Department of Computing
Imperial College London

An A.I. Player for DEFCON: An Evolutionary Approach Using Behavior Trees

Final Year Individual Project

Interim Report

By: Chong-U Lim

Supervisor: Dr. Simon Colton

2nd Marker: Prof. Ian Hodkinson

March 13, 2012

Contents

Chapter 1

Introduction

Video games have become technologically more advanced over the years - exhibiting realistic graphic capabilities coupled with engaging gameplay. The area of Artificial Intelligence (AI) in video games has also arisen as an important factor in determining the quality of the game by measure of a player's experience. With increasing demand for more realistic computer controlled players, events and opponents capable of exhibiting human-like characteritics, AI has been a significant area of interest for the games industry and academics alike. Several methodologies exist to approach the modelling, scripting and the design of AI for video games, aiming to make game-controlled entities more realistic and intelligent. Behavior Trees attempt to improve upon exisiting AI methodlogies by being simple to implement, scalable to handle the most complex of game tasks and modular to improve reusibility - ultimately improving the efficiency for developing and designing game AI.

In this project, we plan to use Behavior Trees to design and develop an AI-controlled player for the commercial real-time strategy game DEFCON. We approach the design of Behavior Trees for the AI using a behavior-oriented methodology which we introduce as Behavior oriented Design (BOD) and also intend for the AI to adapt and learn to play the game of DEFCON by allowing the AI to evolve into a better player automatically using evolutionary machine learning techniques. The project aims to showcase the feasibility of combining such machine learning techniques together with behavior trees as a practical approach to developing an effective and intelligent AI player.

1.1 Motivation

1.1.1 Improvements of Behavior Trees over traditional game AI

Behavior Trees have been proposed as a new approach to the designing of game AI, with advantages over other AI approaches being its simple design, scalability and modularity. Behavior Trees have been adopted for use in commercial games for various uses including controlling character animation and coordinating non-playable character(NPC) behaviors. An example of the use of Behavior Trees in commercial game AI includes the AI developed for the commercial First-Person-Shooter(FPS) Halo2 [**?**]. Thus, the project wishes to investigate the feasibility of Behavior Trees for the purpose of designing an automated AI player for a real-time strategy(RTS) game such as DEFCON.

1.1.2 Complexity of DEFCON as a Real-Time Strategy Game

Real-time strategy (RTS) games offer a level of complexity which differ from other games such as platform games, puzzle games or even first-person-shooters. There exists a need to perform micro-managing over the numerous units in the game, as well as adopting strategies to be able to outwit opponents. DEFCON differs from other RTS games as well. In most other RTS games, winning heavily centres around accumulating resource and maximising the size of one's army or number of units. In DEFCON, however, players have an equal set of resources and number of units. The key factor to winning centres around a coming up with a good strategy to coordinate one's units and resources effectively. DEFCON thus exists as a challenging and interesting platform to develop an AI for.

1.1.3 Machine Learning Applicabilities in Video Games

Previous, the use of machine learning to aid the development of AI in videogames was not commonplace in commercial game development due to factors such as high offline computation and unpredictability of results. However, over time, machine learning techniques have begun to be adopted by commercial game developers after discovering that AI is a necessary ingredient to make games more entertaining and challenging [**?**]. Several machine learning techniques such as Artifical Neural Networks [**?**] have begun to be used, but several areas such as evolutionary programming, remain less popular due to the vast amount of time and computation required to attain satisfactory results - making them unpractical for games developers

and their games - with tight schedules to meet deadlines and requirements for fast, memory efficient computation respectively.

Thus, we intend to show that evolutionary methods, in particular genetic algorithms, can be used as a practical approach to develop an entertaining, believable and challenging AI and pave the way to its adoption in the future.

1.2 Objectives

1.2.1 Implement Behavior Trees to encode game AI using Behavior Oriented Design as a design methodology

Exhibit the use of Behavior Trees as a way of developing an AI that is able to handle the complexity of a real-time strategy game. Also, to showcase the Behavior Tree's ease, flexibility and modularity as a means for developing game AI.

1.2.2 Research on feasibility of using evolutionary algorithms to develop and improve upon AI

Demonstrate the ability of game AI to learn to play effectively, and improve upon itself by the use of evolutionary techniques, and thus, the possibility of using such techniques in the development of improved AI in commercial games.

1.2.3 Demonstrate the use of reactive planning using Behavior Trees

Allow the AI developed to accomodate for unpredictability in the game space using reactive planning, allowing it to exhibit intelligent behavior rather than naively following a given plan.

Chapter 2

Background

Overview

In this chapter, we present information from various sources that provide a deeper understanding of certain methodologies, applications and technical information pertaining to this project. For the purpose of this interim report, in addition to the details of the background topic, suggested applications of each of these topics for the purpose of this project are also presented.

In Section **??**, we introduce the commercial real-times strategy game DEFCON covering gameplay, game rules and coverage of existing AIs that have been developed for the purpose of automating the playing of DEFCON.

In Section **??**, we introduce the design and usage of Behavior Trees - a hierarchical structure used to encode AI. We go over what defines a behavior tree, its composition and run through various examples of the usage of different types of behavior trees.

In Section **??**, Behavior-Oriented Design (BOD) is introduced as a programming methodology to approaching AI design. We introduce the motivation and proposed design process, and also present our proposed approach to using Behavior Trees to make use of this design methodology.

In Section **??**, we cover Genetic Algorithms as a Machine Learning technique, covering its definition and operators, and also present our proposed approach to its application to Behavior Trees.

2.1 Defcon

DEFCON is a multiplayer real-time strategy game that allows players to take the roles of the military commanders of one of six possible world territories. Players are given a set of units and structures at their disposal, and it is up to them to manage these resources and inflict the biggest amount of damage against opposing players. The game is split into 5 discrete intervals, named from DEFCON5 to DEFCON1, and these intervals dictate the possible movements and manipulation of units.

It's described as a *genocide*-em-up, a play on the term *shoot*-em-up and *beat*-em-up often used to classify other game genres such as first-person-shooters and action games respectively, and since a large element of the game of DEFCON involves the use of nuclear missles to annihalate large populations of opponent players. Battles are fought in sea and air, and the game involves strategic planning and decision making in coordinating all these units in order to win. Figure **??** shows an in-game screenshot of a coordinated attack in DEFCON, taken from DEFCON's official website [1].

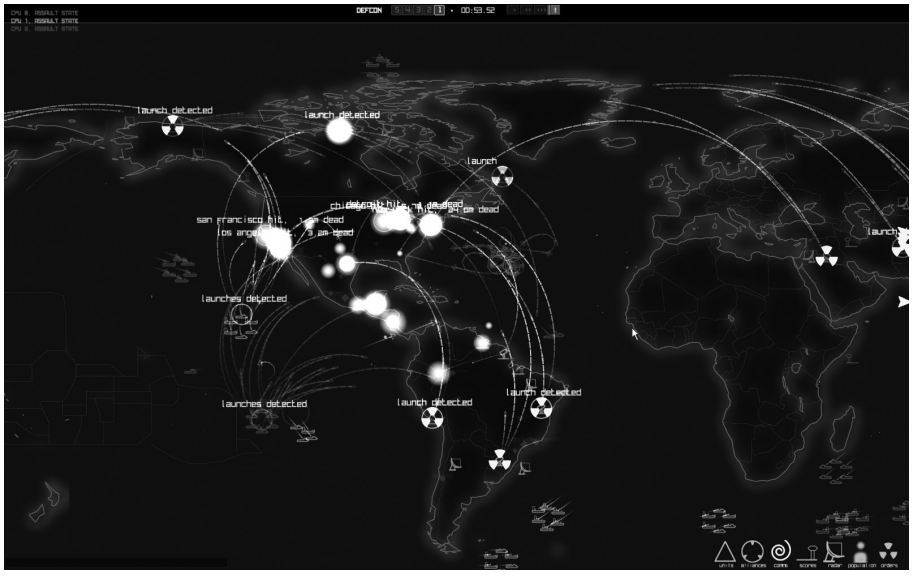


Figure 2.1: Global conflict erupts with a series of co-ordinated strikes in Defcon

In this section, we begin by covering an overview of the game of DEFCON, covering basic game rules and gameplay elements. We proceed to present existing AIs which are presently available. Finally, we briefly cover the DEFCON API which is being used for the purpose of developing an AI

[1] Introversion Software's Official DEFCON Site: http://www.introversion.co.uk/defcon/

for this project.

2.1.1 Overview

The information here has been consolidated from both the official DEFCON manual[2] and Robin Baumgarten's Thesis "Automating the Playing of DEFCON" [**?**]

Parties & Territories

A **party** represents the player, either human or AI-controlled, in the game of DEFCON. As such, there are at least 2 parties in the game. Each party is assigned a **territory** at the start of a DEFCON match, and this allocation is either done randomly or chosen.

There are 7 available **territories** in each game, and each territory is controlled by up to 1 party. As such, in a game involving two players, two territories will be controlled by one player, and the remaining five territories will remain unassigned to any party. Each controlled territory possesses at least one **city**, and each city consists of a non-negative number representing its population. The total sum of the population for each territory is equal between all 7. The territory designates the points on the map that a player may place **land units** and each player may only do so in his or her respective territories.

A portion of the sea, **sea-territories**, are associate with each party whereby **navy units** may be placed in.

Units

Each party is allocated a fixed quantity of units that it may place and make use of throughout the course of the game. Each unit possesses one or more states which indicate the type of actions is may execute. The units are classified into 3 groups, namely,

- Ground Installations
- Navy Units
- Aerial Units

[2] Available at the official website

Ground Installations

- **RadarStation**: Radars scan a wide range and display enemy units within their range on the map.
- **Silo**: Silo contains 10 Long Range Ballistic Missiles (LRBMs) and can take 3 direct hits before it is destroyed. A Silo has two modes:
  - **Nuclear Launch**: A target on the map may be manually selected to launch a missle. Upon launch, it can not defend itself, and reveals its position to other players after launching its missiles.
  - **Air-Defense**: Enemy airforce units and nukes in range are automatically attacked, in the decreasing priority of Nukes over Bombers over FIghters. If there are several of the same class, it chooses the closest.
- **Airbase**: Starts with 5 bombers, 5 fighters and 10 Short Range Ballistic Missiles (SRBMs) by default. An Airbase has two modes:
  - **Launch Fighters**: A target within range may be targetted, in which Fighter units are launched. The recharge time after the launch of a Fighter, whilst no other Fighters can be launched is 20 seconds.
  - **Launch Bombers**: A target within range may be targetted, in which Bomber units are launched. The recharge time after the launch of a Bomber, whilst no other Bombers can be launched is 20 seconds.

Air Forces

- **Fighter**: Fighters may attack other Fighters and Bombers, and may used for scouting to discover enemy installations. A Fighter has limited fuel, and automatically return to any Airbase or Carrier upon completion of objective, otherwise it will crash and is lost.
- **Bomber**: Bombers carry a single Short Range Ballistic Missle (SRBM), and can be used to fly and deploy nukes at close range. A Figher has two modes:
  - **Naval Combat** Attacks visible enemy Navy units.
  - **Missle Launch** A target may be targetted to launch a misle against.

Navy Units

- **Carrier**: Starts with 5 fighters, 2 bombers and 6 SRMBs. Can also launch depth charges against submarines in the vicinity. It consists of 3 modes:
  - **Fighter Launch**: Hostile units within range may be targeted and Fighters are launched.
  - **Bomber Launch**: Hostile units within range may be targeted and Bombers are launched.
  - **Anti-Submarine**: A sonar scan is performed, and enemy submarines within range are revealed, where a depth charge is released.

