by the Training Manager. It also contains fields that are used both by the Strategic Manager and the Build Order Manager.

The working memory contains facts about the number of units currently in the game. This is used by a number of components. Sensors add information gathered from the world while actuators and managers retrieve this information to make various calculations.

4.5.4 Planning Managers

Build Order Manager

The build order manager is responsible for creating units. It is initiated whenever a goal is "posted" on the blackboard by the strategic manager after which, it creates a plan that will reach the goal. If a plan cannot be created, it will force the Strategic Manager to replan. On the other hand, if an action is invalidated, it will not invalidate the strategic plan, it will simply replan.

Strategic Manager

While the Build Order Manager creates reactively plans, the Strategic Manager, on the other hand, constantly has an active plan. Each strategic goal has a relevancy value that is calculated before plan selection. Depending on the plan, the relevancy value is set between 0.0 and 1.0. Once all goals have calculated their values, the strategic manager selects the goal with the lowest.

There are some cases where the Strategic Manager creates a plan reactively. For example, the BuilAgainst goal is activated whenever an enemy unit appears and there is no available unit that is equally strong. This goal is of the highest importance and, as such, any plan currently active is terminated to create a new plan.

4.5.5 Sensors

Unit Sensor

The unit sensor is a reactive sensor that adds information in the WorkingMemory whenever a unit is created or destroyed. This information is then used primarily during the build order planning process to create the appropriate amount of units needed.

Enemy Sensor

The enemy sensor detects enemy units and creates facts that are placed in the WorkingMemory. This information is used both by the strategic manager and build order manager to make relevant decisions on how to attack and what types of units to construct.

4.5.6 Other Managers

Resource Manager

The resource manager has the responsibility of constantly increasing the economy and keeping the number of workers in a stable level. For every base it creates 12 workers and orders them to constantly gather minerals and gas. If a worker dies, a new one will be created. Also, whenever the current supply is about to reach the limit, it will create a Supply Depot.

Building Manager

The building manager is responsible of carrying out building orders. It constantly goes through the build order queue in the blackboard and, if an order is idle, selects a worker and orders it to start building.

The manager also takes care of various situations that may go wrong. When the assigned worker dies, it assigns a new one. If no worker exists, it waits until one does. If the building is destroyed during construction, it will order the worker to create a new one. Finally, if there are not enough resources, the manager will wait until there are.

All this ensures that the orders will be carried out one way or another.

Training Manager

The training manager carries out training orders much like the building manager. Whenever a training order is added to the queue, it looks for the relevant buildings and starts training the units. It is optimized to share the training of units to multiple buildings. For example, if 12 Marine units need training and there are 2 Barracks buildings, the training manager will use both.

5 Evaluation

5.1 General

The purpose of the evaluation phase is to demonstrate that the solution developed and, by extension, planning, meets the demands of modern video games. The demands were outlined in sections 2.1.2 through 2.1.4 and are repeated below.

The system must:

1. be able to work under limited resources
2. provide NPCs with autonomy as well as the means to control this autonomy
3. be sufficiently reusable for future projects

In addition, it was decided to take advantage of the evaluation phase to emphasize the dynamic and flexible nature of a planning system such as GOAP.

5.2 Operating under limited resources

There were no resources problems encountered during the execution of the system. This is due to the fact that the components developed where not very demanding. Furthermore, it was limited to construction problems and simple battle scenarios. This is not always the case in planning systems and cannot be considered as a general observation.

As the system expands to include more sensors for better decision making, the demands for resources increase rapidly. Jeff Orkin discusses these issues in his paper *Agent Architecture Considerations for Real-Time Planning in Games* [22]. One way of overcoming sensor resource demands, he advises, is distributed processing of the sensors. Instead of updating all sensors, each one is updated depending on their required resources; high-demanding sensors update less frequently than low-demanding.

5.3 Control of behaviors

As shown in Section 4.5.2, StarPlanner contains a GUI where the user can enable or disable goals or actions and, thus, affect the behavior of the system. This is especially evident when there are multiple actions that solve the same problem differently. For example, there are two actions that solve the Barracks symbol: BuildBarracksNormal, BuildBarracksAggressive. The first action builds one Barracks unit while the seconds builds two (making marine production faster).

However, there might be a case when disabling an action renders some plans useless. If no action that can build a Barracks unit exists, then the goal GoalMarineRush cannot be formulated. Currently this can be overcome by generating relevant plans before playing the game to ensure that they can be formulated. A better solution for the future would be to automate this process.

5.4 Reusability

The entire solution consists of a series of abstractions. A* engine, GOAP engine, high-level planning and mid-level planning are all abstractions that are built one on top of the other:

- A* engine is a general purpose A* searching system
    - GOAP engine uses A* engine to generate planning
        - High-level planning is done using GOAP
            - Mid-level planning is done when necessary (according to the high-level plan)
                - Low-level managers carry out all planning steps

This architecture is highly reusable and flexible. Improvements and modifications can be done in any level (that may or may not affect higher levels).

