**Technical Manual  
*“MicroRTS AI System”***

**Authors:**

Matthew Burr - 10365565

Justin Homsi - 10373949

Lucas Ryan

Nicholas Pleass

*Applied IT Project: MicroRTS AI System - Behaviour Trees and Applied Evolutionary Algorithms 2018*  
*Edith Cowan University*

***Date:*** 30-10-18

***Revision:*** 0.9

***Project Supervisors:*** Martin Masek, Peng Lam

Table of Contents:

[1.](#_30j0zll) Introduction 4

[1.1.](#_1fob9te) Document Identification 4

[1.2.](#_3znysh7) System Overview 4

[1.3.](#_2et92p0) Document Overview 4

[1.4.](#_tyjcwt) Reference Documents 4

[1.4.1.](#_3dy6vkm) Acronyms and Abbreviations 4

[1.4.2.](#_1t3h5sf) Definitions 5

[2.](#_4d34og8) System Description 6

[2.1.](#_2s8eyo1) Introduction 6

[2.2.](#_17dp8vu) System tools: 6

[2.3.](#_3rdcrjn) System installation: 7

[2.4.](#_26in1rg) Operational Scenarios 9

[2.4.1.](#_lnxbz9) MicroRTS\_States\_Traces: 9

[2.4.2.](#_35nkun2) BT\_App: 12

[2.4.3.](#_1ksv4uv) BT\_App\_TraceGetter: 14

[2.5.](#_44sinio) Classes of User 15

[2.6.](#_z337ya) System Requirements 15

[2.6.1.](#_3j2qqm3) Functional: 15

[2.6.2.](#_1y810tw) Non-Functional: 15

[2.6.3.](#_4i7ojhp) Other requirements: 16

[3.](#_2xcytpi) Software Design: 16

[3.1.](#_1ci93xb) Software Design Process 16

[3.1.1.](#_3whwml4) Software Development Environment 16

[3.2.](#_2bn6wsx) Software Design Description 17

[3.2.1.](#_qsh70q) Architecture 17

[3.2.2.](#_3as4poj) Software Interface 19

[3.2.3.](#_1pxezwc) JBT software module components 19

[3.2.4.](#_49x2ik5) MicroRTS software module components 21

[3.2.5.](#_2p2csry) ECJ software module components 25

[3.3.](#_147n2zr) Preconditions for Software 35

[3.3.1.](#_3o7alnk) Pre-conditions for system start-up 35

[3.3.2.](#_23ckvvd) Pre-conditions for system shutdown 35

[3.3.3.](#_ihv636) Pre-conditions for system modification 35

[4.](#_32hioqz) Conclusions 35

[4.1.](#_1hmsyys) State of the system as delivered 35

[4.2.](#_41mghml) Future considerations: 36

[5.](#_2grqrue) Appendix A: (Installation Appendices) 36

[6.](#_vx1227) Appendix B: (Code Snippet Appendices) 36

|  |  |
| --- | --- |
| **Revision History:** | |
| **Revision by:** | **What was added/changed/removed?** |
| 0.1 – Matthew Burr | Document template created |
| 0.2 – Matthew Burr | Added: Section 1 Introduction, Section 2 System introduction |
| 0.3 – Matthew Burr | Added: Section 4 Software Design, Section 5 System Performance |
| 0.4 – Justin Homsi | Added: Section 2.2 Operation scenarios UML diagram, Appendix A - K |
| 0.5 - Justin Homsi | Added: Section 1.4.2 Definitions, Section 4.2.1 Process Flow Description & Sequence Diagram, Section 2.2 Links to tools |
| 0.6 - Matthew Burr | Added: Section 3.2.3 Software Modules |
| 0.7 - Matthew Burr & Justin Homsi | Added: Section 4 future considerations |
| 0.8 - Justin Homsi | Updated: UML Diagram |
| 0.9 – Matthew Burr | Added: Section 2.3 Installation instructions, Section 2.4.1-2.4.3 User scenarios  Updated: Section 2.1-2.2 |

# Introduction

## Document Identification

This technical document details the design and implementation of the MicroRTS AI system which is comprised of three interfaced software modules: MicroRTS application, Java Behaviour Trees (JBT) framework and the Evolutionary Computational Toolkit in Java (ECJ). This document was constructed for use in the ECU Applied IT Project (2018).

## System Overview

The system will use genetic programming present within ECJ to construct and perform computational evolution on a behaviour tree. The aim is to produce an evolved behaviour tree which will control unit decision making within a MicroRTS simulation and discover novel strategies.

## Document Overview

This document provides a detailed look at:

1. Installation of the system
2. A description of the systems three major modules and sub-modules
3. User interfaces and interaction with the system
4. How these modules were constructed or modified
5. How modules are interfaced to produce a behaviour tree evolved using Genetic Programming and any resulting outputs.
6. Future considerations for the system

## Reference Documents

* ECJ - L, Sean. (2015, September 1st). *The ECJ Owner’s Manual* [PDF file]*.* Retrieved from https://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf
* OMG. (2017, December). *Unified Modeling Language (version 2.5.1)* [PDF file]. OMG UML Retrieved from http://www.omg.org/spec/UML/2.5.1
* JBT - D, Ricardo. (2010, August 23rd). *Java Behaviour Trees, User Guide* [PDF file]. Retrieved from https://github.com/gaia-ucm/jbt/blob/master/UserGuide/UserGuide.pdf
* MicroRTS - Santiago Ontañón (2013) *The Combinatorial Multi-Armed Bandit Problem and its Application to Real-Time Strategy Games*, In AIIDE 2013. pp. 58 - 64.

### Acronyms and Abbreviations

*Table 1 - Document acronyms and abbreviations*

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| MicroRTS | Micro Real Time Strategy Simulator |
| JBT | Java Behaviour Trees Framework |
| ECJ | Evolutionary Computation in Java |
| AI | Artificial Intelligence |
| GS | Game State |
| PGS | Physical Game State |
| GUI | Graphical User Interface |
| UTT | UnitTypeTable |
| HP | Hit Points |

### Definitions

*Table 2 - Document acronyms and abbreviations*

|  |  |
| --- | --- |
| **Definitions** | **Meaning** |
| Controller | Software controller which controls the behaviours (gives the units actions) of units within MicroRTS |
| Behaviour Tree | Tree of hierarchical nodes that controls the flow of decision making for units within MicroRTS |
| Generic Behaviour Tree | A behaviour tree which is compatible with all MicroRTS units. |
| Fitness | A score defined by how well a given solution is at solving the defined problem |
| Units | A unit controlled within the MicroRTS application |
| Individual | Candidate solution to the given problem |
| Tournament | A type of simulation within MicroRTS allowing for two or more competing players to combat each other. |
| Map | Grid of arbitrary size |
| Simulation | A MicroRTS tournament using set parameters which is used to test the fitness of a controller and its related behaviour tree. |
| GameState | Stores full game information (Time, PGS, Actions, UTT) |
| Physical Game State | Stores the terrain and a list of units currently in the game |
| State | Defines the size of the map, the players, the starting locations of any units and their hitpoints |
| UnitTypeTable | Definition of unit attributes within MicroRTS (damage, hitpoints, speed, attack range) |

# System Description

## Introduction

The software system is comprised of three major modules: MicroRTS, JBT, and ECJ.

The MicroRTS module is a simple real time strategy application which runs simulations containing multiple competing AI controllers each capable of controlling multiple differing unit types. Thus, the module is used to test/run behaviour trees created via genetic programming and computational evolution. The results of simulations are used to determine the fitness of said behaviour tree.

JBT is a framework which defines and implements a function-set consisting of terminal and non-terminal functions used to construct a functioning behaviour tree. This behaviour tree is utilised by MicroRTS to control unit decision making at each turn of a simulation.

ECJ is an evolutionary toolkit which provides the facilities for Genetic Programming and computational evolution. This module consists of a function-set mimicking the function-set within JBT and used to construct non-functioning behaviour trees in a String format.

When computational evolution is initiated, an initial generation containing a population of individuals is created. Each individual stores a randomised behaviour tree constructed with functions from the function-set. The problem definition ‘MyProblem’ class is used to evaluate the individuals one at a time. This is accomplished by firstly invoking the ‘converter’ class to convert the randomised behaviour tree into a functioning JBT behaviour tree. And then invoking the ECJ\_Tournament class, which assigns a controller with this behaviour tree for use in a simulation. The results of the simulation are returned to ECJ and the fitness of the behaviour tree is evaluated and assigned to the individual. This module runs for a specified number of generations and outputs the statistics of each run.