  .
- **Battleship**: Battleships can only attack with conventional weapons,however they are extremely effective against other naval units and aircraft.
- **Submarine**: Submarines contain 5 Medium Range Ballistic Missiles(MRBMs) and are either submerged or surfaced. They are invisible to radar while submerged, but must surface to launch. They can be detected by sonar pings from carriers or other subs. Once detected or surfaced they are very vulnerable to attack. It has 3 modes:
  - **Passive-Sonar**: Submarine remains invisible to radar and can only be detected by carriers in anti submarine status and other submarines in active sonar mode. It can attack hostile naval units if they are visible to the party of the submarine.
  - **Active-Sonar**: Similar to Passive-Sonar, except in addition, Submarine creates a sonar scan, which reveals all naval units within a certain distance from the submarine.
  - **Launch Missle**: A target within range may be attacke, where a missle is launched. In this state, the submarines surfaces and is no longer invisible until all its missles have been depleted whereby it returns to Passive-Sonar mode.

Defcon Level	Description	Time
5	No hostile actions. Ground installations may be placed. Fleets may be placed and moved within international waters	3
4	Radar coverage will provide information on enemy units if within range	3
3	Units and ground installations may no longer be placed. Naval and Air attacks not involving Missles are permitted	3
2	This is essentially the same as in DEFCON 3	3
1	No units may be placed, and missle attacks are now allowed. This phase continues until 80% of all missles in the game has been launched, after which a victory timer counts down until the end of the game.	-

Figure 2.2: Summary of DEFCON levels and permitted actions

Defcon Levels

The course of the game of DEFCON spans over the couse of 5 DEFCON phases, starting from DEFCON5. In each DEFCON phase, certain actions and information are permitted whilst others are not. Table **??** gives an explanation of each of the DEFCON phases and a description of what is involved in each.

Winning Conditions

Winning in the game of DEFCON involves attaining the higest score, and how these scores are calculated differently according to the *Scoring Mode* which is decided at the start of a game. The 3 modes are:

- **Default**: 2 points awarded for every million of the opponent's people killed. -1 point penalty for every million people belonging to the player.
- **Genocide**: 1 point is awarded for every million of the opponent's people killed. No penalty for losing a player's own people.
- **Survivor**: 1 point is awarded for every million people surviving in the player's territory. The points start at 100 for both teams and decrease throughout the game.

2.1.2 AI in DEFCON

Several implementations of AIs have been created for the puporse of automating the playing the game of DEFCON. These implementations of AI-controlled players are often referred to as **bots**[3], or **CPU Players**.

We cover two such bots in this section, namely the default bot that ships together with the game which allows human players to pit themselves against an opponent without the need for a network connection with another human player. The second is the result of a Master's thesis by Robin Baumgarten [**?**], which was developed using a combination AI machine learning techniques.

Introversion Bot

The default bot[4] that comes with DEFCON is a deterministic, finite-state-machine driven bot [**?**]. It consists of a set of 5 states and transits from one state to the next in sequence. Upon reaching the final state, remains in it until the end of the game. This is depicted in Figure **??**, The states and a brief description, as identified by Baumgarten [**?**] of what occurs in each are:

[3]For the purpose of this report, the term **bot** is used to refer to any AI-controlled player in DEFCON

[4]Henceforth termed as the Introversion Bot

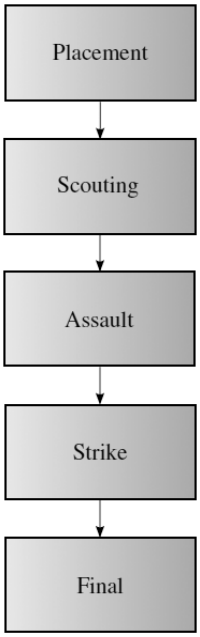


Figure 2.3: Sequence of States in Introversion Bot

- **Placement**: Fleets and structures are placed. The fleet is randomly placed at predefined starting positions. Structures are placed near cities. Once all units are placed, the AI proceeds to the next state.
- **Scouting**: The AI tries to uncover structures of a random opponent by moving fleets towards occupied territories and launching fighters towards them. Once 5 structures have been uncovered or a predefined assault timer 10 expires or the victory timer starts the next state is invoked.
- **Assault**: The AI starts to launch missile attacks with bombers and subs on the previously chosen opponent. Once 5 structures have been destroyed or the assault timer expires or the victory timer starts, the strike state is invoked.
- **Strike**: Silos launch their missiles and other missile carrying units continue to attack. After these attacks have been initiated, the system changes into the final state.
- **Final**: In the final state no more strategic commands are issued. Fleets continue to approach random attack spots.

Robin's Bot

In 2007, a bot was developed by Baumgarten [**?**] using a combination of AI machine learning techniques such as case-based reasoning, decision tree algorithms and hierarchical planning[5]. For the case-based reasoning system, high-level strategy plans for matches were automatically created by querying a case base of recorded matches and building a plan decision tree. A broad overview of the system design is depicted in Figure **??**.

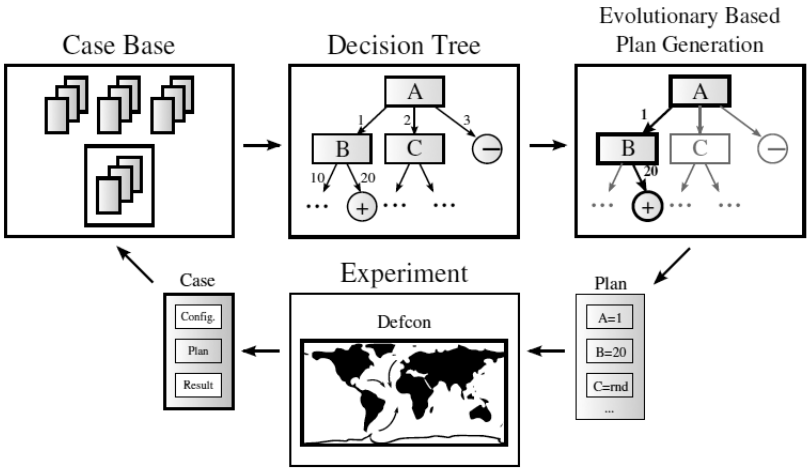


Figure 2.4: Overview of System Design in Robin's Bot

[5]Henceforth referred to as Robin's Bot

The implementation of the bot can be described as a series of steps:

1. At the start of the game, territories are assigned randomly to both the Introversion Bot and Robin's bot
2. Upon assignment, the starting configuration is used as similarity measure to retrieve a case from the case-base
3. Plans from the retrieved case classified into a decision tree with plan attributes
4. The decision tree determines a high-level plan coordinating groups of fleets, termed metafleets, and are placed as dictated by the plan.
5. The placement of structures is controlled by an algorithm taking recent matches into account by querying the case base
6. With everything placed, an attack on the opponent's mainland is prepared establishing where and when to attack
7. A synchronised attack is performed, and the game ends
8. Results, the plan used, fleet movement and structure information are extracted into the case and retained in the case base.

2.1.3 DEFCON API

Funded by a Technology Strategy Board feasibility study grant, via a partnership of Introversion Software and Imperial College, Imperial College has developed an Advanced Programming Interface (API) for Introversion's DEFCON game [6]. The API provides an interface to DEFCON's methods and function calls, and is used to produce a dynamic-link library (.dll) file that contains an AI implementation which can be then included as a module for DEFCON to access and use as an AI player.

The API provides a way for AI developers to develop AI bots for the game of DEFCON, without having to work with the source-code of DEFCON directly, expanding its reach to people to give a shot at developing an AI for DEFCON. At the time of writing, the API is at version 1.51, and provides the base framework for which this project is based on. By making use of the API to develop the bot, we hope to progressively contribute ideas and improvements to the API together with advances in this project.

[6] API and documentation available from http://www.doc.ic.ac.uk/ rb1006/projects:api

2.2 Behavior Trees

Behavior Trees represent a new way of defining the AI for video games. Its key characteristics are being simple to define, scalable to exhibit complex AI and modular for reusability [**?**]. This is achieved by introducing a set of constructs which a Behavior Tree is essentially composed of in the form of nodes. Behavior Trees are essentially goal-oriented, with each tree associated with a distinct, high-level goal which it attempts to achieve. Behavior Trees can be linked together, allowing the exhibition of complex behaviors by first defining smaller, sub-behaviors.

In this section, we take a look at the motivation for Behavior Trees, making comparisons to other traditional forms of defining game AI used in practice. We then cover the 5 basic constructs that are used to create Behavior Trees. Following that, we discuss the goal-oriented design of Behavior Trees and finally, a discussion on how this enables Behavior Trees to perform planning to achieve these goals.

We have adopted a similar convention for the style and design representing the types of nodes in Behavior Trees from Alex J. Champandard's presentation on Behavior Trees [**?**]

2.2.1 Motivation

A traditional approach to developing AI for games has been to use Finite State Machines (FSM). To illustrate this, we use the following diagrams and example regarding the AI for a simple service robot from Ian Millington's Artificial Intelligence for Games [**?**].

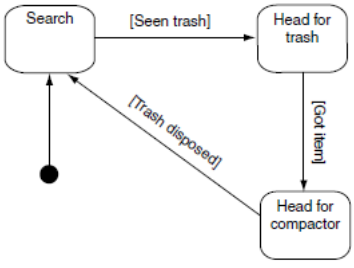


Figure 2.5: Basic FSM of a cleaning robot

Figure **??** illustrates the FSM for a the AI of simple service robot, and how a FSM handles its abilities to search around for objects that have been dropped, pick one up when it finds it, and carry it off to the trash compactor.

Now consider Figure **??**, whereby the robot now has an alarm system to indicate whether its power is running low.

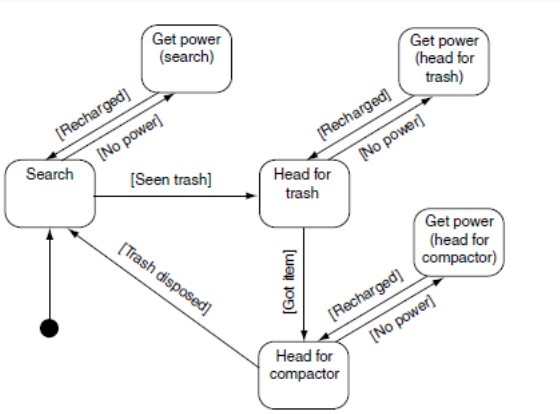


Figure 2.6: Extended FSM of a cleaning robot

It can be seen that the number of states have now doubled in order to accomodate this new functionality. Thus, it can be seen that if the subject grows in complexity, the number of states that are required to represent the AI logic increases, along with the transitions between the states. This led to the introduction of Hierarchical Finite State Machines (HSFMS), which were introduced to overcome these drawbacks, improving scalability by grouping states to share these transitions to develop larger and more complex AI systems. This is illustrated in Figure **??**.

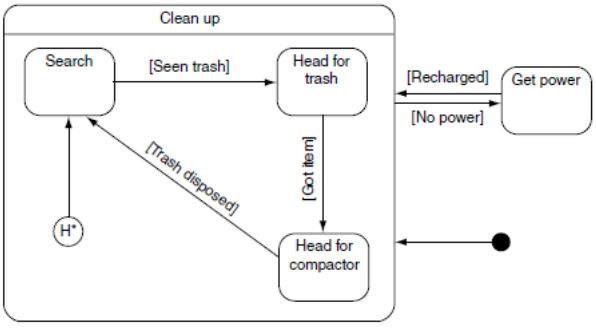


Figure 2.7: Hierarchical FSM of a cleaning robot

However these numerous transitions still have to be defined carefully in order for them to be reused. Another drawback is that, supposing we wanted to define additional AI logic, and realise that a portion of this new

logic consists of states that are in principle similar to states that were defined before. Reusing older states would involve a rather tricky task of identifying transitions which are compatible for this new portion of the AI logic before attaching them.

One solution to this problem would be to allow states to be reused easily, without worrying about transitions being invalid when they are reused for different portions of the AI logic - essentially increasing the modularity. A Behavior Tree enables such modularity by encapsulating logic transparently within the states, making states nested within each other and thus forming a tree-like structure such as in Figure **??**, and restricting transitions to only these nested states. This exhibits what is termed as **latent computation** [**?**]. The root node branches down to more nodes until the leaf nodes are reached, and these leaf nodes are the base actions that define the behavior of the AI from a state beginning at the root. The concept of an AI state is now seen as a high level AI behavior, or task, whereby the links to nested children nodes define sub-tasks which make up the main behavior. The leaf nodes are essentially then a group of basic actions that define a behavior.