In regards to the agent architecture, the blackboard allows high-levels of extensibility. Adding a new planning manager is as simple as creating one and using the blackboard to communicate with other managers, sensors or actuators.

Removing a planning manager is equally effortless. For example, to remove the build order manager and place a non-planning manager, all there is to be done is the integration of the blackboard in the new system. That is, the new system must be able to "understand" what the goal placed by the strategic manager is and perform accordingly.

5.5 Plan validation and re-planning

During plan execution many things can go wrong. The agent constantly queries the current action for validity. If the action is invalid, then the plan is halted and re-planning takes place.

Re-planning is treated differently in each planner. If an action of the construction planner is invalidated, it is not propagated to the high-level (strategic) plan. Instead, the construction planner will try to produce a new plan. However, if a plan cannot be formulated, then the high level plan is invalidated.

This demonstrates the flexibility of using planning in multiple levels. Adding new planners in the system can be accomplished by modifying the blackboard in order to be used by other systems.

5.6 Goal Selection

Each planner uses different methods of goal selection. Each strategic goal has a relevancy value; the lower the value the more relevant the goal. Every time the strategic planner needs to select a goal, each goal recalculates its relevancy value.

The construction planner generates plan reactively. When a step of the strategic plan requires the construction of units, a goal is positioned in the blackboard where the construction picks it up and formulates a plan.

Once again, this highlights the flexibility of planning. Goal selection can be done independently, thus enabling optimizations at this level.

5.7 Dynamic Goal Generation

Finally, another example of the system's flexibility is seen when the goal BuildAgainst is in effect. When an enemy unit appears, the agent will look whether a counter-unit is available (a unit that can defeat the enemy unit). If no such unit exists, then the BuildAgainst goal is set as the most relevant strategic goal and re-planning is enforced.

This goal will result in a plan where the first step is to build the counter-unit. This involves creating a goal where the preconditions are set dynamically.

6 Conclusion & Discussions

6.1 Conclusion

Since the success of the NPC's decision making in F.E.A.R. the video game industry has been increasingly using planning techniques. This inspired the author to investigate what distinctive characteristics make planning interesting. After an initial study on F.E.A.R.'s planning system, it was discovered that planning had many was more suitable to the demands of modern video games than conventional techniques. This was the motivation behind the project; to demonstrate how planning meets these demands.

To achieve this goal, more research had to take place. The research was done in two main areas: artificial intelligence in video games and automated planning. The former would define the demands of modern video games, while the latter would help understand how planning works. Finally, a research on video games that used planning was done to determine which planning technique to use for the system and in which video game the system would run.

The results of this research where then analyzed to discover what the requirements of such a system. The planning technique decided was Goal-Oriented Action Planning (GOAP) and the video game was StarCraft.

The next step was to actually develop the system. A bottom-up approach was taken to ensure extensibility (which was one of the demands of modern video games). A general-purpose GOAP system was developed and then extended to a StarCraft-specific system. A lot of hours of testing took place to make sure that everything worked fine.

Finally, based on the demands previously discovered, the system was evaluated. Overall, the system indeed met those demands and the aim of the project was successful. As a bonus, a lot of knowledge and experience was gained.

6.2 Difficulties encountered

The most difficult part of the integration was deciding how to represent the world symbolically and when to initiate strategic plans.

Initially it was decided that the world state of the agent will consist of the type of the unit (as the property name) and the number of units of that type (as the value of the property). A goal would set, for example, "marines = 12" to denote that there must be 12 marines in the game. But what would an action's precondition be? It cannot be "marines = 12" because this would be true only in the rare case when marines are indeed 12. An action is interested in whether marines are *fewer than* 12; not exactly 12. The temporary solution was to create "dummy" preconditions that will be set at runtime.

This was a bad decision as many things had to be change during runtime. Sometimes an action would be included in the plan even though there was no need for it.

The next problem encountered was the initiation of plans. When do you start planning for a goal in an RTS? When an enemy unit is in sight? Of course not; it is too late to start creating an army when the opponent has already sent troops.

It is very difficult to decide when a strategic plan needs to be created and what the goal will be. StarPlanner has a very basic solution for this problem; each strategic goal calculates its relevancy based on how much time has passed since the start of the game. A better solution would use a learning technique that would calculate each goal's relevancy based on the knowledge of the opponents strategies. However, this was outside of the scope of the dissertation.

Finally, the development of any Artificial Intelligence technique for a video game requires many hours of testing because many issues can only be discovered during the game. This is especially the case when the source code of the video game is not available which would allow some level of automation. However, the debugging screen of JStarPlanner helped overcome this problem.

6.3 Future work

There are many aspects of the system that can be improved; especially if the goal is to participate in the StarCraft AI competition. For example, goal selection is a very important factor in RTS and, as mentioned earlier, it could be based on how the opponent plays. Using a neural network to set the relevancy values of each goal would make this possible.

Adding more actions and goals to extend the system's behavior could also be done. For example, since the implementation only includes actions and goals of the Terran race, one could add behavior for the Protoss and the Zerg races as well.