## System tools:

The tools used to construct the MicroRTS AI software include:

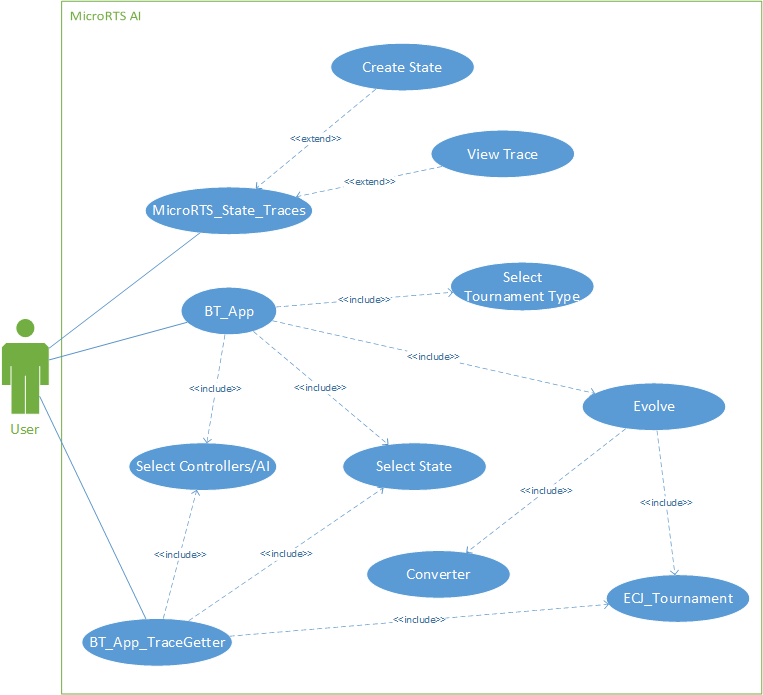
* Java Behaviour Trees Framework
  + Retrieved from: https://github.com/gaia-ucm/jbt
* Evolutionary Computational Toolkit in Java v.26
  + Retrieved from: https://cs.gmu.edu/~eclab/projects/ecj/ecj26.zip
* MicroRTS application
  + Retrieved from: https://github.com/santiontanon/microrts
* Eclipse 2018-09 IDE v.4.9 for Eclipse Committers
  + Retrieved from: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2018-09/R/eclipse-committers-2018-09-win32-x86\_64.zip
* Java programming language JDK/JRE v.1.8
  + JDK Retrieved from: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
  + JRE Retrieved from: https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
* All supplied MicroRTS AI system files for the three modules:
  + MicroRTS
  + JBT
  + ECJ
* Eclipse plugins installed from within the Eclipse IDE:
  + Swing Designer
  + Swing Designer Documentation
  + SWT Designer
  + SWT Designer Core
  + SWT Designer Documentation
  + SWT Designer SWT\_AWT Support
  + WindowBuilder Core
  + WindowBuilder Core Documentation
  + WindowBuilder Core UI
  + WindowBuilder GroupLayout Support
  + WindowBuilder Java Core

## System installation:

Steps for system installation are as follows:

1. Download and install the Eclipse IDE 2018-09 version 4.9 from: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2018-09/R/eclipse-committers-2018-09-win32-x86\_64.zip
2. Download and extract the following software applications and frameworks:
   1. **MicroRTS** application from: *https://github.com/santiontanon/microrts*
   2. **Evolutionary Computational Toolkit in Java v.26** from: *https://cs.gmu.edu/~eclab/projects/ecj/ecj26.zip*
   3. **JBT** framework from: *https://github.com/gaia-ucm/jbt*
3. Once the Eclipse IDE has been installed the next step is to install several required plugins pertaining to the WindowBuilder plugin. These plugins must be present for any sort of future modification of the user GUI’s in the future. The steps to obtain these plugins are as follows:
   1. Click the help tab within Eclipse IDE and select ‘Install New Software’ (*Appendix A - Figure 4*).
   2. In the Install Available Software window set the Eclipse IDE you’re working with to Eclipse IDE 2018-09. (*Appendix A - Figure 5*)
   3. Expand the ‘General Purpose Tools’ plugin list and tick the following plugins:
      1. Swing Designer
      2. Swing Designer Documentation
      3. SWT Designer
      4. SWT Designer Core
      5. SWT Designer Documentation
      6. SWT Designer SWT\_AWT Support
      7. WindowBuilder Core
      8. WindowBuilder Core Documentation
      9. WindowBuilder Core UI
      10. WindowBuilder GroupLayout Support
      11. WindowBuilder Java Core
   4. Click ‘Next’ followed by ‘Finish’ to download and install the required plugins. (*Appendix A - Figure 6*)
   5. Finally restart the Eclipse IDE when prompted to do so.
4. Create a new Java Project with a name of your choice which uses the Java Runtime Environment version 1.8. (*Appendix A - Figure 7*)
5. Right click the new Java Project and click import File System. Browse to the directory of the extracted software applications and frameworks and import each file. (*Appendix A - Figure 8*)
6. Right click the new Java Project and click ‘Build Path -> Configure Build Path’:
   1. Select the ‘Source’ tab and click the ‘Add Folder’ button adding the following source files to the Java Project. Click ‘Apply’. (*Appendix A - Figure 9*)
   2. Select the ‘Libraries’ tab and click the ‘Add JARs...’ button adding the following JAR files to the Java Project. Click ‘Apply’. (*Appendix A - Figure 10*)
   3. Several of the JAR files required by ECJ are retrieved via either running a MakeFile or using a Maven command. To remove these unnecessary steps these JAR files have been supplied within the MicroRTS AI System components file and exist in a nested file named ‘External Required Libraries’. Create a folder named libraries within the Java Project and copy these JARS to the libraries folder.
7. Once all required JAR files have been added to the project it is time to add all supplied MicroRTS AI System files. The first step is to open the ‘Installation components’ folder, open the ‘Components’ folder and copy the relevant files to the relevant source folders:
   1. Open the ‘ECJ’ folder and merge its contained ‘src’ folder with the Java Projects existing ECJ ‘src’ folder. Replace any files in the destination with the same names.
   2. Open the ‘JBT’ folder and merge its contained ‘src’ folder with the Java Projects existing JBT ‘src’ folder. Replace any files in the destination with the same names.
   3. Open the ‘MicroRTS’ folder and merge its contained ‘src’ folder with the Java Projects existing JBT ‘src’ folder. Replace any files in the destination with the same names.
8. This completes the installation of the MicroRTS AI System.

## Operational Scenarios



*Figure 1 – MicroRTS AI System scenarios UML diagram*

## MicroRTS\_States\_Traces:

*Table 3 - MicroRTS AI System scenario: MicroRTS\_States\_Traces GUI - State tab*

|  |  |  |
| --- | --- | --- |
| **Scenario: MicroRTS\_States\_Traces GUI - STATE TAB** | | |
| **Required functionality:**  Required functionality of the MicroRTS\_States\_Traces GUI - State Tab:   * The user will be able to create a state * The user will be able to load a state | | |
| **Scenarios:** | | |
| **Clear a state:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user clicks the ‘Clear’ button | 2 - The system removes all units on the state’s map |
| *Alternate path:* | N/A |  |
| *Assumptions:* | N/A | |
| **Load a state:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user clicks the ‘Load’ button  3 - The user navigates to and selects a state.xml file using the dialog box.  5 - The user clicks the ‘Open’ button | 2 - The system opens a dialog box displaying the system directory.  4 - The system displays the selected files name in the ‘File Name:’ JTextField  6 - The system loads the state.xml file and displays its contents |
| *Alternate path(s):* | *Alternate Path 1:*  3 - The user clicks the ‘Cancel’ button | *Alternate Path 1:*  4 - The system closes the dialog box. |
| *Alternate Path 1.1:*  5 - The user clicks the ‘Cancel’ button | *Alternate Path 1.1:*  6 - The system closes the dialog box. |
| *Assumptions:* | * The state.xml file is configured correctly * A state.xml file exists | |
| **Create a state:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user enters values for the width and height of the state’s map into the width and height JTextFields respectively.  2 - The user clicks a position on the map.  4 - The user selects a unit within the drop-down list  6 - The user clicks the ‘Save’ button  8 - The user navigates to a desired location within the dialog box.  10 - The user clicks on the ‘File Name’ JTextField and inputs a name for the file ending in .xml. The user then clicks the ‘Save’ button. | 2 - The system creates a new state map with the dimensions of the width and height variables.  3 - The system displays a drop-down list of all available units.  5 - The system adds the selected unit to the state map at the specified position  7 - The system opens a dialog box displaying the system directory.  9 - The system updates the dialog box system directory.  11 - The system outputs the state to the specified location within the directory. |
| *Alternate path(s):* | *Alternate Path 1:*  6 - The user clicks the unit on the state map  8 - The user selects ‘Remove’ | *Alternate Path 1:*  7 - The system displays a drop-down list containing the options: Remove  9 - The system removes the unit from the state map |
| *Alternate Path 2:*  6 - The user clicks the ‘Cancel’ button | *Alternate Path 2:*  7 - The system closes the dialog box. |
| *Assumptions:* | N/A | |

*Table 4 - MicroRTS AI System scenario: MicroRTS\_States\_Traces GUI - Trace tab*

|  |  |  |
| --- | --- | --- |
| **Scenario: MicroRTS\_States\_Traces GUI - TRACE TAB** | | |
| **Required functionality:**  Required functionality of the MicroRTS\_States\_Traces GUI - Trace Tab:   * The user will be able to load a trace * The user will be able to step through a trace | | |
| **Scenarios:** | | |
| **Load a trace:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user clicks the ‘Load Trace’ button.  3 - The user navigates to and selects a trace.xml file using the dialog box.  5 - The user clicks the ‘Open’ button. | 2 - The system opens a dialog box displaying the system directory.  4 - The system displays the selected files name in the ‘File Name:’ JTextField.  6 - The system loads the trace.xml file and displays its contents. |
| *Alternate path(s):* | N/A |  |
| *Assumptions:* | * The trace.xml file is configured correctly * A trace.xml file exists | |
| **Step through a trace:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user clicks the ‘+1 Frame’ button.  3 - The user clicks the ‘+1 Action’ button.  5 - The user clicks the ‘+5 Action’ button. | 2 - The system displays the state saved within the trace file one frame into the simulation  4 - The system displays the state saved within the trace file one action into the simulation.  6 - The system displays the state saved within the trace file five actions into the simulation. |
| *Alternate path(s):* | *Alternate Path 1:*  1 - The user clicks the ‘-1 Frame’ button.  *Alternate Path 2:*  3 - The user clicks the ‘-1 Action’ button.  *Alternate Path 3:*  5 - The user clicks the ‘-5 Action’ button. | *Alternate Path 1:*  2 - The system displays the state saved within the trace file one frame prior to the current state displayed in the simulation.  *Alternate Path 2:*  4 - The system displays the state saved within the trace file one action prior to the current state displayed in the simulation.  *Alternate Path 3:*  6 - The system displays the state saved within the trace file five actions prior to the current state displayed in the simulation. |
| *Assumptions:* | * A trace.xml file has been loaded | |