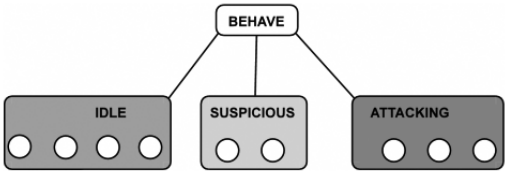


Figure 2.8: A Tree of modular Behaviors [**?**]

These behavior tasks are formally constructed out of 2 classes of constructs[7]. The Primitive constructs form the leaves of the tree, and define low level actions which describe the overall behavior. The composite constructs provide a standard way to describe relationship between children nodes, such as whether all should be executed or only a subset of them. In the next section, we cover these constructs, describing in detail their usage and applicabilities.

2.2.2 Primitive Constructs

Primitive Constructs form the leaves of a Behavior Tree, and they define low-level basic actions. In Game AI, they are esentially the interface between the AI logic and the main game, and are wrappers over function calls that are used to execute an action within the game world, or query the state of

[7]As such, these constructs are also referred to as **tasks** or **task nodes**

the game.

Actions and Conditions

Two primitive constructs consist of 2 types, namely,

1. `Actions` - which cause an execution of methods or functions on the game world, e.g. Move a character, Decrease health, ...
2. `Conditions` - which query the state of objects in the game world, e.g. Location of character, Amount of Health points, ...

These actions and conditions form the definitions of behavior tasks. For example, in Figure **??**, we define a Behavior Tree task Deduct Money From Player.

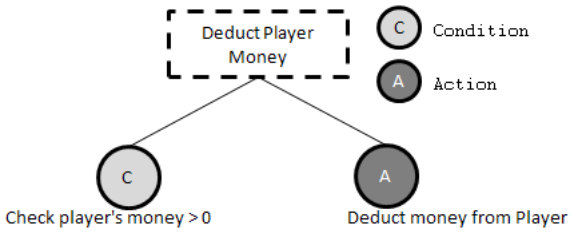


Figure 2.9: A Behavior Task - Deduct Money From Player

The main task is decomposed into two child tasks, the left child being a `Condition` that the player has money to start with, and the right child being an `Action` which physically deducts the money from the player in the game. Each of these actions and conditions can either succeed or fail, which in turn define whether the high-level behavioral task succeeds or fails.

2.2.3 Composite Constructs

Now that we have introduced the notion of tasks, and how conditions and actions are used as child nodes to define these tasks. However, behavior tasks of these sorts are limited to very basic tasks. We are interested in increasing the complexity of these behavior tasks and this is achieved by building branches of the tree to organise the children nodes. These branches are essentially the composite constructs of behavior trees, and their main purpose involves organising the children nodes of each task. The composite tasks consist of,

1. `Sequences`
2. `Selectors`
3. `Decorators`

Sequences

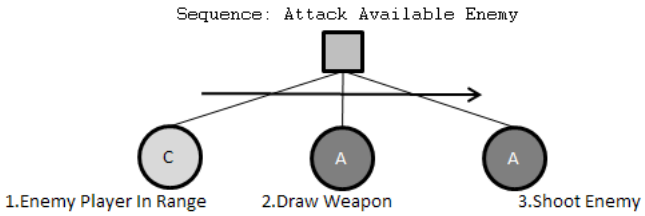


Figure 2.10: Sequence Example - Attack Available Enemy

Figure **??** shows a `Sequence` node, consisting of 3 children nodes, each being a primitive type. A `Sequence` essentially imposes an ordering on the execution of its children nodes.

In this example, it first checks if there is an enemy in range. Secondly, it draws a weapon before finally attacking the enemy. Note that the order of execution is important, since logically, a weapon has to be drawn before it can be used to fire at an enemy. Thus, in order to determine if the entire `Sequence` has executed successfully, each one of its children nodes has to execute successfully in the order that they are specified.

If in the event that an enemy is present, but the AI player does not have a weapon to draw, the second child task node fails, resulting in an automatic failure of the entire `Sequence`. The third child node, in this case, does not need to be checked or executed. Sequences can be seen to perform

implicit pre-condition checking to prevent potentially unexpected behavior from occuring (i.e. an AI player attempting to shoot before drawing a weapon).

Selectors

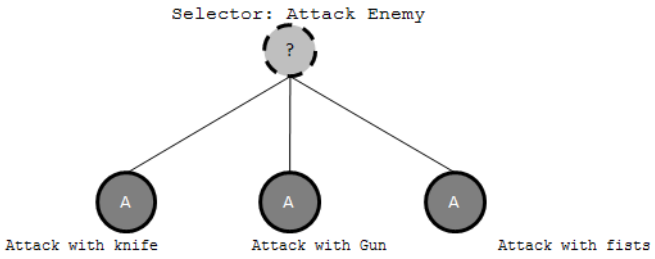


Figure 2.11: Selector - Attack Enemy

The second type of composite we introduce is the `Selector`. Figure **??** shows a `Selector` node, consisting once again of 3 children nodes. The first thing to notice is that there exists no formal ordering on these children nodes. The reason is because, a `Selector` essentially chooses, or picks, one of its children nodes to be executed. This child node can be selected randomly, probalistically, or using priorities. For this example, we assume that each child node has an equal probability of being selected.

On execution of a child node, the `Selector` succeeds only if at least one of its children nodes succeed. In our example, the game AI selects from 3 alternatives to complete the task Attack Enemy. If it first picks Attack with Gun, the corresponding `Action` executes. Supposing the `Action` fails, perhaps due to the AI player not possessing a gun at this point, the `Selector` node does not immediately fail, but rather, it picks a next child node to execute.

Continuing with our example, supposing the next `Action` it picks is Attack with knife. Assuming the AI player possesses a knife at this point and executes the action successfully, the `Selector` node then returns as successful. The `Selector` only fails when none of its children execute successfully. This can be seen as exhibiting an opposite behavior to that of `Sequences`.

Sometimes, we assign priorities to each of the child task nodes. A `Selector` that selects its child task nodes in the order of decreasing priority is termed a `Priority Selector`.

2.2.4 Decorators

We have so far seen primitive and composite tasks which are used to design and define our Behavior Trees. `Decorators` are a class of tasks inspired by the software design pattern of the same name. Just as how Decorators in sofware design aim to wrap over an existing class, adding additional features on top of, but without modifying, the existing class, Behavior Tree `Decorators` aim to extend existing defined Behavior Trees to improve its functionality.

This is best illustrated with an example - consider the Behavior Tree in Figure **??**. Supposing that in order to achieve a more realistic AI, we want the AI player to exhibit this ATTACK AVAILABLE ENEMY behavior for only a certain duration of time within the game, such that there are points when the AI player is considered "off-guard", to mimic how a human player would not always choose to attack an enemy even though one were present.

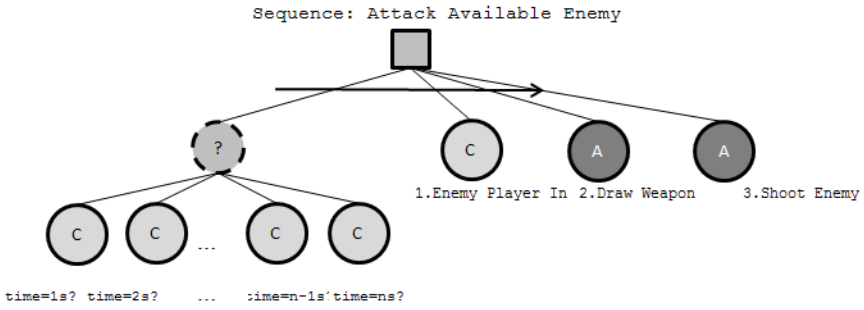


Figure 2.12: An approach to handling timed behaviors

If we were to try to design a Behavior Tree to exhibit this behavior, we could try modifying our existing Behavior Tree to include several more `Condition` tasks linked together by a `Selector` into the existing `Sequence`, with each `Condition` checking for specific points of time within the game, such as in Figure **??**.

This, however, can be seen to be undesirable on three accounts. Firstly, it breaks the simplicity of designing Behavior Trees because of the need to tediously create multiple `Condition` nodes. Next, this is definitely not scalable because if we were to increase the time duration to execute this behavior, more `Condition` tasks have to be added. Lastly, it breaks the rule of modularity since we are unable to reuse this Behavior for a different time duration other than that covered by the existing `Conditions`.

`Decorators` are thus the solution to this problem. What occurs is that

we define a type of `Decorator` - a `Timer`, which executes it's child node for a specified duration of time. The existing behavior does not need to be modified in any way, except that the `Timer Decorator` is linked as the parent to the existing Behavior Tree, as depicted in Figure **??**. Now, the Behavior is simple to create because it only involves the addition of a single `Decorator Timer`, and maintains its modularity since the original behavior of ATTACK AVAILABLE ENEMY has not been modified in any way. Scalability does not pose a problem since there are no additional nodes to add.

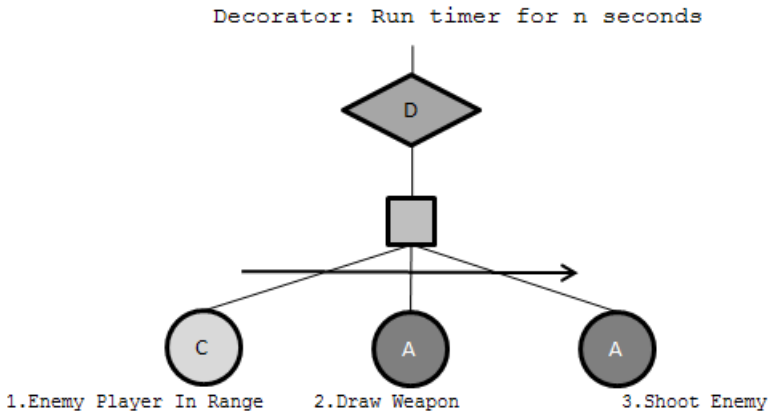


Figure 2.13: Using timer decorators to handle timed behaviors

This exhibits the usefulness of `Decorators` to extend the functionality of Behavior Trees, and more examples of types of `Decorators` include `Counters` which track and execute behaviors a certain number of times or `Debug Decorators` which output information about the corresponding child tasks. A non-exhaustive list of different types of `Decorators` can be found on page **??** of the Appendix.

Lookup Decorators

From the above discussion on `Decorators`, one noticeable point is that `Decorators` never appear as the leaves of a Behavior Tree, since their functionality essentially depends on their children tasks. However, one type of `Decorator` always appears as a leaf node and that is the `Lookup Decorator`, or `Lookups`.

`Lookups` attempt to further increase the modularity and reusiblity of Behavior Trees by using such `Lookups` to search for particular sub-behavior tasks rather than explicitly stating them in the tree. This can be illustrated

by first considering the Behavior Tree in Figure **??**, describing a behavior of an AI player in the game of Defcon in the midst of placing a structure.

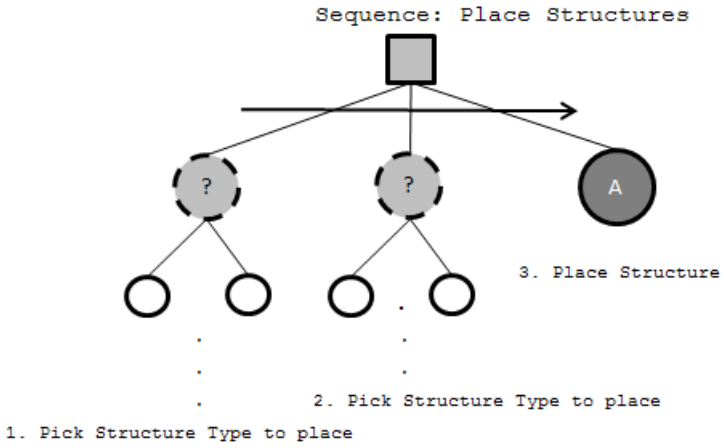


Figure 2.14: Behavior for AI Placing Structures in DEFCON

The Behavior takes the form of a `Sequence`. First it picks a structure to place, then it decides on a location for the structure and finally, it places it. The first two tasks involve `Selectors`, basically because we might design the AI in such a way that it has different implementations of deciding which structure to place. An example would be picking the structure that has the fewest remaining number or an alternative could be picking one with the highest remaining number. Similarly for the second task, we have different implementations from which we wish to select from in deciding the location to place the structure. For example, placing structures nearer the sea areas, closer to cities or closer together.