Finally, a plan validator could ensure that a plan can be formulated after enabling or disabling actions or goals.

References

[1] B. Schwab, *AI game engine programming*. Hingham, MA: Charles River Media, 2004.

[2] A. J. Champandard, *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Games, 2003.

[3] P. L. Shari, *Software engineering theory and practice*. England: Prentice Hall, 2006.

[4] T. Jones, *Artificial Intelligence: A Systems Approach*. Hingham, MA: Infinity Science Press, 2008.

[5] A. Nareyek. Special Interest Group on Artificial Intelligence. [Online]. http://archives.igda.org/ai/

[6] A. J. Champandard. (2007, Sep.) Top 10 most influential AI games. [Online]. http://aigamedev.com/open/highlights/top-ai-games/

[7] J. Orkin. (2009, Sep.) Goal-Oriented Action Planning (GOAP). [Online]. http://web.media.mit.edu/~jorkin/goap.html

[8] A. J. Champandard. (2009, Jun.) Paris Game AI Conference '09: Highlights, photos & slides. [Online]. http://aigamedev.com/open/coverage/paris09-report/

[9] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2002.

[10] M. Buckland, *Programming Game AI by Example*. Plano, Texas: Wordware Publishing, Inc, 2005.

[11] I. Millington, *Artificial Intelligence for Games*. San Francisco: Morgan Kaufmann, 2006.

[12] J. Gregory and J. Lander, *Game Engine Architecture*, J. Lander, Ed. Wellesley, MA: A K Peters, Ltd., 2009.

[13] D. S. Cohen, S. A. Bustamante, and T. J. Park, *Producing games : from business and budgets to creativity and design*. Amsterdam: Focal Press, 2010.

[14] K. Salen and Z. Eric, *Rules of play: game design fundamentals*. Cambridge, Mass: MIT Press, 2003.

[15] G. Tavinor, *The art of video games*. Malden, MA: Wiley-Blackwell, 2009.

[16] M. A. Boden, *Artificial Intelligence*. San Diego: Academic Press, 1996.

[17] N. J. Nilsson, *Artificial Intelligence: a new synthesis*. San Francisco, Calif: Morgan Kaufmann Publishers, 1998.

[18] B. Coppin, *Artificial intelligence illuminated*. Boston, USA: Jones & Bartlett Publishers, 2004.

[19] J. Orkin, "Applying Goal-Oriented Action Planning to Games," in *AI Game Programming Wisdom 2*. Charles River Media, 2003.

[20] K. Graft. (2009, Feb.) Edge. [Online]. http://www.edge-online.com/news/blizzard-confirms-one-frontline-release-09

[21] B. Weber. (2010) StarCraft AI Competition. [Online]. http://eis.ucsc.edu/StarCraftAICompetition

[22] J. Orkin, "Agent Architecture Considerations for Real-Time Planning in Games," in *AIIDE*, 2005.

[23] J. Orkin, "Symbolic Representation of Game World State: Toward Real-Time Planning in Games," in *AAAI Challenges in Game AI Workshop*, 2004.

[24] R. Burke, D. Isla, M. Downie, Y. Ivanov, and B. Blumberg, "Creature Smarts: The Art and Architecture of a Virtual Brain," in *Game Developers Conference*, San Jose, California, 2001, pp. 147-166.

[25] Unknown. (2010) bwapi. [Online]. http://code.google.com/p/bwapi/

[26] R. Inder, "Planning and Problem Solving," in *Artificial Intelligence*, M. Boden, Ed. Academic Press, 1996, ch. 2.

Appendix A Setting up the environment for JBridge

To set up JBridge, one must perform the following steps (taken from the JBridge website)

1. Download the Chaoslauncher and extract it anywhere
2. Download BWAPI 2.4 BETA from the BWAPI project page and extract it anywhere
3. Copy the entire contents of BWAPI's Chaoslauncher/ directory into where you extracted the Chaoslauncher
4. Copy the entire contents of BWAPI's Starcraft/ directory into the folder where `StarCraft` is installed
5. Copy the entire contents of BWAPI's WINDOWS/ directory to your C:\WINDOWS or C:\WINNT folder depending on your version of Windows
6. If you do not have Java 6 (which you should since you're a Java developer), then download it and install it (JDK preferred)
7. Make sure your JAVA_HOME environment variable points to where Java is installed (jre path recommended if you installed a jdk, but not required)
8. Download the latest release of the BWAPI Bridge in this project (0.1 ALPHA) and extract it anywhere
9. Copy the bridge's bwapi-bridge.dll, bwapi-bridge.jar, and lib/collections-generic-4.01.jar to your `StarCraft` install folder under /bwapi-data/AI
10. Open bwapi-data/bwapi.ini in your `StarCraft` install folder and change ai_dll property to use bwapi-bridge.dll instead of `ExampleAiModule.dll`that's default (or just change the name of bwapi-bridge.dll to `ExampleAiModule.dll` in the bwapi-data/AI folder)

Appendix B StarPlanner in action