## BT\_App:

*Table 5 - MicroRTS AI System scenario: BT\_App*

|  |  |  |
| --- | --- | --- |
| **Scenario: BT\_App** | | |
| **Required functionality:**  Required functionality of the BT\_App:   * The user will be able to specify parameters for use in computational evolution * The user will be able to change the opponent used for the simulations. * The user will be able to select the type of tournament used which runs the simulation. * The user will be able to select the state to use for the simulations. | | |
| **Scenarios:** | | |
| **Run Evolution** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user inputs values for the seed, population, generations, mutation rate and crossover rate JTextFields. The user then clicks the ‘Opponent’ JComboBox field.  3 - The User selects an opponent from the JComboBox list  5 - The user then clicks the ‘Tournament’ JComboBox field.  7 - The User selects a tournament from the JComboBox list  9 - The user clicks the ‘Choose map’ button.  11 - The user navigates to a state.xml file and clicks the ‘Open button’.  13 - The user clicks the ‘Run’ button | 2 - The system opens up the ‘Opponent’ JComboBox list displaying all opponent names.  4 - The system displays the user selection and assigns it as the opponent to use for simulations run during computation evolution evaluation.  6 - The system opens up the ‘Tournament’ JComboBox list displaying all tournament names.  8 - The system displays the user selection and assigns it as the tournament to use for simulations run during computation evolution evaluation.  10 - The system opens a dialog box displaying the system directory.  12 - The system closes the dialog box and assigns the state.xml as the state to use for simulations run during computation evolution evaluation.  14 - The system runs an evolution using the input values. |
| *Alternate path:* | *Alternate Path 1:*  3 - The user clicks the ‘Clear’ button.  *Alternate Path 2:*  11 - The user clicks the ‘Cancel button’  *Alternate Path 3:*  1 - The user inputs no values for the seed, population, generations, mutation rate and crossover rate JTextFields. The user then clicks the ‘Opponent’ JCombobox field.  (Follows the same main path to action 13) | *Alternate Path 1:*  4 - The system clears all input within the JTextFields  *Alternate Path 2:*  12 - The system closes the dialog box and outputs “Open command cancelled by the user”  *Alternate Path 3:*  2 - The system opens up the ‘Opponent’ JComboBox list displaying all opponent names.  (Follows the same main path to action 12)  14 - The system fills all values for the JTextFields with default values and runs an evolution. |
| *Assumptions:* | * A user inputs valid values for each of the evolution parameters. * A user selects a valid state.xml file for use in the simulations. | |

## BT\_App\_TraceGetter:

*Table 6 - MicroRTS AI System scenario: BT\_App\_TraceGetter*

|  |  |  |
| --- | --- | --- |
| **Scenario: BT\_App\_TraceGetter** | | |
| **Required functionality:**  Required functionality of the BT\_App\_TraceGetter:   * The user will be able to select the state to use for the simulations. * The user will be able to change the opponent used for the simulations. * The user will be able to select stat.out files resulting from evolution runs and retrieve trace files for the best trees of the run. | | |
| **Scenarios:** | | |
| **Get Trace files:** | | |
|  | **User action:** | **System response:** |
| *Main path:* | 1 - The user clicks on the ‘Opponent JComboBox’  3 - The User selects an opponent from the JComboBox list  5 - The user clicks the ‘Choose map’ button.  7 - The user navigates to a state.xml file and clicks the ‘Open button’.  9 - The user clicks the ‘Select Stat File(s)’ button.  11 - The user navigates to a single stat.out file and clicks the ‘Open’ button.  13 - The user clicks the ‘Generate Traces’ button | 2 - The system opens up the ‘Opponent’ JComboBox list displaying all opponent names.  4 - The system displays the user selection and assigns it as the opponent to use for the trace getting simulation.  6 - The system opens a dialog box displaying the system directory.  8 - The system closes the dialog box and assigns the state.xml as the state to use for the trace getting simulation.  10 - The system opens a dialog box displaying the system directory.  12 - The system closes the dialog box and adding the stat.out file to an array of files.  14 - The system generates trace files for each stat.out file in the array of files using the user input values. |
| *Alternate path:* | *Alternate Path 1:*  11 - The user clicks the ‘Cancel’ button.  *Alternate Path 2:*  11 - The user navigates to and selects multiple stat.out files and clicks the ‘Open’ button.  (Follows the same main path to completion) | *Alternate Path 1:*  12 - The system closes the dialog box and outputs “Open command cancelled by the  user”  *Alternate Path 2:*  12 - The system closes the dialog box and adds all selected stat.out files to an array of files.  (Follows the same main path to completion) |
| *Assumptions:* | * The user selects the correct statistics files (Only out.stat files are considered valid format) | |

## Classes of User

There are two classes of user’s that will interact with the system in some way.

1. Typical end-user with little to no experience in the Java programming language and programming in general. These users will be able to use the simple and intuitive GUI’s supplied and will be accessing the system in the same ways as the scenarios described in 2.4.1 - 2.4.3.
2. Technicians with experience in the Java programming language and programming in general. These users will be able to use the contents of this technical document to modify the system in the capacity that suits them, the company or project they represent.

## 

## System Requirements

## Functional:

1. The AI will use behaviour trees to make decisions
2. The AI will discover novel strategies (i.e. win in an interesting way)

## Non-Functional:

1. The Evolutionary Computational Toolkit (ECJ) must be used to perform evolution on the AI with a proper fitness function.
2. The AI’s decisions must include: detect enemies, detect enemy structures, detect allies, detect ally structures, move, attack and idle
3. The AI controller must be used for at least 8-12 units on a team against 8-12 opponents.
4. The AI must be run within the confines of MicroRTS
5. The AI must be evolved to function at least within a single map called ’24 x24’ which consists of the dimensions 24 x 24
6. The AI must be coded with the Java programming language
7. The AI must be constructed using the Netbeans IDE 8.2 software
8. Linux based operating systems must be used to run the Evolutionary Computational Toolkit (ECJ)

## Other requirements:

* Java JDK/JRE 1.8 or higher
* Basic knowledge of the Java programming language
* Basic knowledge of eclipse-IDE

# Software Design:

## Software Design Process

### Software Development Environment

Initial software implementation:

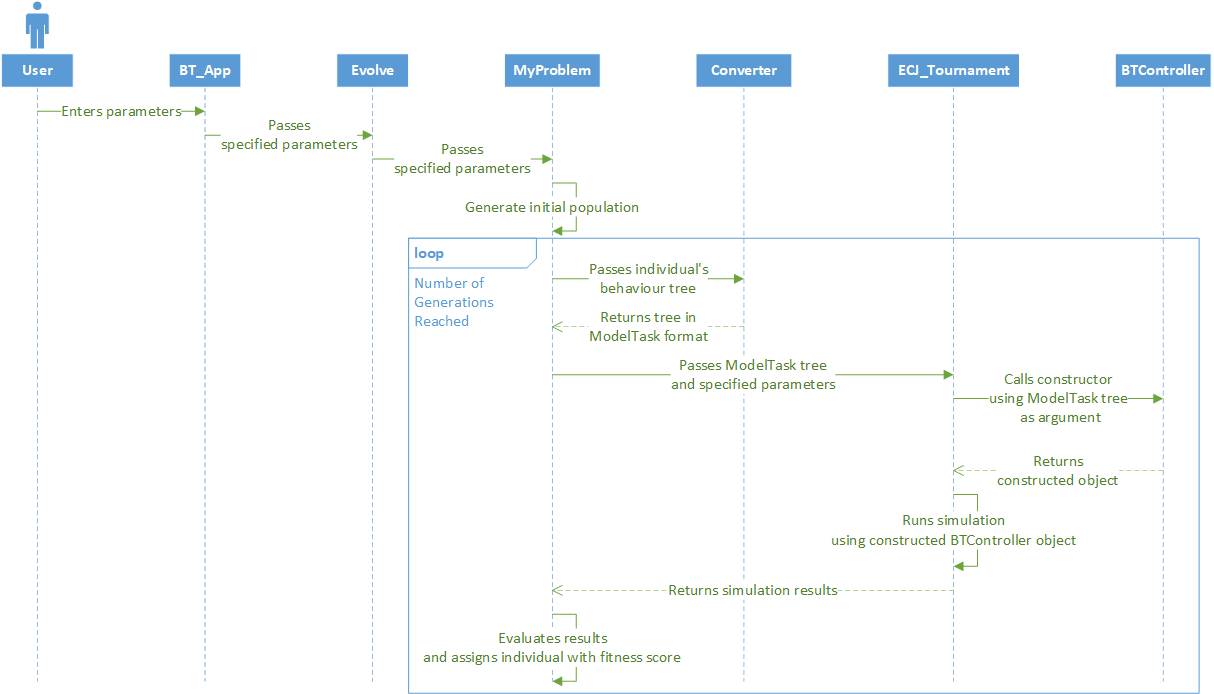
* Initial development began with the writing of pseudocode specifying the general process of how each lower level module such as MicroRTS, JBT and ECJ would function. Later development involved adding increasing complexity to the lower level modules that were implemented and modifying existing code to meet the changes.
* Dependencies were identified requiring that each component be implemented at a particular stage. Implementation occurred in the following order.
  + Implementation began with the MicroRTS module. Work first started on the design of a MicroRTS compatible AI controller. This was very basic to begin with and didn’t use behaviour trees for decision making.
  + JBT was then introduced to the project, providing the facilities for behaviour tree implementation. All required unit decisions were added as functions in the JBT function-set
  + A test JBT behaviour tree was created and initialised within the basic AI controller. This was the first public interface created, connecting and allowing communication between the MicroRTS and JBT software modules.
  + Communication between the modules was achieved with a modification to the AbstractLayerAI class extending the basic AI controller. This modification involved the creation of a method ‘getActionsHash()’ which returned a HashMap variable local to each controller used to store unit actions. Whenever a function within the behaviour tree was accessed it would create an instance of the simple AI controller, add an action to the HashMap variable for a unit and stored within a context local to the behaviour tree. This context was then accessed by the main MicroRTS AI controller utilising the behaviour tree. The HashMap variable was retrieved and merged with the basic AI controllers own HashMap and at the end of its turn would execute the actions retrieved from the JBT module.
  + ECJ was then introduced to the project, providing the facilities for computational evolution and genetic programming. A problem definition was created in the format of the class ‘MyProblem’. The necessary parameter files for basic computational evolution were created and/or retrieved from existing ECJ packages. A function-set derived from the functions within the JBT function set were also constructed and contained within a parameter file. The problem definition and parameter files were then tested thoroughly to confirm computational evolution was being performed as per the specifications of project supervisors.
  + A converter class file was created to convert the behaviour trees generated by ECJ in a String format into a ModelTask type behaviour tree.
  + Conversion tests were performed by outputting the resulting trees before and after conversion into text files. Conversion was occurring successfully however, problems in the form of implementation of the converted behaviour tree were encountered. The converted behaviour tree wasn’t being set as the current behaviour tree for the AI controller to use for a simulation. Thus, at this stage in development the conversion process was incomplete and not working as intended.
  + The converter class scrapped and re-implemented providing a new method of conversion directly to a ModelTask behaviour tree and passed as into the AI Controllers constructor. An update method was created within the JBT BTLibrary file which stores a JBT behaviour tree and called to update its currently stored tree. This updated class and its new functionality are detailed in section 3.2.5.
  + An ECJ\_Tournament class file was created and contains differing custom simulation setups the AI system may use for evolution.
  + The MyProblem class file was further modified to include ECJ\_Tournament methods, and acts as an interface between the two software modules. An ECJ individual’s behaviour tree, once converted, is passed to the ECJ\_Tournament method as an argument. The ECJ\_Tournament file passes this converted tree to the constructor for the AI controller where it is set as the current behaviour tree to use. A custom simulation within ECJ\_Tournament is run and the results returned to the MyProblem problem definition class for use in fitness evaluation.
  + Several GUI’s were then constructed to provide ease of use for a typical end-user and exist in the form of:
    - BT\_App – GUI allowing a user to perform evolutions.
    - BT\_App\_TraceGetter – GUI allowing a user to generate trace files.
    - MicroRTS\_States\_Traces – GUI provided by MicroRTS (FrontEnd) modified slightly for use in creating state files and reviewing a trace file.