Now, it can be seen that we are not interested from a high level perspective about the sequence of actions which are required to execute the different implementations of these tasks, we merely want the behavior tree to decide for us and accomplish the tasks Pick Structure Type to Place and Pick Location to Place. The idea is then to associate these different implementations into a lookup table, grouped by their high-level task which they are accomplishing, and replace the children nodes of the main `Sequence` by `Lookup Decorators`, as shown in Figure **??**.

In grouping these implementations, into a hierarchy of tables, we begin to move towards a hierarchical goal-oriented architecture. These high-level tasks can be viewed as high-level goals which the Behavior ultimately wants to achieve, and does so by decomposing down to smaller sub-goals in order to

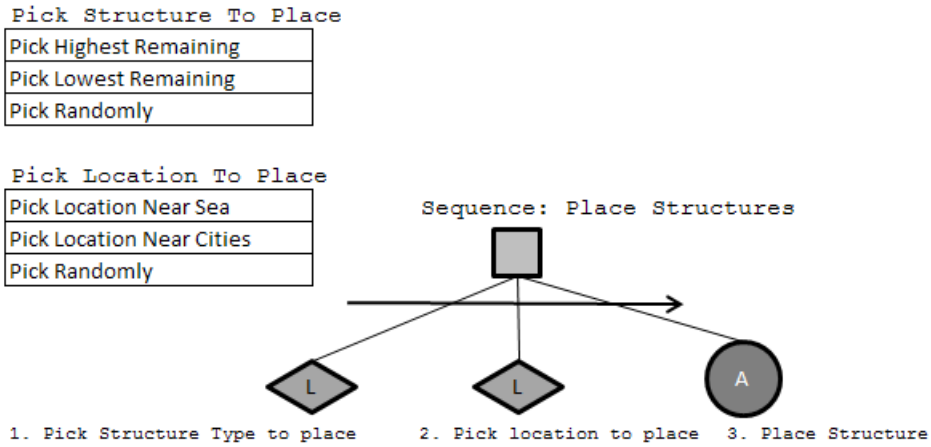


Figure 2.15: Using Lookups to modularise Behaviors

achieve the high level goal. In Section **??**, we discuss a theoretical methodogy of architecting game AI in what is called **Behavior Oriented Design (BOD)** [**?**], and then propose our implementation of using Behavior Trees for the game of DEFCON to approach BOD.

The idea of using `Lookups` to search for possible different sub-behaviors in a goal-oriented manner forms the basis of the idea of performing Planning in AI using Behavior Trees. In brief terms, planning involves the selection and sequencing of related actions from a given initial state to a target state. Its applicabilities to Behavior Trees and a description of how we chose to allow the Behavior Trees for DEFCON to exhibit planning is described in greater detail in Section **??**,

2.3 Behavior Oriented Design

Behavior Oriented Design (**BOD**) is a development methodology proposed by Joanna Bryson [**?**] which aims to design and construct what are termed as *complete complex agents*(**CCA**) in a modular fashion. The methodology serves as an improvement over current arhitectures for these **CCA**. In this section we provide an overview of **BOD**, its proposed approach to designing an AI which focuses on iterative rapid prototyping and how such an architecture would allow the AI to exhibit functionality such as reactive planning, which we will show to be desirable when attempting to design an AI for a complex and dynamic AI.

What interests us most about **BOD** is its close correspondence with Behavior Trees, which we introduced in Section **??**, such as the similar functionality of BOD's `Action Patterns` and `Competences` to the `Sequences` and `Selectors` of Behavior Trees respectively. It can then be shown, based on this correspondence, that idea of planning for arises naturally for both `BOD` and Behavior Trees.

As a result, we shall then present the approach, using a combination of these two concepts, that we use to essentially create a **BOD**-inspired goal oriented Behavior Tree AI architecture for the purpose of this project.

2.3.1 Methodology

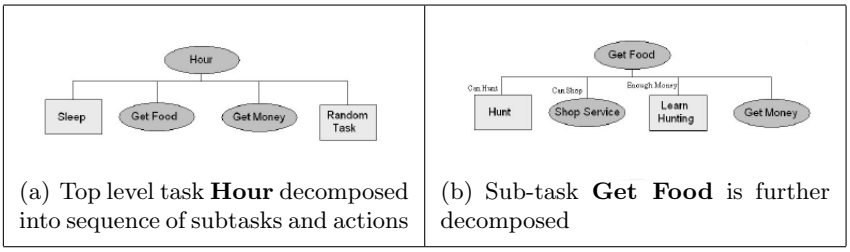
Using Behavior Oriented Design, we approach a modular and behavior-oriented approach to coding the AI. This is achieved by *Behavior Decomposition* - essentially breaking down the complex high-level behavior into smaller sequences or groups of sub-behaviors. The problem that arises often taking this approach is determining how exactly do we decide to decompose a behavior. For example, we might wonder how much decomposition is necessary, how many sub-behaviors is ideal to define the top-level behavior and also how complex should they be. BOD approaches this via a cyclic development process involving a set of guidelines for this decomposition.

We present the two main aspects of this development process, namely the *Initial Decomposition* and subsequent *Iterative Development* as proposed by Bryson [**?**]. After which, we demonstrate how this approach was used to cover the design of the Behavior Trees for this project.

Initial Decomposition

1. Specify at a high level what the agent is intended to do
2. Describe likely activities in terms of sequences of actions. These sequences are the basis of the intial reactive plans
3. Identify an initial list of sensory and action primitives from the previous list of actions
4. Identify the state necessary to enable the described primitives and drives. Cluster related static elements and their dependent primitives into specifications for behaviors. This is the basis of the behavior library.
5. Identify and prioritize goals or drives that the agent may need to attend to, This descrivbes the intial roots of the reactive plan hierachy
6. Select a first behavior to implement.

The initial decomposition process can be seen to be goal-directed, with the initial task being the high-level goal that it intends to achieve, and the sequences of actions representing the execution required to achieve this goal. Several other AI methods bear similarities to this, such as Hierarchical Task Networks [**?**] and Goal Trees [**?**]. Figure **??** shows an example of a **HTN**, and it can be seen to bear the characteritics of **BOD**'s task decomposition. We shall soon present and see that Behavior Trees may also be designed to exhibit such goal-directed behaviors.



(a) Top level task **Hour** decomposed into sequence of subtasks and actions

(b) Sub-task **Get Food** is further decomposed

Figure 2.16: Hierarchical Task Network

The main motivation for such a goal-direct planning architecture arises from the second of two general philosophies of game AI [**?**]. The first holds the belief that AI has to be strictly controlled using techniques such as finite automata like Finite State Machines and scripting. The reason for this is to give the AI developers a close control over the direction of the AI, what it is enabled to do in each state or situation and ultimately was to enable predictable behavior so as to ease the process of testing and debugging.

The second philosophy, and the one which **BOD** pertains to, reasons that it is essentially impossible to adequately dictate and desribe every possible every possible action from every possible state. We have shown earlier that this is indeed true for Finite State Machines. Thus, it proposes that it is better to develop a reasoning AI that understands the basic rules of the game and makes its own decision based on the world as it perceives it, in hope that the emergent behavior would be more effective and believable than what could be specified by hand.

Iterative Development

The development process for the **BOD** methodology involves the following steps:

1. Select a part of the specification to implement next.
2. Extend the agent with that implementation:
    - code behaviors and reactive plans, and
    - test and debug that code
3. Revise the current specification

2.3.2 Goal-directed Reasoning

With both *task decomposition* and the *iterative development* process, we now highlight the approach undertaken in this project, using the **BOD** methodology to dictate the designing of the Behavior Trees for DEFCON's AI. In this section, we cover terms and ideas introduced in Section **??** and **??**. We introduce the idea of *goal-directed reasoning*, which is used in tandem with **BOD** for the construction of behavior trees

Approach

The first thing to identify is the top-most goal that we want to achieve for our AI. Quite simply put, this top most goal would be to *Win the Game*,

and then we identify what are the sequences or groups of actions that this top level goal can be decomposed into, before recursively breaking down these subgoals down to primitive tasks such as `Actions` and `Conditions`.

This is an example of *Goal-Directed Reasoning*, which is the opposite of another common methodology called *Forward Reasoning*, which works by taking all the possible actions from a state, and applying each one to generate several subsequent states. This process continues until a target destination state is reached. This, however, is extremely cumbersome for a real-time strategy game such as Defcon, as the complexity of states and the unpredictibility of actions makes this approach prohibitive [**?**].

Goal-Directed Reasoning does the opposite, we identify a target goal state and then identify moves that might generate this goal state. The process is repeated for all predecessor states until the initial state is reached. An example of a planning procedure that employs this form of reasoning is the STRIPS approach [**?**].

In Figure **??**, we have identified our top most goal of our Behavior Tree to be, as we mentioned, *Win the Game*. This is decomposed into 5 sub-goals which are named *Defcon 5* to *Defcon 1*. What this means is that, in order to achieve the top level goal of *Win the Game*, we have to essentially *Behave Optimally in Defcon 5*, *Behave Optimally in Defcon 4*, ... and so on. This covers the first 2 steps of the *initial decomposition* process introduced in Section **??**. Note that we group these set of Defcon goals with a `Selector` node rather than a `Sequence` node, because we want the AI to choose the associated Defcon plan for each Defcon state the game is in. Using a `Sequence` would make the Behavior Tree fail once the game has proceeded to Defcon 4, and is not the correct behavior we wish to exhibit.

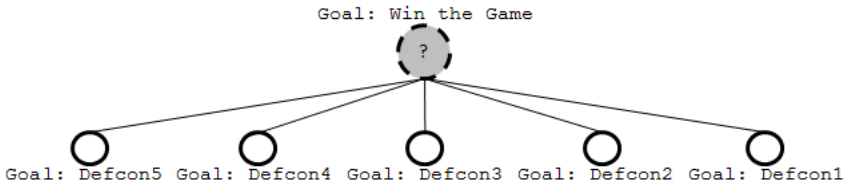


Figure 2.17: The Goal-Oriented Behavior Tree with its initial decomposition

Now, taking a look at these sub-goals which we have identified, we identify the need for a sensor primitive, or in Behavior Tree terms, a `Condition` task node, that performs a conditional check of the current Defcon state of the game. From here, we group these different sub-goals by registering them with Lookup Tables, which as we introduced, are used to group and mod-

ularize the Behavior Trees into a hierarchy, and we may later use `Lookup Decorators` to locate these trees to enable reusability. This essentially covers steps 3 and 4 in Section **??**. Figure **??** shows the original tree now including the `Condition` nodes for each branch, as well as the grouping of them into a Lookup Table. Do note that the proposed Behavior Library essentially is the collection of Lookup Tables that we possess.