## Software Design Description

### Architecture

At a high level the system operates as such with BT\_App:

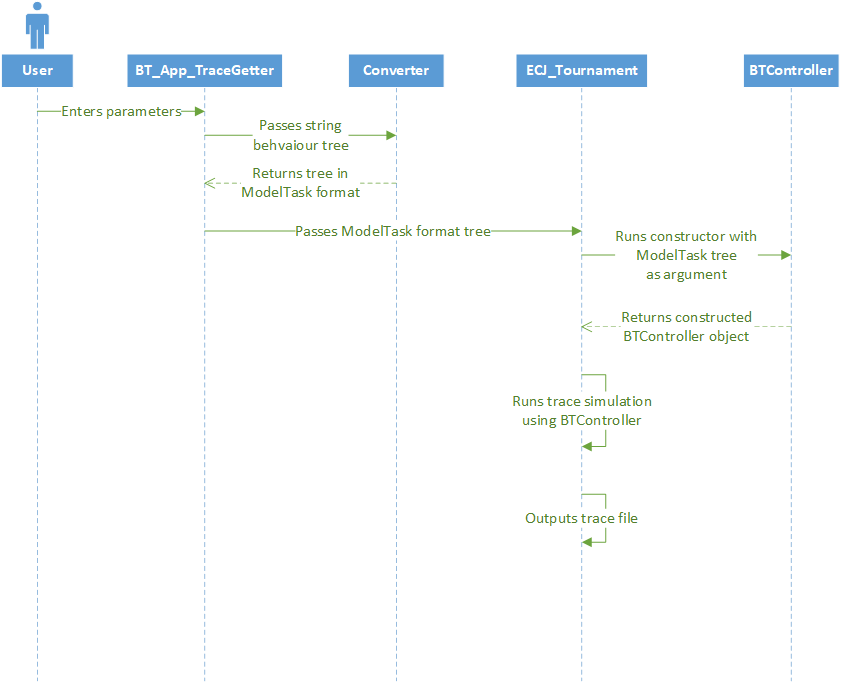
* BT\_App GUI makes a call to the ECJ Evolve class with user specified parameters.
* Evolve makes a call to the ECJ MyProblem class with user specified parameters.
* MyProblem class generates an initial population of individuals. During the evaluation of each individual:
  + A call is made to the Converter class and the individuals behaviour tree used as an argument.
  + The converter class returns this behaviour tree in ModelTask format.
  + A call is made to the ECJ\_Tournament class with the ModelTask format behaviour tree and other retrieved parameters.
  + ECJ\_Tournament class calls the BTController constructor with the ModelTask format behaviour tree.
  + ECJ\_Tournament runs a simulation using BTController.
  + BTController updates the behaviour tree to use as the ModelTask format behaviour tree at each turn of the simulation.
  + ECJ\_Tournament returns simulation results to MyProblem.
  + MyProblem evaluates the results and assigns an individual with a fitness.
* MyProblem generates another population of individuals for a specified number of times and the evaluation loop continues until computational evolution has been completed.



*Figure 2: BT\_App Sequence Diagram*

At a high level the system operates as such with BT\_App\_TraceGetter:

* BT\_App\_TraceGetter GUI makes a call to the Converter class with a string behaviour tree.
* The Converter Class returns this behaviour tree in ModelTask format.
* A call is made to the ECJ\_Tournament class with the ModelTask format behaviour tree
* ECJ\_Tournament class calls the BTController constructor with the ModelTask format behaviour tree.
* ECJ\_Tournament runs a trace simulation using BTController.
* BTController updates the behaviour tree to use as the ModelTask format behaviour tree at each turn of the simulation.
* ECJ\_Tournament outputs a trace file.



*Figure 3: BT\_App\_TraceGetter Sequence Diagram*

### JBT software module components

Within the JBT software module there are two sets of **ModelTask action and condition tasks** which are generated through the use of the JBT framework tool ActionsAndConditionsGenerator. One set of tasks are termed ModelAction task classes, the other, ExecutionAction task classes.

Both types of ModelTask classes share a common class name, with the ModelAction classes constructing instances of the ExecutionAction classes which actually run the pre-defined task. The ExecutionAction classes therefore are the classes which were modified for use in assigning units with decisions. A ModelTask ModelAction class contains a constructor for the ExecutionAction class of the same name, which is used to initialise the task and a further function to execute the task. E.g. The Idle ModelAction class constructor constructs an instance of the Idle ExecutionAction class, and then executes said task.

Seven tasks have been generated with the JBT framework: AttackClosestBase, AttackClosestEnemy, Idle, MoveToClosestAlly, CheckForAllies, EnemyIsClose and EnemyBaseIsClose.

When any task is executed it creates a local instance of BTController, this local controller contains HashMap variable ‘actions’ which is used to store assigned unit actions. (code snippet)

There are two methods also common to all tasks: internalSpawn() and internalTick(). InternalSpawn() is the method initially used to create and run an instance of a particular task, and InternalTick() is the method used to determine the status of the task in the tree. The status could be returned as RUNNING, SUCCESS or FAILURE depending on criteria local to each task. The result of the status is used to determine if the task has succeeded in supplying an action or failed.

AttackClosestBase: InternalSpawn() retrieves two variables from the context, an array list of enemy unit bases and the current unit which needs a decision. If enemy bases exist, the closest base is calculated and an attack command is given to the unit and stored within the local controller’s actions variable. This actions variable is retrieved using the controller’s getActionsHash() method and stored within the context. InternalTick() checks whether enemy bases exist and if not returns the status FAILURE, otherwise SUCCESS is returned.

AttackClosestEnemy: InternalSpawn() retrieves two variables from the context, an array list of enemy units (not including bases) and the current unit which needs a decision. If enemy units exist, the closest enemy unit is calculated and an attack command is given to the unit and stored within the local controller’s actions variable. This actions variable is retrieved using the controller’s getActionsHash() method and stored within the context. InternalTick() checks whether enemy units exist and if not returns the status FAILURE, otherwise SUCCESS is returned.

Idle: InternalSpawn() retrieves the current unit which needs a decision from the context. An idle command is given to the unit (wait for 10 turns) and stored within the local controller’s actions variable. This actions variable is retrieved using the controller’s getActionsHash() method and stored within the context. InternalTick() returns the status SUCCESS.

MoveToClosestAllies: InternalSpawn() retrieves two variables from context, an array list of ally units and the current unit which needs a decision. The closest ally is calculated and its position retrieved in the form of integer x (incremented by one) and y coordinates. The unit is then given a move command with the coordinates and stored within the local controller’s actions variable. This actions variable is retrieved using the controller’s getActionsHash() method and stored within the context. InternalTick() checks whether ally units exist and if not returns the status FAILURE, otherwise SUCCESS is returned.

EnemyIsClose: InternalSpawn() retrieves an array list of enemy units (not including bases) and a check is performed, setting a boolean variable check to true if they exist. InternalTick() checks the value of the check variable, if it’s set to true the status SUCCESS is returned, otherwise FAILURE is returned.

EnemyBaseIsClose: InternalSpawn() retrieves an array list of enemy bases and a check is performed, setting a boolean variable check to true if they exist. InternalTick() checks the value of the check variable, if it’s set to true the status SUCCESS is returned, otherwise FAILURE is returned.

CheckForAllies: InternalSpawn() retrieves two variables from context, an array list of ally units and the current unit which needs a decision. The closest ally is calculated and the position of the unit and closest ally are compared to determine the distance between the two. If the difference in distance is two or less, a boolean variable check is set to true. InternalTick() checks the value of the check variable, if it’s set to true the status SUCCESS is returned, otherwise FAILURE is returned.

**BT\_Library file:** A java class file which is initially generated through the use of the JBT framework tool BTLibraryGenerator. This file contains a static ModelTask ‘TestTree’ variable which stores a behaviour tree as well as two methods to manipulate this variable:

1. getBT() - This method retrieves the ModelTask TestTree variable
2. updateBT() - This method updates the ModelTask TestTree variable

**IBTLibrary file:** A Java class interface file common to all behaviour tree libraries created within the JBT Module. This class had to be modified to contain an updateBT() method so that the BT\_Library file using this interface also contained the specified method.

**GenericBTLibrary:** A Java class file implementing the IBTLibrary class interface file. Thus, it had to be modified to contain the new updateBT() method created within the interface to avoid errors during compilation.

### MicroRTS software module components

**AbstractionLayerAI:** is a java class used via BTController to extend functionality in regard to pathfinding, how actions are carried out in the backend and all potential actions a unit may be assigned. The only change made to this part of the software was the addition of the method getActionsHash() which returns the actions HashMap<Unit, AbstractAction> of each controller extending this class.

**FrontEnd:** is a pre-existing GUI found within the MicroRTS application and provides the ability to create and save states, run simulations, save and load traces as well as run a tournament. The FETracePane which details the trace portion of the GUI layout and enables a user to step forward and back through traces one or more frames or actions at a time. The FETracePane was modified to include the following functionality:

* Button and required functionality to step through the trace five actions at a time.
* Button and required functionality to step backwards through the trace five actions at a time.

This modified FETracePane and the FrontEnd GUI associated with it were exported as an executable JAR file now termed **‘MicroRTS\_States\_Traces’.**

**Custom\_LightRush:** was a controller created for the purpose of being a reliable opponent for testing and is essentially just a modified version of the pre-existing WorkerRush controller. It controls all unit behaviours the same way a standard WorkerRush controller would with the exception of the base behaviours which have been disabled.

**BTController:** is a MicroRTS compatible controller which utilises a behaviour tree to control unit decision making. There are two constructors inherent to this class:

* One accepting a UnitTypeTable variable which stores information about the unit types and associated attributes. This constructor is used within JBT tasks.
* One accepting UnitTypeTable, PathFinding and ModelTask variables. The UnitTypeTable variable stores information about the unit types and associated attributes while the PathFinding variable determines the type of pathfinding units will use for movement actions. Lastly the ModelTask variable is used to accept a ModelTask behaviour tree.

Within the second constructor body an instance of the BT\_Library class is initialised; its update method is called with the argument of the ModelTask behaviour tree. This ModelTask behaviour tree within the BT\_Library class is then retrieved with the getBT() method. A context variable used to store variables necessary within the JBT tasks is created by passing the BT\_Library class. Finally, an executable behaviour tree is initialised using the retrieved ModelTask behaviour tree and context variables.

Within the main BTController class, after construction, the getAction() method, when called, accepts a player integer and assigns the controller to use that particular player configuration within a simulation. Also accepted is a GS variable containing all information related to the current simulation (All units and their associated attributes, the map and size). The getAction() method proceeds to examine the GS for all units on the map and adds them to arrays based on the player controlling them or the type of the unit.

* If the unit is an enemy unit add it to an enemy array, else if it’s an enemy base add it to an enemy bases array.
* If the unit is an ally unit add it to a friendly array.
* These arrays are then all set within the context variable for use within the JBT tasks.

A for loop then loops through each ally unit within the friendly array, re-initialises the behaviour tree and sets the current unit within the context. The behaviour tree is executed once, and the actions variable is retrieved from the context. This actions variable is a HashMap<Unit, AbstractAction> and contains the decision or actions that have been assigned by a task within the behaviour tree to the given unit. All actions within this variable are then added to the local actions HashMap<Unit, AbstractAction> inherent to BTController.

Once all units have been assigned with an action the translateActions() method is called to check the validity and perform the actions assigned within the actual simulation.

**ECJ\_Tournament:** This class was designed to run custom simulations with differing controller and PGS setups and to save traces.

Two constructors exist for this class:

* Constructor one accepts a ModelTask behaviour tree and assigns this to a ModelTask variable ‘tree’ contained within the class.
* Constructor two accepts a modelTask behaviour tree, String scenePath, String opponent and integer count variables. The ModelTask behaviour tree and count variables are assigned to a ModelTask variable ‘tree’ and integer traceCount variables respectively. A call is then made to the tournamentTrace method inherent to the class using the two String variables (scenePath, opponent) as arguments.

The first method of this class ‘tournament’ accepts two arguments: A String scenePath and a String opponent. The scenePath contains the absolute path of a state used for setting up a scenario and is loaded in as the PGS to use for the simulation, while opponent is used later to select the opposing controller to use within the simulation.

* Firstly, the PGS is set with the scenePath variable, a GS is created using the PGS and a UTT (defines the scenario and unit attributes of the simulation).
* The maximum number of turns and a period of delay is set for the simulation. The maximum number of turns specifies how long the simulation will run for while the delay specifies a visual delay slowing the simulation down by a certain amount allowing it to be more readable. The maximum number of turns is set to 3000, while the delay is set to 10.
* The calcHP() method is called to set the starting HP of each controller’s unit set and a maximum HP variable based on the team with the highest starting HP.
* The AI to use in the simulation is set, the first AI being set to BTController which is passed a UTT, PathFinding and ModelTask tree variable. The second AI is set via a call to the getOpponent() method passing it the opponent and UTT variables.
* A JFrame is then constructed with the GS to provide a visual display of the simulation occuring.
* The simulation is then run, for each turn in the simulation the getAction() method is called for each controller. This method retrieves requested actions from either controller, updates the GS and and executes the actions within the simulation at which point the JFrame is updated to demonstrate these actions visually. This continues until the maximum number of turns have been reached or one controller kills all associated opponent controller units.
* The post-game GS variable is then returned.

The second method of this class **‘**tournamentFast’accepts two arguments: a String scenePath and a String opponent. The scenePath contains the absolute path of a state used for setting up a scenario and is loaded in as the PGS to use for the simulation, while opponent is used later to select the opposing controller to use within the simulation.

* Firstly, the PGS is set with the scenePath variable, a GS is created using the PGS and a UTT (defines the scenario and unit attributes of the simulation).
* The maximum number of turns and a period of delay is set for the simulation. The maximum number of turns specifies how long the simulation will run for while the delay specifies a visual delay slowing the simulation down by a certain amount allowing it to be more readable. The maximum number of turns is set to 3000, while the delay is set to 0.
* The calcHP() method is called to set the starting HP of each controller’s unit set and a maximum HP variable based on the team with the highest starting HP.
* The AI to use in the simulation is set, the first AI being set to BTController which is passed a UTT, PathFinding and ModelTask tree variable. The second AI is set via a call to the getOpponent method passing it the opponent and UTT variables.
* No JFrame is constructed within this method so as to allow the simulations to occur more efficiently and only the outcome of the simulation is important.
* The simulation is then run, for each turn in the simulation the getAction method is called for each controller. This method retrieves requested actions from either controller, updates the GS and executes the actions within the simulation. This continues until the maximum number of turns have been reached or one controller kills all associated opponent controller units.
* The post-game GS variable is then returned.

The third method of this class **‘**tournamentComputationBudget’ accepts two arguments a String scenePath and a String opponent. The scenePath contains the absolute path of a state used for setting up a scenario and is loaded in as the PGS to use for the simulation, while opponent is used later to select the opposing controller to use within the simulation.

* Firstly, the PGS is set with the scenePath variable, a GS is created using the PGS and a UTT (defines the scenario and unit attributes of the simulation).
* The maximum number of turns, period of delay and maximum number of inactive turns are set for the simulation. The maximum number of turns specifies how long the simulation will run for while the delay specifies a visual delay slowing the simulation down by a certain amount allowing it to be more readable. The maximum number of inactive turns specifies the number of turns a controller must calculate the actions of its units before an action is performed. The maximum number of turns is set to 3000, the delay is set to 0 and the maximum number of inactive turns is set to 30.
* The calcHP method is called to set the starting HP of each controller’s unit set and a maximum HP variable based on the team with the highest starting HP.
* The AI to use in the simulation is set, the first AI being set to BTController which is passed a UTT, PathFinding and ModelTask tree variable. The second AI is set via a call to the getOpponent() method passing it the opponent and UTT variables.
* No JFrame is constructed within this method to allow the simulations to occur more efficiently as only the outcome of the simulation is important.
* The simulation is then run, for each turn in the simulation the getAction() method is called for each controller. This method retrieves requested actions from either controller, updates the GS and executes the actions within the simulation. A variable lastTimeActionIssued is used to store the current time an action was issued in the simulation. This continues until the maximum number of turns have been reached or one controller kills all associated opponent controller units. One further check is performed to determine whether the game has ended or not by comparing the current time of the GS, subtracting the lastTimeActionIssued variable and checking that this result is less than the maximum number of inactive cycles.
* The post-game GS variable is then returned.