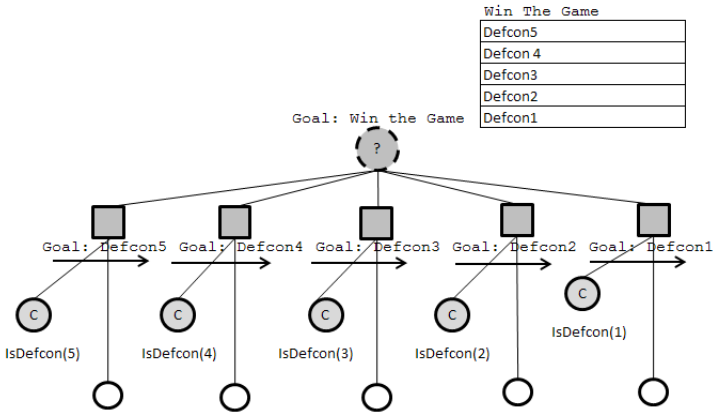


Figure 2.18: Each sub-goal introduces a set of primitives and other subgoals

Finally, as we mentioned, the use of Selectors was required to accomplish steps 5 and 6. Priorities are assigned to each branch corresponding to each Defcon state, and we assign a higher priority to *Defcon 5* and continue in a decreasing order towards *Defcon 1*, since at the start, we always begin in Defcon 5. This concludes the initial decomposition of the task *Win the Game*.

Subsequent Iterations

With the completion of the initial decomposition, we essentially test to ensure that all new introduced nodes are working correctly before moving on to the next phase of the iteration. A similar process is then replicated for each subgoal, decomposing the task further into subtasks until we reach a point when further decomposition is no longer permissible, which in the case of Behavior Trees, is when we have reached the leaves of the tree where only primitive task nodes exist. We now present the complete behavior tree of one of the branches, namely *Defcon 5*, which was constructed for the purpose of the project.

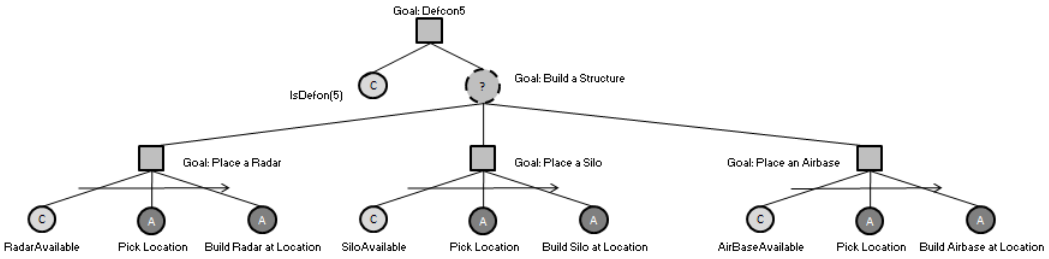


Figure 2.19: Complete Behavior Tree for Task: Defcon5

Figure **??** above shows the complete behavior tree for the *Defcon5* task. As we can see, the entire task consist of a sequence of actions. First, it checks whether the current DEFCON state of the game is DEFCON 5 using **Condition** *IsDefcon(5)*. Next, it picks a structure type to place within the game world, leaving the choice randomly to the `Selector` *Build a Structure*. This task is broken down into 3 sub-tasks.

In order to place a structure, what involved is a `Sequence` first checking if there exist any available sturctures of the chosen type to be placed with a `Condition`. Next, it performs as `Action` to choose random coordinates to place the structure. Finally, upon picking the location, the AI proceeds to place the structure at the designated location with yet another `Action`.

During the designing of the task *Defcon5*, we have identified both the composites required, but more importantly, the primitive task nodes that are required to be constructed. These nodes, *IsDefcon*, *PickLocation* and *Build Structure at Location* were identified as required primitives - which were then constructed and added into the list of actions and primitives for the entire AI. This process was repeated for each of the DEFCON tasks, and resulted in a cumulatively constructed library of primitive `Actions` and `Conditions`. The complete tree for the entire AI, covering all 5 DEFCON tasks can be found on page **??** of the Appendix.

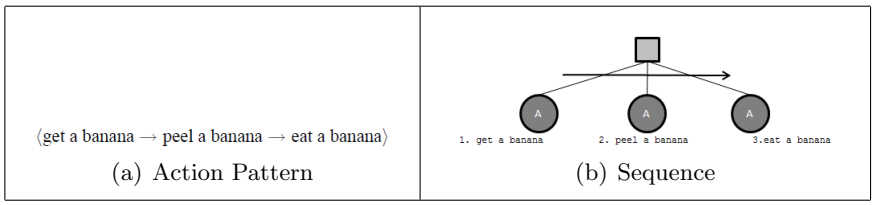
2.3.3 Reactive Planning

In the previous section, we have described task decomposition in **BOD** and one importance of the ability to decompose our tasks is that by decomposing them down to simple enough subgoals, there arises the possibility of interleaving actions and tasks. This is possible for DEFCON, as a real-time strategy game due to its unpredictability of the game state due to the dynamic set of choices that both an AI and human player may execute. Overcoming this unpredictability by having this form of task decomposed interleaving is a form of **Reactive Planning** [**?**]. This reactive planning is described in **BOD** by *Action Selection*, and shows advantage over traditional architectures without similar support for reactive planning such as the Subsumption Architecture or Agent Network Architecture [**?**]. We shall cover plan elements which **BOD** introduces to support *action selection* in such reactive plans, namely **Action Patterns** and **Competences**, and subsequently draw their respective correspondence to Behavior Trees and how they implement something similar to them.

Action Patterns

An *action pattern* describes a sequence of primitives, where the primitives are either actions or sensory actions, and its usefulness include:

- Aiding the AI designer to keep the system as simple as possible, communicating clearly the expected behavior of the plan segment
- Allows for speed optimization of elements that reliably run in order.



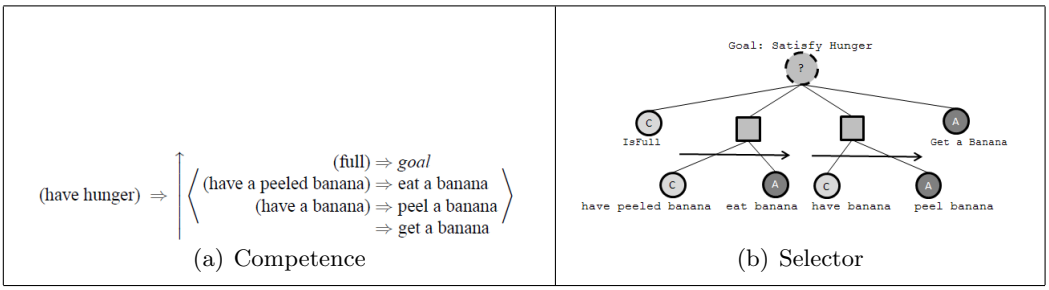
(a) Action Pattern (b) Sequence

Figure 2.20: Correspondence between Action Patterns and Sequences

We can see that *action patterns* can be represented in Behavior Trees as *Sequences*. Figure **??** illustrates the this correspondence.

Competences

Competences attempts to drop the temporal ordering of its group of actions in order to allow any of its elements to fire when in a perceptually equivalent context. Rather, a prioritization is assigned to each child action, shown in increasing order in the direction of the arrow. Preconditions are used to determine the executability of each element.



(a) Competence (b) Selector

Figure 2.21: Correspondence between Competences and Selectors

Figure **??** shows the correspondence between *Competences* and `Selectors`. It is worthwhile to note that the `Selector` in Figure **??** is a `Priority Selector` - where it places a priority values of decreasing value from left to right on its children task nodes.

Reactive planning for DEFCON fleets

Reactive planning provides a means for handling changes in the game world which cannot be pre-determined. One area in which this arises is in the coordination of the movement of fleets in DEFCON. For example, consider a fleet's movement from one point to another - during the length of its journey, an event, such as the detection of an enemy unit in the vicinity, might occur. We might want to be able to handle this by attempting to avoid the enemy by readjusting our route in such a way as to avoid the enemy.

Some AI approaches may find this problem difficult to handle. For example, the AI presented in Robin's thesis [**?**] does not actually handle this explcitly, instead, it the fleet will continue to move to its target destination as dictated by its plan. The unpredictability of such occurences makes this very difficult to plan for, and we require a solution that can perform a replanning at run-time. This is where reactive planning comes in useful.

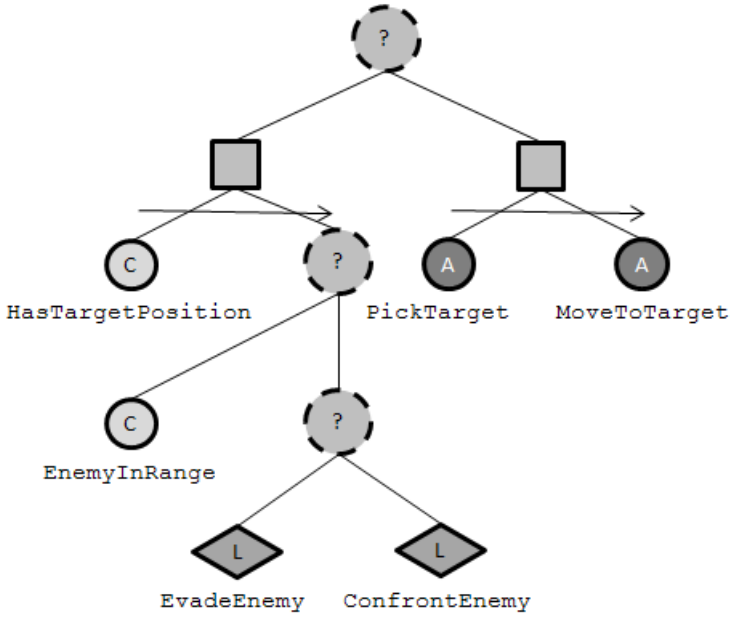


Figure 2.22: Behavior Tree to coordinate Fleet movement

In Figure **??**, we present a behavior tree used to dictate the movement of a fleet. This tree can be seen as a replacement for the *PatrolOrIdle* node for the behavior tree defined for *Defcon4* on page **??** of the Appendix. At the top, we have a `Priority Selector`[8], with two branches. In the left branch, we first check if the active fleet unit has a target position set. If not, the node fails, and the `Selector` chooses the right branch's node which picks a target destination and orders the fleet to move.

Considering the left child's right branch now, we first check if any enemy has been detected within its radius in *Enemy In Range?*. If none, no further action is required. But if we do detect an enemy, we then select whether to *evade* or *confront it* with yet another selector.

The behavior tree provides provides a way for a Fleet to proceed towards a target destination, but at the same time, allows it to handle the occurences of events easily by means of this form of reactive planning. For example, in the case where the the unit does not yet have a target destination, the sequence of actions would be:

```
HasTargetPosition(Fail) -> PickTarget-> MoveToTarget
```

[8]We assume the convention that priority values of the children nodes decrease from left to right

In the case where the fleet already has an active target, but no enemy is detected, we have:

```
HasTargetPosition(Success) -> NoEnemyInRange(Sucess)
```

And in the final case, the fleet has an active target destination, and an enemy is detected during this time frame, we have:

```
HasTargetPosition(Success) -> NoEnemyInRange(Fail)-> Evade
```

Evade and *Pursue* are assumed to have implementations defined in their respective behavior trees. Also, in this case, we assumed that *Evade* had the higher priority, perhaps due to the AI's overall state of being defensive rather than offensive.

2.4 Genetic Algorithms

Designing an AI for a game is a complex task [**?**], more so when developing one for a real-time strategy game like DEFCON, with its ever-changing game state, objects and their respective actions. One approach would be to hand-craft the AI based on the AI designers idea of what befits an appropriate AI, and such an approach would prove to be rather time-consuming in order to eventually produce an AI that is satisfiably robust. The AI designer would require to run through several iterations, each time changing or adding new states or variables to improve upon the AI of its previous iteration. The Introversion Bot of Section **??** is one such example.

A second approach would be to introduce the ability of the AI to *learn to play the game* [**?**] via machine learning techniques. Machine Learning techniques have been successful in areas such as robotics [**?**], and Robin's bot introduced in Section **??** made used of *case-based reasoning*, a form of machine learning technique

In this section, we introduce *genetic algorithms* as our approach towards machine learning. We begin by formally introducing the terms and concepts of genetic algorithms. We then propose how they may be used as another key component of the development of DEFCON's AI for the purpose of this project.