The fourth method of this class ‘tournament\_trace’ accepts two arguments: A String scenePath and a String opponent. The scenePath contains the absolute path of a state used for setting up a scenario and is loaded in as the PGS to use for the simulation, while opponent is used later to select the opposing controller to use within the simulation.

* Firstly, the PGS is set with the scenePath variable, a GS is created using the PGS and a UTT (defines the scenario and unit attributes of the simulation).
* The maximum number of turns and a period of delay is set for the simulation. The maximum number of turns specifies how long the simulation will run for while the delay specifies a visual delay slowing the simulation down by a certain amount allowing it to be more readable. The maximum number of turns is set to 3000, while the delay is set to 0.
* The calcHP() method is called to set the starting HP of each controllers unit set and a maximum HP variable based on the team with the highest starting HP.
* The AI to use in the simulation is set, the first AI being set to BTController which is passed a UTT, PathFinding and ModelTask tree variable. The second AI is set via a call to the getOpponent() method passing it the opponent and UTT variables.
* A Trace variable is initialised at this point in the method and an initial trace entry specifying the current GS time and PGS of the GS is added to the variable.
* No JFrame is constructed within this method so as to allow the simulations to occur more efficiently as only the outcome of the simulation is important.
* The simulation is then run, for each turn in the simulation the getAction() method is called for each controller. This method retrieves requested actions from either controller, updates the GS and executes the actions within the simulation. At this point in the method, a trace entry is created containing the current GS time, PGS configuration and the actions performed by each controller and added to the Trace variable. This continues until the maximum number of turns have been reached or one controller kills all associated opponent controller units.
* Once the simulation has completed, a final trace entry is created containing the current GS time and the PGS configuration. An XMLWriter is initialised and writes the completed trace to an xml file under the name “trace ” + traceCount + result +”.xml”.

The fifth method of this class **‘**getOpponent’accepts the opponent variable and a UTT. getOpponent performs a simple comparison of the opponent variable and a set of potential matches, if a match is found a constructor is called for the match (e.g. opponent variable equals “Custom\_LightRush”, a match is made to a pre-set string “Custom\_LightRush” and the constructor is called AI ai2 = new Custom\_LightRush(utt)) and returned as the AI opponent.

The sixth method of this class **‘**reset’ sets the starting HP of each team to 0.0, maximum HP to 0.0 and the stored result of a simulation to 0;

The seventh method of this class **‘**calcHP’ accepts a PGS variable, from which it calculates the total starting HP of each team (set of units owned by a controller stored within the PGS) before the simulation has begun, as well as the maximum HP, which is set to the starting HP of the team with the highest HP.

The eighth method of this class **‘**getMaxHP’returns the maximum HP variable.

The ninth method of this class **‘**getStartingHP’returns the starting HP of a team depending on an int variable related to each controller. If an integer of one is entered, the opponent controllers starting HP is returned, otherwise BTControllers starting HP is returned.

The tenth method of this class **‘**getResult’accepts a GS variable and retrieves the total HP of each team’s remaining units for use in a local fitness calculation using a fitness function mimicking that found within ECJ’s MyProblem class. This fitness calculation result is used as a variable for trace naming.

### ECJ software module components

Within the ECJ software module multiple class files defining terminal and non-terminal tasks to be a part of a **function-set** were created and are based on the ModelTask function-set within JBT. Terminal class files define the actions and conditions while non-terminal tasks define selectors and sequences of varying aritys. Each terminal and non-terminal class file has three associated methods, toString(), expectedChildren() and eval(). toString() is used to return a string representing a task within an ECJ behaviour tree once constructed, expectedChildren() is used to determine whether the class can contain nested children terminals and specify the number of children terminals if so, and eval() performs evaluation of the class within the MyProblem class and is used to assign children to the non-terminal tasks.

Terminal class files are listed below with their associated methods:

* ACB - AttackClosestBase
  + String() returns the string “AttackClosestBase”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* ACE
  + String() returns the string “AttackClosestEnemy”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* CA
  + String() returns the string “CheckForAllies”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* CB
  + String() returns the string “CheckForBases”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* CE
  + String() returns the string “CheckForEnemies”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* MCA
  + String() returns the string “MoveToClosesAllies”
  + expectedChildren() returns 0
  + eval() assigns the class with no children
* I
  + String() returns the string “Idle”
  + expectedChildren() returns 0
  + eval() assigns the class with no children

Non-terminal class files are listed below with their associated methods:

* SelectorChildOne
  + String() returns the string “SelectorChildOne”
  + expectedChildren() returns 1
  + eval() assigns the class with one child terminal class
* SelectorChildTwo
  + String() returns the string “SelectorChildTwo”
  + expectedChildren() returns 2
  + eval() assigns the class with two child terminal classes
* SelectorChildThree
  + String() returns the string “SelectorChildThree”
  + expectedChildren() returns 3
  + eval() assigns the class with three child terminal classes
* SelectorChildFour
  + String() returns the string “SelectorChildFour”
  + expectedChildren() returns 4
  + eval() assigns the class with four child terminal classes
* SequenceChildOne
  + String() returns the string “SequenceChildOne”
  + expectedChildren() returns 1
  + eval() assigns the class with one child terminal class
* SequenceChildTwo
  + String() returns the string “SequenceChildTwo”
  + expectedChildren() returns 2
  + eval() assigns the class with two child terminal classes
* SequenceChildThree
  + String() returns the string “SequenceChildThree”
  + expectedChildren() returns 3
  + eval() assigns the class with three child terminal classes
* SequenceChildFour
  + String() returns the string “SequenceChildFour”
  + expectedChildren() returns 4
  + eval() assigns the class with four child terminal classes

Parameter files are used within ECJ to set the parameters for the Evolve class which will perform genetic evolution and apply the genetic processes specified within the parameters. There are four parameter files which were utilised for the software system: BT.params, ec.params, koza.params and simple.params.

**BT.params:** The BT.params parameter file firstly specifies that there’s another linked parameter file containing additional parameters for use within the evolution, koza.params. Function-set parameters are then specified:

* Number of function-sets is set to one
* The type of the function-set specified as a genetic programming type function-set
* The size of the function-set is set to fifteen
* Each terminal and non-terminal function is defined as a member of the function-set and their number of children outlined
* The problem definition is specified as the MyProblem class
* The statistics output file is specified out.stat

**koza.params**: The koza.params parameter file firstly specifies that there’s another linked parameter file containing additional parameters for use within the evolution, simple.params. The following parameters are then specified:

* Fitness of an individual is specified as simple fitness
* Initialiser is specified as GPInitialiser
* Subpopulation
  + Of type GPSpecies
  + Individuals are of type GPIndividual
  + Number of retries for duplicates within the subpopulation set to 0
* Individuals consist of:
  + One tree
  + Of type GPTree
  + Tree constraints set to tc0
* GPSpecies breeding pipelines:
  + MultiBreedingPipeline
    - CrossoverPipeline of probability 0.6
    - MutationPipeline of probability 0.1
    - ReproductionPipeline of probability 0.1
* Default values for crossover, reproduction and mutation
  + Reproduction source - SUS
  + Crossover source - SUS
    - Node selector - KozaNodeSelector
    - Try crossover one time
    - Max depth of 6
  + Mutation source - SUS
    - Build selector - KozaNodeSelector, GROW
    - Max depth of 6
    - Try mutation one time
* Default selection values
  + SUS shuffle is true
* Default GROW values for sub-tree mutation
  + min depth of 5
  + max depth of 5
* Default values for KozaNodeSelection:
  + 10% pick a terminal
  + 90% pick a non-terminal
  + 0% pick a root
* Specify the the GP type associated with each terminal or non-terminal node. A non-terminal node may only have children of the same type.
  + Only one type
  + Name of “nil”
  + Size of one
  + Each node shares the same type
* Define the default values for tree constraints tc0
  + only one tree constraints object
  + Of the type GPTreeConstraints
  + Name of tc0
  + HalfBuilder may create nodes only from a defined functionset fc0 (defined within BT.params)
* Define default values for tree initialisation
  + Uses HalfBuilder to construct an initial tree
  + Min depth of 2
  + Max depth of 6
  + Grow probability of 0.5
* Define GPNodeConstraints
  + Defines nodes with one - four children

**simple.params:** The simple.params parameter file firstly specifies that there’s another linked parameter file containing additional parameters for use within the evolution, ec.params. The following parameters are then specified:

* Simple generational evolution will be used
* The standard initialisation method will be used
* The standard finalisation method will be used
* The standard exchanger method will be used
* The standard breeder method will be used
* The statistics will be output as SimpleStatistics and SimpleShortStatistics
* The number of generations is set to 500
* The population size is set to 50
* Do not perform elitism
* Output SimpleStatistics to the file out.stat
* Output SimpleShortStatistics to the file out2.stat

**ec.params:** The ec.params parameter specifies the following parameters:

* Number of threads used for evaluation is set to 1
* Number of threads used for breeding is set to 1
* The seed is set to time
* Checkpointing is disabled

**Evolve**: The Evolve class file is the class that initialises evolutionary computation using the parameter files and the MyProblem class problem definition. Usually once evolutionary computation has completed the system is terminated, however this statement on line 784 of the class Evolve has been commented out to allow multiple evolutions to occur.

**Converter**: The converter class file is comprised of methods used to convert a string version of a behaviour tree created by ECJ into an equivalent behaviour tree constructed of JBT ModelTask functions.

The makeTask() method accepts a String formatted behaviour tree and an integer variable max representing the maximum depth of the behaviour tree.

* String and ModelTask arrays are initialised
* While the maximum depth is greater than 0
  + The method loops through the string behaviour tree and locates the first instance of the special character integer combination where the integer matches the current depth and sets an index to the start of this instance. The string is looped through again and a subsequent instance of the special character integer combination is found where the integer matches the current depth, an end index is set to the end of this instance.
  + A sub-string is created using the index and end index to retrieve the non-terminal children combination at this depth.
  + The curly brace special characters are replaced with apostrophes and it is added to the string array and the method evalSub() is called and passed the following parameters:
    - The sub-string
    - The current string array
    - The current model array
  + evalSub() creates a ModelTask for the substring which is added into the ModelTask array
  + The loop through the string continues from the location of the end index to find and evaluate all sub-string non-terminal children combinations at this depth.
  + The depth is decremented, and the while loop evaluates the lower depths
* The last ModelTask behaviour tree within the ModelTask array which has been completely evaluated at all depths is returned.

The formatString() method accepts a String variable behaviour tree comprised of terminal and non-terminal functions represented via their string names with each non-terminal encapsulated with its children via parentheses (example behaviour tree string).

* The formatString() method firstly parses the string character by character, checking for parentheses.
* Once a parenthesis is found it is replaced with a special character and integer combination in the form of ‘{intString}’. The intString variable is a string representation of a count variable which is incremented on each open parenthesis found and decremented on each closing parentheses found.
* Each non-terminal and its children are now delimited and identifiable by their depth within the string. The lowest depth is represented by an intString of “0” with increasing depth for each encapsulated non-terminal children pair.
* The formatted String behaviour tree is then returned.

The getMax() method accepts a String variable formatted behaviour tree and iterates through the string to find the maximum depth.

* The string is parsed character by character and each time the special character integer combination ‘{intString}’ is found, regex is used to retrieve the intString and convert it into an integer. This integer is checked against the current max depth found and if it is greater the max depth is set to the integer.
* The max depth is then returned.

The evalSub() method accepts three parameters: a String sub-string of the behaviour tree, a String ArrayList stringArray and a ModelTask ArrayList modelArray:

* The sub-string is firstly checked to determine whether it contains any other previous sub-strings within stringArray and if so a boolean ‘requirements’ is set to true, otherwise it is false.
* If requirements is false:
  + All special characters and integers are stripped from the sub-string.
  + The sub-string is split on whitespaces and added to a String array termed ‘taskSplit’
  + Count the number of strings within ‘taskSplit’ and set an integer numChild to this result
  + A string named parent is set to the value of the first index of the taskSplit array
  + The parent string is then compared against both “Selector” or “Sequence” to determine the non-terminal function it represents.
  + If it is a selector:
    - Create an ArrayList of ModelTasks named child.
    - For the remaining indexes of the taskSplit array call the method getTaskType() passing it the taskSplit[index]. This returns a ModelTask representing a terminal function and is added to child.
    - Call the method createSel() and pass it the child ModelTask array. This returns a completed Selector ModelTask object which is added to modelArray.
  + If it is a sequence:
    - Create an ArrayList of ModelTasks named child.
    - For the remaining indexes of the taskSplit array call the method getTaskType() passing it the taskSplit[index]. This returns a ModelTask representing a terminal function and is added to child.
    - Call the method createSeq() and pass it the child ModelTask array. This returns a completed Sequence ModelTask object which is added to modelArray.
* If requirements is true:
  + Loop through the stringArray backwards and check if the sub-string contains a string within the stringArray. If a string matches a portion of the sub-string, replace this section of the sub-string with ‘[c+intNum+]’ where intNum is the index of the matching string.
  + The sub-string is split on whitespaces and added to a String array termed ‘taskSplit’
  + Count the number of strings within ‘taskSplit’ and set an integer numChild to this result
  + A string named parent is set to the value of the first index of the taskSplit array
  + The parent string is then compared against both “Selector” or “Sequence” to determine the non-terminal function it represents.
  + If it is a selector:
    - Create an ArrayList of ModelTasks named child.
    - For the remaining indexes of the taskSplit array.
      * Check whether the value of the taskSplit array at that index contains “[c” and if it does retrieve the intNum integer value. This integer value is used to retrieve a modelTask at that index of the modelArray and is added to child.
      * Otherwise, call the method getTaskType() passing it the taskSplit[index]. This returns a ModelTask representing a terminal function and is added to child.
    - Call the method createSel() and pass it the child ModelTask array. This returns a completed Selector ModelTask object which is added to modelArray.
  + If it is a sequence:
    - Create an ArrayList of ModelTasks named child.
    - For the remaining indexes of the taskSplit array.
      * Check whether the value of the taskSplit array at that index contains “[c” and if it does, retrieve the intNum integer value. This integer value is used to retrieve a modelTask at that index of the modelArray and is added to child.
      * Otherwise, call the method getTaskType() passing it the taskSplit[index]. This returns a ModelTask representing a terminal function and is added to child.
    - Call the method createSeq() and pass it the child ModelTask array. This returns a completed Sequence ModelTask object which is added to modelArray.

getTaskType() method accepts a String variable task representing a terminal function and, using a switch statement, compares the string against string representations of all possible terminal functions within JBT’s function-set. Once a match is made a JBT ModelTask representing the terminal function is returned.

createSeq() method accepts a ModelTask array children and depending on the number of children will return a selector JBT ModelTask representing the non-terminal function and its terminal child functions.

createSel() method accepts a ModelTask array children and depending on the number of children will return a sequence JBT ModelTask representing the non-terminal function and its terminal child functions.

**MyProblem:** The MyProblem class is used as the problem definition for the computational evolution. It is invoked to evaluate individuals within a generation and assign a fitness to said individuals based on a fitness function. When called, it runs the evaluate method with arguments pertaining to the current state of the evolution, an individual, subpopulation and thread numbers it has access to for the evaluation.

* The following variables are initialised:
  + result of type double to store the result of the fitness calculation
  + RedHP of type double to store the remaining HP of the opponent controller’s units post-simulation
  + BlueHP of type double to store the remaining HP of BTController’s units post-simulation
  + TotalRedHP of type double to store the total starting HP of the opponent controller’s units
  + TotalBlueHP of type double to store the total starting HP of BTController’s units
  + MaxHP of type double to store the maximum starting HP of either team with the highest starting HP.
* In order to evaluate an individual, the individuals associated behaviour tree is retrieved and stored as a string.
* The converter’s makeModelTask() method is called and the string tree passed as an argument. The makeModelTask() method returns an equivalent ModelTask tree constructed with ModelTask functions from the JBT function-set.
* An ECJ\_Tournament object is initialised, and its constructor is called and passed this ModelTask tree as an argument.
* Several parameter values set within the evolution state’s parameter database are retrieved and assigned to strings:
  + path - a parameter used to store the absolute path of a MicroRTS state file used to set the scenario up within a simulation set via a user within BT\_App
  + opponent - a parameter used to store a string pertaining to an opponent controller set by a user within BT\_App
  + tournament - a parameter used to store a string pertaining to the type of tournament to run which is set by a user within BT\_App
* The getTournamentType() method is called and passed the above strings representing parameter values and runs the specified tournament, returning a GS variable containing post-simulation results.
* The ECJ\_Tournament object’s getStartingHP() method is invoked to retrieve the total starting HP of both the opponent controller and BTController, and assign them to TotalRedHP and TotalBlueHP respectively.
* The ECJ\_Tournament object’s getMaxHP() method is invoked to retrieve the maximum starting HP of either team with the highest HP.
* The post-simulation GS is used to calculate the remaining HP of both teams and assigned to the variables BlueHP and RedHP.
* The fitness of the individual is calculated using a fitness function in the form of: (code snippet) and is set to result and assigned to the individual.

**MyThread:** The MyThread class uses a thread to execute the Evolve class and initiate evolutionary computation. The constructor accepts a String array of arguments which is set to an array variable called arguments local to the class. When a thread is initialised and the start method called it will run Evolve’s main() method with the array as an argument.

**MySecondThread:** The MySecondThread class uses a thread to execute the tournament\_Trace method within ECJ\_Tournament and retrieve trace files.

* An ArrayList of Strings trees, and an ArrayList of ModelTask tasks are initialised
* Initially a constructor is passed three variables: A file array files, a string mapPath and a string opponentName. Variables of equivalent name local to the MySecondThread class are assigned to these values.
* When a thread is started the findTrees() method is called and passed the files array which searches each file within the files array and retrieves the best trees of the run(s). These trees are added to trees.
* The makeTask() method is then called and passed the trees ArrayList. For each string within trees, the converter is called and passed the string representing a behaviour tree, which returns an equivalent ModelTask behaviour tree which is added to tasks.
* The runTasks() method is then called and passed tasks, the mapPath, and opponentName. For each task in tasks, ECJ\_Tournament’s constructor is called and passed the tasks, mapPath, and opponentName variables as well as a local count integer.
* Each time the constructor is called with these arguments, ECJ\_Tournament will run its tournament\_Trace which will run a simulation based on the given arguments and save a trace XML file.
* The count integer is incremented with each call and is used as part of the trace naming scheme.