2.4.1 Methodology

Genetic algorithms were inspired by the biological process of as *evolution*. It serves to find near optimal solutions to complex non-linear problems [**?**], one such example being the designing of an appropriate AI for DEFCON. The approach usually begins by starting our with an initial population of solutions to the problems. These solutions are rated on how optimal they are, termed its *fitness*, and subsequent populations are produced by a set of *genetic oeprators*. This is allowed to repeat over numerous runs, or *generations*, until a satisfactory performance is achieved. Informally, this can be desribed as a series of steps [**?**] [**?**]:

1. Create a first-generation of the population of random organisms
2. Test them on the problem we are trying to solve, and rank them according to fitness. If the best organisms have reached our performance goals, stop.
3. We produce the offspring organisms for the next generation via any combination of the following methods
    - Carrying forward offsprings from the previous generation
    - Producing new offsprings by genetic operators such as crossovers and mutation to the fittest individuals as identified in step 2
    - Creating entirely new offsprings to introduce variety and prevent local maxima convergence
4. With this new set of individuals, we then repeat step 2.

Implementation

The first step proposed above is to create the first-generation of the population of random organisms. In this case, each organism corresponds to each implementation of our AI bot using Behavior Trees. Each implementation can vary from one another by modifying the topology of the Behavior Trees or by maintaining the same topology and structure, but using a different set of values for the nodes in the tree.

Genetic algorithms can be applied both to find an optimal topology or optimal values for these nodes in the Behavior Tree. We first discuss the latter approach, where we present an approach to evolve our AI on one area of interest - the *Optimal Placement of Silos*. We hope our AI to identify the best Silo placement positions in order to maximise its chance of success. We thus focus our attention to the *Defcon5* branch of the our AI Behavior Tree which we constructed using the **BOD** process described in Section **??**.

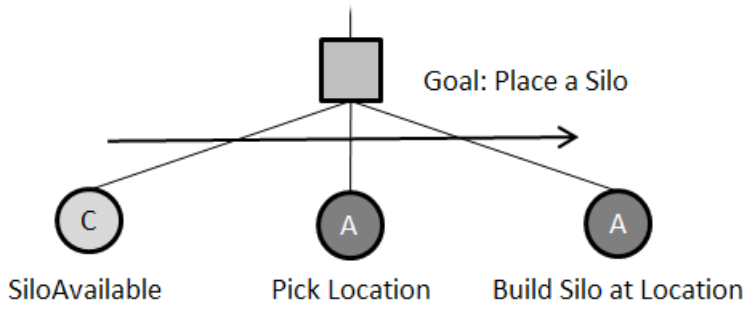


Figure 2.23: The Place Silo branch of Defcon 5's Behavior Tree

Figure **??** above is from the middle branch of the *Build a Structure* behavior tree from Figure **??**. With this original implementation, the AI attempts to place a Silo at randomly picked coordinates, calculated by the `Action` node *Pick Location*. Since the placement of structures for each player is only limited to their specific territory, there will be a chance that even if the `Condition` that a Silo is available to be placed, that the selected coordinates might end up being invalid. Thus, by running multiple runs of the game, we are able to generate a log file of valid and invalid coordinates for Silos as seen in Figure **??**.

```
1   InvalidLocation Silo    -171.600006     76.900002
2   InvalidLocation Silo    -101.300003     95.699997
3   InvalidLocation Silo    -162.500000     -18.000000
4   InvalidLocation Silo    105.199997      -3.900000
5   InvalidLocation Silo    -77.900002      27.000000
6   InvalidLocation Silo    26.299999       88.800003
7   ValidLocation   Silo    112.300003      29.200001
8   ValidLocation   Silo    81.400002       31.000000
9   ValidLocation   Silo    114.500000      41.000000
10  ValidLocation   Silo    56.500000       31.600000
11  ValidLocation   Silo    72.300003       37.000000
12  ValidLocation   Silo    53.799999       24.000000
13  ValidLocation   Silo    102.199997      41.900002
14  ValidLocation   Silo    83.300003       21.200001
15  ValidLocation   Silo    109.400002      39.500000
16
```

Figure 2.24: Log of Valid and Invalid Silo placement positions for South Asia

Fitness

Fitness is a numerical measure of the performance of the organisms concerning the problem at hand. In this case, we may assign a simple fitness measure of each coordinate position by determining whether it was a valid or invalid placement location. However, it is important to consider several fitness functions, since incorrect convergence might occur [**?**]. Some proposed fitness functions for the *Placement of Silos* are,

- The amount of time the Silo managed to remain in the game, meaning that it was not destroyed.
- The number of enemy units it managed to destroy
- The number of successful nuclear hits on enemy buildings or units

These are just several proposed fitness function, and it would be interesting to investigate the results of applying each fitness function in this problem domain. There is also room to perhaps consider a combination of these fitness functions, using a simple *linear normalisation* method [**?**], but for the purpose of explanaining the following section on genetic operators in this report, we shall assume that only 1 fitness function is adopted.

2.4.2 Genetic Operators

Genetic operators are the set of operators that generate successors for each generation, producing a new population set for the next evolution iteration.

Selection

Selection involves picking a subset of the existing population to carry forward to the next generation. Three common forms of selection [**?**] are

- Fitness Proportionate Selection
- Tournament Selection
- Rank Selection

In *fitness proportionate selection*, individuals from the population are selected based on the ratio of its fitness to the fitness of the other individuals present in the current population. Thus, the probability of selecting an individual `x` from a population of `n` individuals is described by the equation,

$$P(x) = \frac{Fitness(x)}{\sum_i^n Fitness(x_i)}$$

In *tournament selection* involves first choosing at random two individuals from the existing population. The first is given a propability of `p`, and the second `(p-1)`.

In *rank selection*, the individuals are sorted according to their fitness. The probabilty of selecting an individual is then proprtional to its rank in this list, rather than its fitness.

Recombination

In recombination, two new individuals are generated for the next generation from two individuals from the existing population, termed as their parents. The aim of recombination is to allow the characteristics of the parents to be somewhat combined together, and thus, if we selected two parents with a high fitness, we would expect the offsprings to exhibit a hybrid of their characteristics and ideally, and improved fitness.

We begin our explanation of recombination with an example, on the same topic of discussion as for the placement of silos. For the purpose of this explanation, all `Action` nodes are assumed to be `PlaceSilo(x,y)` nodes, where the arguments are labelled at each node. Figure **??** is an example of *single-point crossover*, where the dashed lines on the parent trees identify the point at which the recombination occured. From the diagram, it is clear to see that each offspring has a mixture of nodes from their parents.

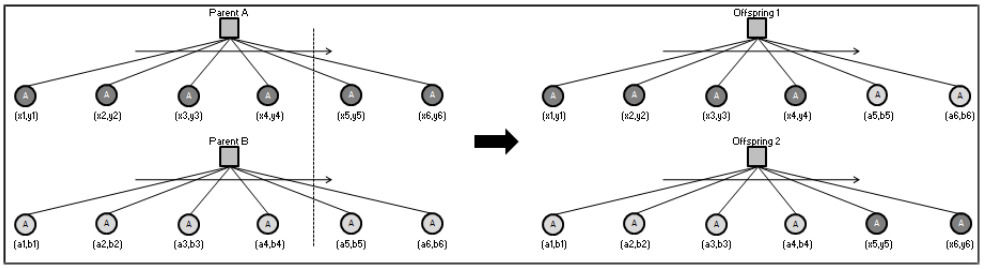


Figure 2.25: Single-Point Crossover

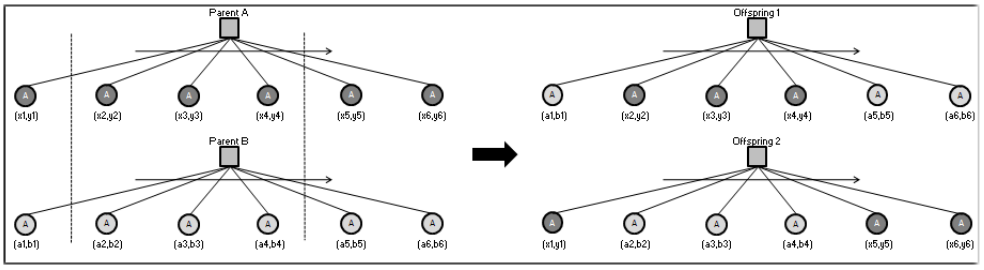


Figure 2.26: Two-Point Crossover

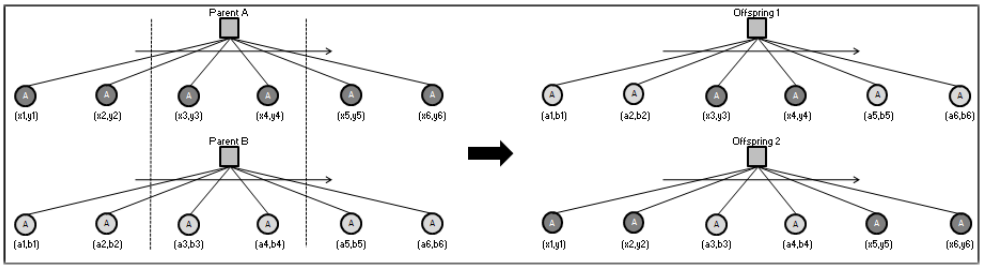
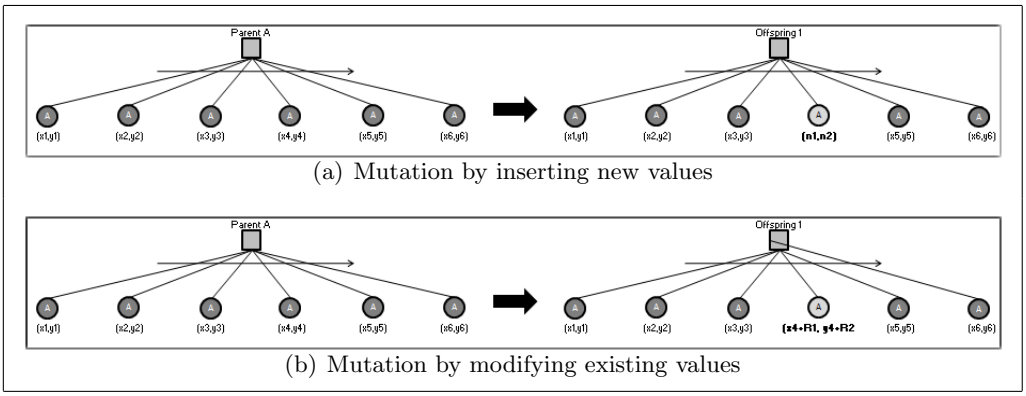


Figure 2.27: Uniform Crossover

Figure **??** shows offsprings generated when *two-point crossover* was used. In this diagram, recombination occurs at two points, resulting in a greater mixture of nodes being passed from parent to offsprings. Figure **??** illustrates *uniform crossover*, where the nodes are combined uniformly from the two parents.

Mutation

Mutations are employed to produce unpredictible random changes into individuals, modifying its attributes, so as to allow the individual to vary across generations. In this case, mutations can be useful to reintroduce placement coordinates that might have been lost or extinct due to being part of an unfavourable individual early in the evolution process. Mutations are also a good way to prevent against a local convergence, which tends to happen in small populations which very quickly become saturated with individuals that are locally optimal.



(a) Mutation by inserting new values

(b) Mutation by modifying existing values

Figure 2.28: Mutations in Action nodes of Behavior Tree

Figure **??** demonstrates mutation occuring for a node in the behavior tree. The coordinate values might be entirely regenerated, or perhaps, modified slightly. Figures **??** and **??** illustrate this respectively.

2.4.3 Evolution Strategies

The previous sections have introduced genetic algorithms, genetic operators and how they may be used for the evolution of Bahavior Trees. We presented the evolution of the *Placement of Silos* behavior trees using such an approach. We now present further areas of investigation of this evolution of behavior trees for the purpose of this project.

- Evolving the time of attacks for both units and structures
- Evolving the composition of fleets
- Evolving the placement and movement of fleets

Evolving time of attacks

One key characteristic about the game of DEFCON is that the timing and coordination of an attack is crucial to winning [**?**]. A player who performs a well-coordinated and timed attack is more likely to do better than one who did random attacks sporadically throughout the game. We plan to investigate if, by the process of evolving these behavior trees, we are able to allow the bot to learn how to coordinate and time its attacks to maximise its score in the game.

Evolving composition of fleets

In DEFCON, fleets present an area of significant interest. Firstly, the composition of fleets is highly variable, and we may investigate the implications of having different fleets of compositions.

Evolving the placement and movement of fleets

Another interesting would concern the placement and movements of fleets. As mentioned in Section **??**, some counties have two choices of sea territories to place fleets in. This, together with the target destination, has an impact as it may result in different lengths of time in order for a fleet to reach its target destination.

Chapter 3

Plan

This chapter provides information on the current state of the project at the approximate point of submission of this report. The chapter is divided into two sections:

- **Project Status**: Covers the current status of the project, implementation details and current AI ability.
- **Plan for Remainder of Project**: Covers plans for the remaining portion of thr project, classified in terms of weeks.

3.1 Project Status

3.1.1 Behavior Tree Design

For the purpose of this project, there was the necessity to develop a Behavior Tree data structure. Our implementation makes use of the **Composite Design Pattern** and the UML class diagram in Figure **??** shows our implementation.

- The nodes of a Behavior Tree are of the *abstract class* `TaskNode`.
- Two derived classes, `Primitive` and `Composite`, inherit from the `TaskNode` class.
  - Concrete classes `Action` and `Condition` are derived from `Primitive`
  - Concrete classes `Selector`, `Sequence` and `Decorator` are derived from `Primitive`

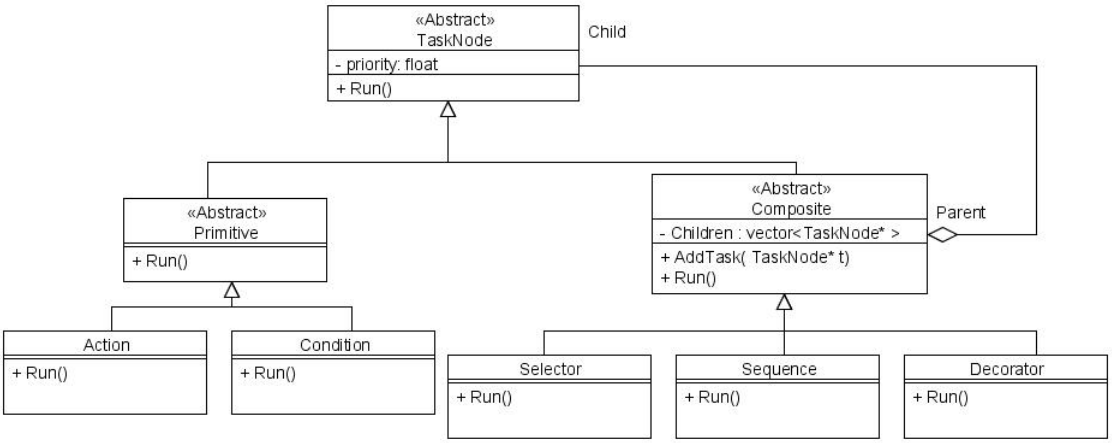


Figure 3.1: UML Diagram for Behavior Trees

3.1.2 System Design

The entire system consists of two main components - the DEFCON API and the Behavior Trees. As we can see from Figure **??**, the API's Bot class makes use of the Behavior Trees in order to execute AI logic and behavior. At the same time, the Behavior Trees need to be able to access the API calls to perform checks on the game state as well as executing commands.

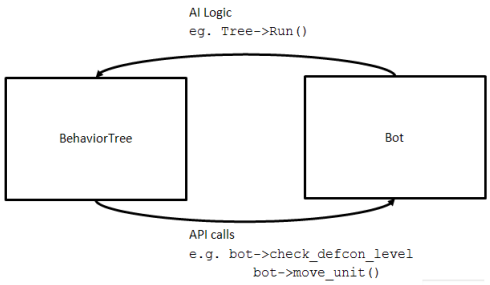


Figure 3.2: Dependence of both the Behavior Tree and the Bot's API on each other

Thus, in order to adhere to one of the core depency principles of having non-cyclic dependencies between classes, we have made use of the *abstract client pattern*, which is illustrated by Figure **??**.

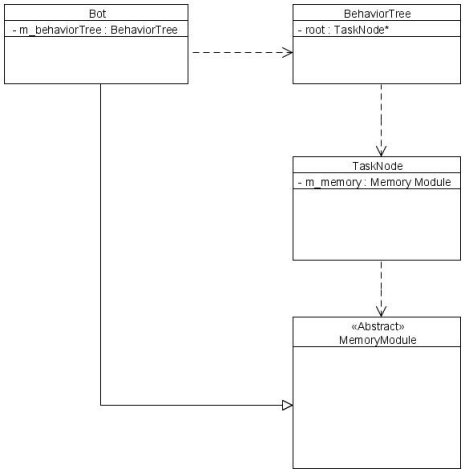


Figure 3.3: Abstract Client Pattern to prevent cyclic dependcies

In this case, the Behavior Trees make use of what we term as a **Memory-Module** abstract class, and this essentially paves the way to to modularizing the Behavior Trees, make them reusable for any other application that may require it. For example, in the event a new game wanted to make use of the Behavior Trees, it would essentially ensure expose its API as a derived class of the MemoryModule abstract class, and create TaskNodes specific to that particular game.

3.1.3 AI Bot abilities

At the point of writing, the AI is capable of playing the game of DEFCON, and more specifically, exhibits the following abilities:

1. Ability to place all 3 structure types - namely **RadarStations**, **Silos** and **AirBases**
2. Ability to create and place fleets of any of the 3 naval forces types - **Battleships**, **Carriers** and **Submarines**
3. Fleets are able to select a target destination and orders are sent for it to proceed to destination automatically.
4. **Fighters** are launched to provide scouting at random locations
5. Silos convert state to **Launch Mode** upon activation of DEFCON5, and target opponent cities.

These abilities cover the main abilities, and allow the bot to perform actions throughout all 5 DEFCON levels. Target coordinates are now randomly picked by the bot, and in the next section we cover plans to improve upon this bot by evolving its behavior trees.

For reference, the behavior trees for each DEFCON level can be found in page **??** of the Appendix.

3.2 Upcoming Plans

The current abilities of the bot exhibits a naive players' approach to the game of DEFCON. The remainder of the project focuses heavily on improving upon this bot. More specifically, the evolving of the bots existing behavior trees, in hope of allowing the bot to gradually adapt and learn to play the game better.

A backlog of tasks pertaining to this project is included in page **??** of the Appendix - which cover completed tasks, tasks in progress and tasks planned for the future. Here, we break down the plan into blocks of weeks, corresponding to the Department of Computing's calendar.

Term & Week	Dates	Description
Term 2 Week 06	15 FEB - 21 FEB	Begin evolution of Silo Placement. Submission of Interim Report.
Term 2 Week 07	22 FEB - 28 FEB	Evolution of Silo Placement, working with Robin Baumgarten to refine API to allow collection of required data from DEFCON. Explore 2 fitness functions
Term 2 Week 08	01 MAR - 07 MAR	Evolution of Fleet movement coordination, aim to get Fleets to identify "ideal" positions to place themselves for synchronised attack
Term 2 Week 09	09 MAR - 14 MAR	Evolution of Aerial movement coordination. Aim to get Fighters to maximise fitness function or locating enemies, and Bombers to synchronise attacks
Term 2 Week 10	10 MAR - 21 MAR	Coordinating evolved existing behaviors, aiming to combine abilities and provide global evolution of the AI
Term 2 Week 11	22 MAR - 28 MAR	Collect preliminary statistics to assess current bot's abilities
Easter Break	29 MAR - 26 APR	Study break for examinations, code tweaking, refining and catch up on any implementation that was not able to be completed
Term 3 Week 01	26 APR - 02 MAY	Examinations
Term 3 Week 02	03 MAY - 09 MAY	Examinations
Term 3 Week 03	10 MAY - 16 MAY	Examinations
Term 3 Week 04	17 MAY - 23 MAY	Assuming implementation working and performing desirably, begin work on reactive planning for fleet coordination
Term 3 Week 05	24 MAY - 30 MAY	Work on reactive planning for Aerial coordination and movements
Term 3 Week 06	31 MAY - 06 JUN	Consolidating report, findings and evaluation
Term 3 Week 07	07 JUN - 13 JUN	Consolidating report, findings and evaluation
Term 3 Week 08	14 JUN - 20 JUN	Final Report Submission
Term 3 Week 09	21 JUN - 26 JUN	Presentation

Appendix

List of Decorators

This presents a non-exhaustive list of proposed **Decorators** types, as suggested by Alex J. Champandard [**?**].

- **Filters**: These prevent a behavior from activating under certain circumstances.
  - Limit the number of times a behavior can be run
  - Prevent a behavior from firing too often with a timer
  - Temporarily deactivate a behavior by script
  - Restrict the number of behaviors running simultaneously
- **Managers & Handlers**: These are decorators that are responsible for managing whole subtrees rather than single behaviors.
  - Deal with the error status code and restart planning or execution
  - Store information for multiple child nodes to access as a blackboard
- **Control Modifiers**: These decorators are used to pretend that the execution of the child node happened differently
  - Force a return status, e.g. always fail or always succeed
  - Fake a certain behavior, i.e. keep running instead of failing or succeeding
- **Meta Operations**: Such decorators are useful during development, and can be inserted into a tree automatically when needed.
  - Debug breakpoint; pause execution and prompt the user
  - Logging; track this node execution and print to the console

Project Backlog

ID	Title	Description	Priority	Estimated	Status	Comments
1	Behavior Tree Structure	Behavior Tree structure that includes the 4 basic nodes, Actions, Conditions, Selectors and Sequences. The composite pattern should be used, and the Behavior Tree structure should be easily be reused for any application, not just specifically for DEFCON.		n/a	Completed	n/a
2	Defcon 5 basic placement behavior	All 3 types of buildings - Radas, Silos and Airbases should be placed based on a random policy. The placement should be independent of the choice of country that the AI is assigned.		n/a	Completed	n/a
3	Defcon 4 Fleet Composition and Placement	Fleets should be randomly placed, and composed, during Defcon4 into allocatable sea regions. The fleet composition should be extensible, allowing different number of units, and also be able to vary the the unit composition to allow a mixture of fleet types and numbers. For the initial implementation, fleets of specific types are created, each with the maximum number of allocatable units - 6		n/a	Completed	n/a
4	Defcon 3 AirOrders and Fleet movement	Airbases should be able to send fighters to perform scouting duties. Fleets are to be able to move randomly around the map to explore and perform scouting. No attacks are performed, except if retaliating.		n/a	Completed	n/a
5	Defcon 2 Fleet and Air Attacks	Fleets should attack enemy fleets or buildings, same for bombers/fighters. The enemies are randomly chosen	Medium	n/a	In Progress	Enemies are currently not being selected explicitly.
6	Defcon 1 Silo Attacks	Silo attacks on random buildings or cities	Medium	n/a	Completed	n/a
7	Memory Module Perception	Memory Module should contain a perception of the entire world and all of it's objects, units and activity. This will make it easier for the Behavior Tree to perform checks and actions, rather than making API calls directly from the nodes. This should allow the Behavior Tree to check for conditions easily by working with the memory module.	High	n/a	In Progress	Progress for this is made iteratively.
8	Load/Save Function	Behavior Trees should be stored in an XML format and read when the game is launched, at runtime, where the behavior tree is constructed. This will allow having multiple trees, localised testing and easier BT management	High	10-Feb-09	Completed	A basic implementation should be achieved that allows the behavior trees to be saved, visualised (if needed), loaded. A further implementation allowing the use of Decorators to link up multiple behavior treees into one would be the natural next step once this has been done.
9	Convert existing Trees	Existing Behavior Trees for Defcon5 to Defcon1 should be converted to XML format, and should be working together with the save/load function	High	10-Feb-09	In Progress	This would serve as a test to ensure that the save/load abilities are working correctly. This process occurs throughout the course of the project - the Estimated date depicts the date to which at least DEFCON 5's behavior trees have been completed
10	Script for statistics	Script to run multiple Defcon games continuously, allowing for the evaluation of the AI's performance. This should be extensible to information of the game, eg. Winning score, no. of planes lost, no. of units lost, to be collected. This would be used to evaluate the fitness of the behavior trees	High	n/a	In Progress	This is an iteratively developed process since different types of data are collected based on the different areas of evolution. Currently working with Robin to extend API to be able to harvest required information.
11	State/Strategy Implementation for Nodes	Extend the current behavior trees to associate with each high level behavior, a state, in which the behaviors are to receive a higher priority. The nodes should have an associated priority, which would allow selectors to perform priority selection. This would also involve the organisation of states-behavior relationships that would reset the priorities	High	10-Feb-09	Completed	Basic implementation into core Behavior Tree implementation expected
12	State/Goal directed placement of structures	The 3 buildings - Radars, Silos and Airbases should be placed based the strategies determined from the Strategy Policy. This will provide the basis of meaningful placement of the buildings, allowing them to maximise the fitness function values. This will be the basis of evolving the placements later.	Low		Not Started	Tentative plans for coordinating the placement of structures based on distance from cities, distance between one another and territorial coerage. These distances are to be set as variables to be used for an evolutionary approach to determining the optimal values bsed on a local fitnes function eg. number of enemies located, total area coverage
13	State/Goal directed Composition of Fleets	Research and implement the policies that determine the composition of fleets. Fleet composition can be evolved based on its survival rate (no. of remaining units) and attack rate (no, of destroyed enemies)	Low		Not Started	n/a
14	Fleet movement coordination behavior	Implement the 5 behaviors as proposed by Robin into Behavior Trees, associating them with strategy policies. This will determine the movement and location of the fleets. This would consist of 'Idle', 'Await Opponent', 'Avoid Opponent', 'Move Direct' , 'Move Intercepting Opponent'	Low		Not Started	n/a
15	Evolving the placement of structures	Multiple games will be run, with the placement of structures evaluated based on a fitness function. This occurs on two levels - the first level would involve finding an optimal placement for each type of building on its own - the second level will involve finding the optimal positions to be placed for all 3 structures.	High	24-Feb-09	In Progress	Placement of Silos is the priority. Plans to evolve the Silo placement based on the fitness functions (Lifetime, Defense Value and Attack Value)
15	Evolve the movement of Fleets	Multiple games will be run, with the aim of allowing fleets to determine the best position to move to move to in order to proceed to attack the enemy cities or structures.	High	24-Feb-09	In Progress	Placement of Silos is the priority. Plans to evolve the Silo placement based on the fitness functions (Lifetime, Defense Value and Attack Value)

DEFCON Behavior Trees

The tree diagrams here were automatically generated from the code. The nodes do not take the shape of the convention used in the report, instead nodes are prefixed to indicate the node types. The prefixes are:

- **ACT**: `Action` node
- **CON**: `Condition` node
- **SEL**: `Selector` node
- **SEQ**: `Sequence` node
- **PSEL**: `Priority Selector` node

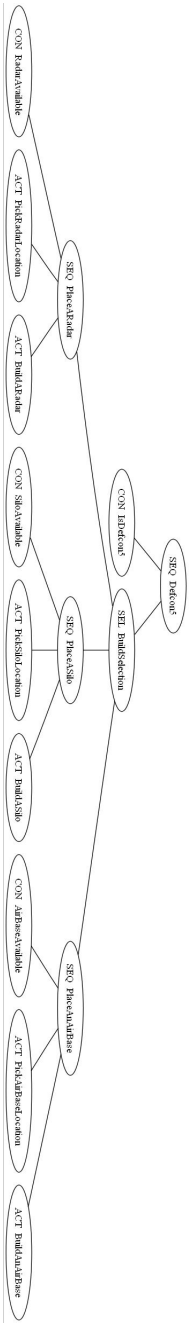


Figure 3.4: Defcon 5

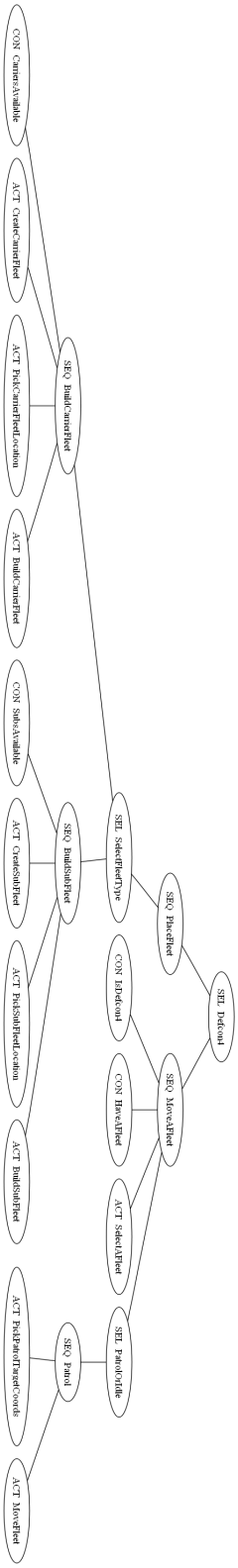


Figure 3.5: Defcon 4

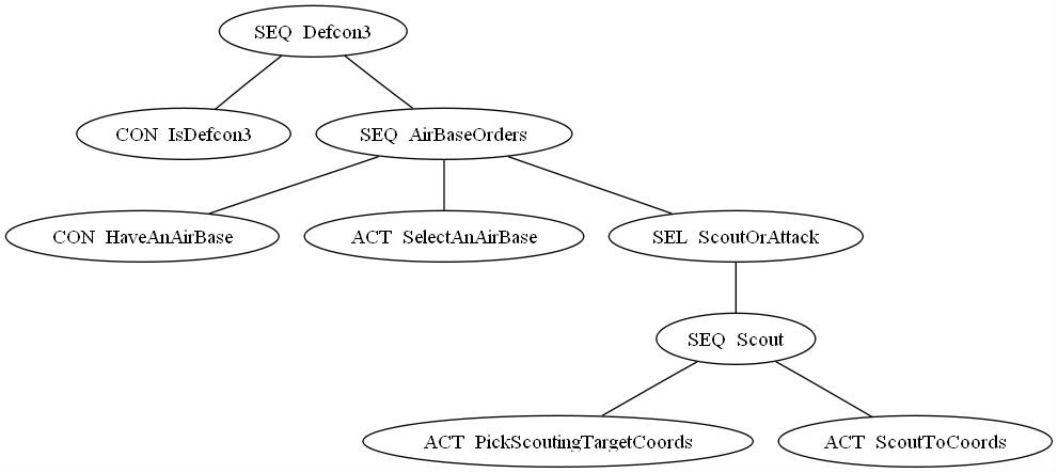


Figure 3.6: Defcon 3 and Defcon 2

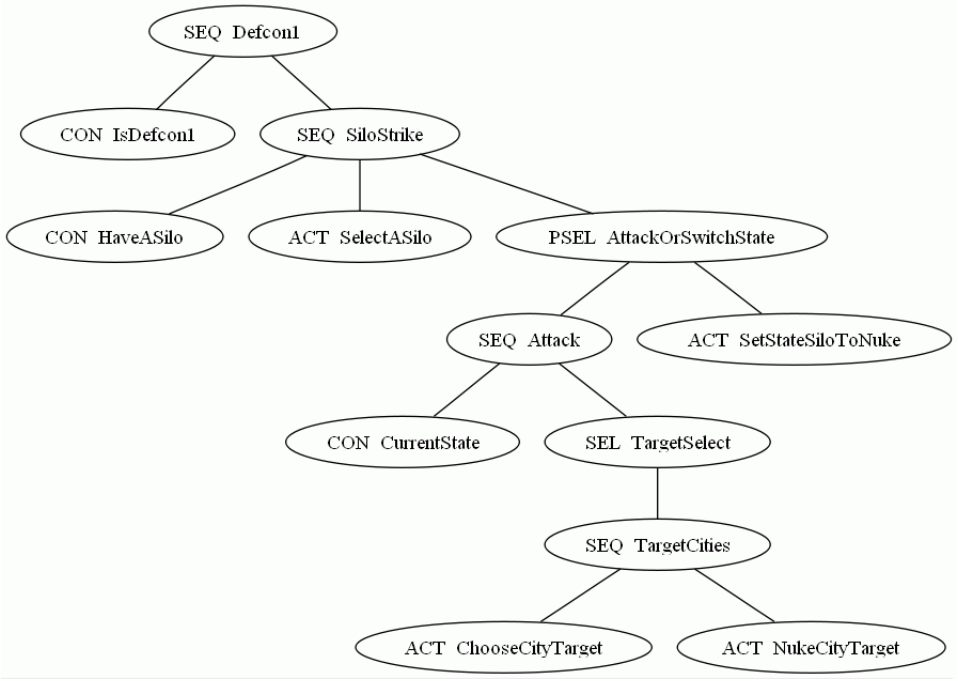


Figure 3.7: Defcon 1

Bibliography

[BC07] Robin Baumgarten and Simon Colton. Case-based player simulation for the commercial strategy game defcon. *CGames, 2007*, 2007.

[BFGM06] Michael Bowling, Johannes Furnkranz, Thore Graepel, and Ron Musick. Machine learning and games. 2006.

[Bry03] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. *Lecture Notes in Computer Science*, 2592, January 2003.

[BT03] Christian Bauckhage and Christian Thurau. Exploiting the fascination: Video games in machine learning research and education. 2003.

[Cha07a] Alex J. Champandard. Behavior trees for next-gen game ai part 1. `http://aigamedev.com/videos/behavior-trees-part1`, 2007.

[Cha07b] Alex J. Champandard. Understanding behavior trees. `http://aigamedev.com/hierarchical-logic/bt-overview`, 2007.

[Cha07c] Alex J. Champandard. Using decorators to improve behaviors. `http://aigamedev.com/hierarchical-logic/decorator`, 2007.

[Dil06] Kevin Dill. Prioritizing actions in a goal-based rts ai. *AI Game Programming Wisdom*, 3, 2006.

[Gar06] Sergio Garces. Extending Simple Weighted-Sum Systems. *AI Game Programming Wisdom 3*, 2006.

[Har06] Vernon Harmon. An economic apporach to goal-directed reasoning in an rts. *AI Game Programming Wisdom*, 3, 2006.

[Isl05] Damian Isla. Managing complexity in the halo 2 ai system. In *Proceedings of the Game Developers Conference*, 2005.

[Joh06] Geraint Johnson. Goal trees. *AI Game Programming Wisdom*, 3, 2006.

[JOS02] Yaochu Jin, M. Olhofer, and B. Sendhoff. A framework for evolutionary optimization with approximate fitnessfunctions. *Evolutionary Computation, IEEE Transactions*, 6, 2002.

[KBK07] John-Paul Kelly, Adi Botea, and Sven Koenig. Planning with hierarchical task networks in video games. 2007.

[Kre92] Jurgen Kreuziger. Application of machine learning to robotics: An analysis. *ICARCV '92*, 1992.

[Lar02] Francois Dominic Laramee. Genetic algorithms: Evolving the perfect troll. *AI Game Programming Wisdom*, 1, 2002.

[Mil06] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Mit97] Tom M. Mitchell. *Artificial Intelligence: A Modern Aproach*. McGRAW-HILL, 1997.

[RN08] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Aproach*. Prentice Hall, 2nd edition, 2008.

[SB98] R. S. Sutton and A. G. Barto. Course notes: Reinforcement learning i: An introduction, 1998.

[Tho06] Dale Thomas. Encoding schemes and fitness functions for genetic algorithms. *AI Game Programming Wisdom*, 3, 2006.