**BT\_App:** The BT\_App class forms one of the GUIs a user will be able to interact with in order to operate the system. The BT\_App GUI provides the user with the functionality to perform evolution. Components of the GUI include:

* Multiple JTextField components for the following:
  + Seed - Accepts an integer value which is set as the seed parameter for the evolution.
  + Population - Accepts an integer value which is set as the population size parameter, specifying the number of individuals within each generation.
  + Generations - Accepts an integer value which is set as the number of generations parameter, specifying the number of generations to evolve.
  + Mutation rate - Accepts a double value which is set as the mutation rate probability parameter, specifying how often mutation occurs.
  + Crossover rate - Accepts a double value which is set as the crossover rate probability parameter, specifying how often crossover occurs.
* Multiple JComboBox components for the following:
  + Opponent - Allows the user to set the opponent parameter, specifying the opponent controller to use within simulations.
  + Tournament - Allows the user to set the tournament parameter, specifying the type of tournament to use within simulations.
* Multiple JButton components for the following:
  + Clear - Clears all JTextField input
  + Choose map - Allows the user to choose a map
  + Start - Initiates computation evolution run
* JTextArea which uses a custom output stream redirecting all error and system output to the GUI JTextArea.

There’s a method actionPerformed() which accepts an ActionEvent provided by an ActionListener attached to each of the JButton and JComboBox components.

* If the source of the ActionEvent was the Choose map button a JFileChooser is used to allow a user to select a state to be used in the simulation. The absolute path of this state file is assigned as the value of a String mapPath variable. This state is necessary to set up the scenario including for both controller: units, unit positions and the map size.
* If the source of the ActionEvent was the Clear button, all JTextField values are reset to null allowing for new input.
* If the source of the ActionEvent was the Opponent JComboBox, the value of the selected JComboBox is converted to a String and assigned to the variable opponentName.
* If the source of the ActionEvent was the Tournament JComboBox, the value of the selected JComboBox is converted to a String and assigned to the variable tournamentName.
* If the source of the ActionEvent was the Start button, the values of the JTextField are checked for user input. If no user input was received, they’re set to default pre-set values. A String array of arguments is constructed using all input retrieved from each JComponent object. A thread is initialised as a MyThread class is passed the arguments String array and it’s start method called to initialise evolution in the background.

**BT\_App\_TraceGetter:** The BT\_App\_TraceGetter class forms one of the GUI’s a user will be able to interact with in order to operate the system. The BT\_App\_TraceGetter GUI provides the user with the functionality to create trace files for the best behaviour trees of an evolution run. Trace files are used for the playback of a given simulation. Components of the GUI include:

* JComboBox components for the following:
  + Opponent - Allows the user to set the opponent parameter, specifying the opponent controller to use within simulations.
* JButton components for the following:
  + Choose statistics files - Allows the user to select multiple statistics files containing the best trees of an evolution run.
  + Choose map - Allows the user to choose a map.
  + Generate Traces - Initiates trace creation.
* JTextArea which uses a custom output stream redirecting all error and system output to the GUI JTextArea.

There’s a method actionPerformed which accepts an ActionEvent provided by an ActionListener attached to each of the JButton and JComboBox components.

* If the source of the ActionEvent was the ‘Choose map’ button a JFileChooser is used to allow a user to select a state to be used in the simulation. The absolute path of this state file is assigned as the value of a String mapPath variable. This state is necessary to set up the scenario including for both controller: units, unit positions and the map size.
* If the source of the ActionEvent was the ‘Opponent’ JComboBox, the value of the selected JComboBox is converted to a String and assigned to the variable opponentName.
* If the source of the ActionEvent was the ‘Choose statistics file(s)’ JButton a JFileChooser is used to allow a user to select multiple ‘stat.out’ statistics files. These files are then stored within a File array files.
* If the source of the ActionEvent was the ‘GenerateTraces’ JButton, a thread is initialised as a MySecondThread class and passed the arguments for files, mapPath and opponentName. It’s start method is called to run simulations based on the given values and output trace files.

**CustomOutputStream:** A Java Class extending the OutputStream which is used to redirect output to a JTextArea within the BT\_App and BT\_App\_TraceGetter classes.

## Preconditions for Software

### Pre-conditions for system start-up

The following pre-conditions must be met for system start-up:

* To run BT\_App, BT\_App\_TraceGetter, and MicroRTS\_States\_Traces:
  + Java Runtime Environment (JRE) 8 (or higher) must be installed
  + The user has executed either BT\_App, BT\_App\_TraceGetter or MicroRTS\_States\_Traces

### Pre-conditions for system shutdown

The following pre-conditions must be met for system shutdown

* All pre-conditions within section 3.3.1 have been satisfied
* For a clean shutdown:
  + BT\_App has completed the current evolution run and the GUI window is closed.
  + BT\_App\_TraceGetter has completed trace generation and the GUI window is closed.
  + MicroRTS\_States\_Traces GUI window is closed
* Shutdown while interrupting the system:
  + While BT\_App is performing computational evolution the GUI window is closed.
  + While BT\_App\_TraceGetter is performing trace generation the GUI window is closed.
  + Shutting down the system in this way will result in partial/incomplete evolution statistics files/trace files.

### Pre-conditions for system modification

* + - * Java Development Kit (JDK) 8 (or higher) must be installed.
      * An Integrated Development Environment compatible with Java, preferably Eclipse IDE 2018-09 v.4.9.
      * System is installed correctly as per installation instructions.

# Conclusions

## State of the system as delivered

The system as delivered meets all the functional requirements specified, meaning it generates behaviour trees for use in AI decision making. As well assisting in the discovery of novel strategies, with the ability to generate and replay traces to examine the resulting strategies prevalent in the best behaviour trees generated.

The system also met most of the non-functional requirements originally specified. The system and all its components were coded with the Java programming language. As well as making use of the Evolutionary Computational Toolkit and its ability to perform computational evolution to evolve an AI through its behaviour trees. The behaviour trees provided AI controlled units with the ability to make decisions pertaining to movement, attacking, idling, detecting enemies and enemy structures as well as detecting allies. The system employs a custom MicroRTS AI controller which provides the facilities to:

* + - * Control a team of 8-12 units or greater against an enemy of a similar size.
      * Operate within the confines of MicroRTS in any type of simulation
      * Function on at least a single map consisting of the dimensions 24 x 24.

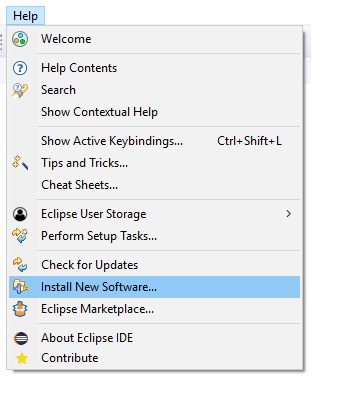
The system was not however constructed within the NetBeans IDE v.8.2 software as originally specified. This was due to the constraints of ECJ and the decision was made to change to the Eclipse IDE at the suggestion of the ECJ author. This meant the ECJ module was much easier to work with, as many tutorials that existed for the Eclipse IDE and ECJ were now readily available. It was also determined that the Linux based operating system originally thought to be useful for running ECJ was obsolete with the introduction of the Eclipse IDE.

## Future considerations:

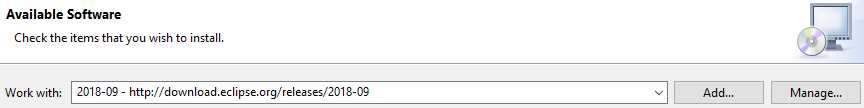
Future considerations in order of priority for future release include:

1. Combine all three GUI’s into one comprehensive GUI
2. Allow a user to specify multiple parameters within BT\_App for running multiple evolutions over e.g. multiple different crossover rates
3. Implement a more robust input validation for the GUI applications
4. Implement a more robust computation budget tournament
5. Implement additional nodes to increase complexity
6. Experiment with the perfect mutation parameters for KozaNodeSelector
7. Implement additional GUI elements pertaining to the number of maximum game cycles, simulation visualisation and simulation delay

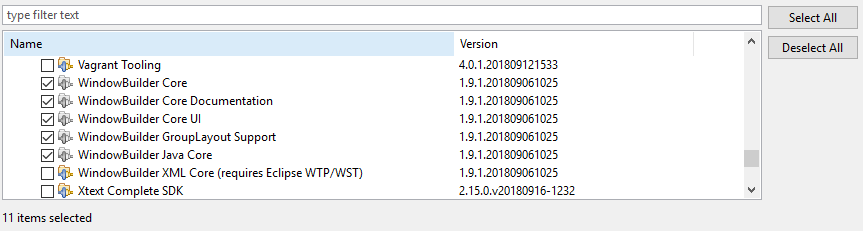
# Appendix A: Installation



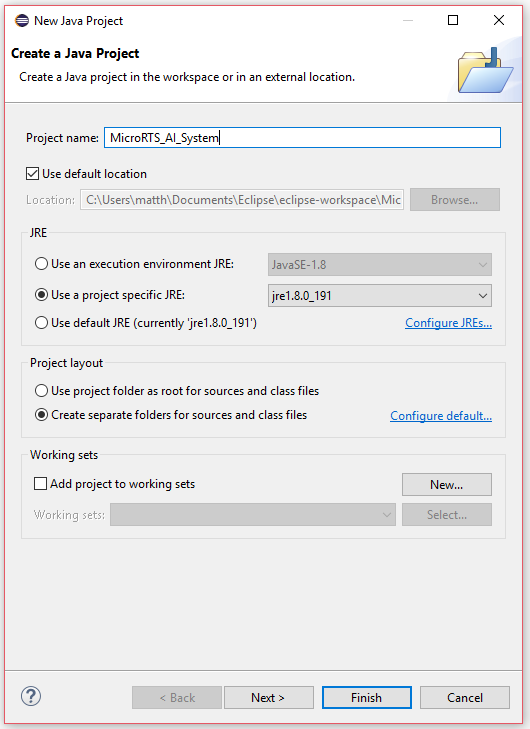
*Figure 4 – Installation - Install New Software*



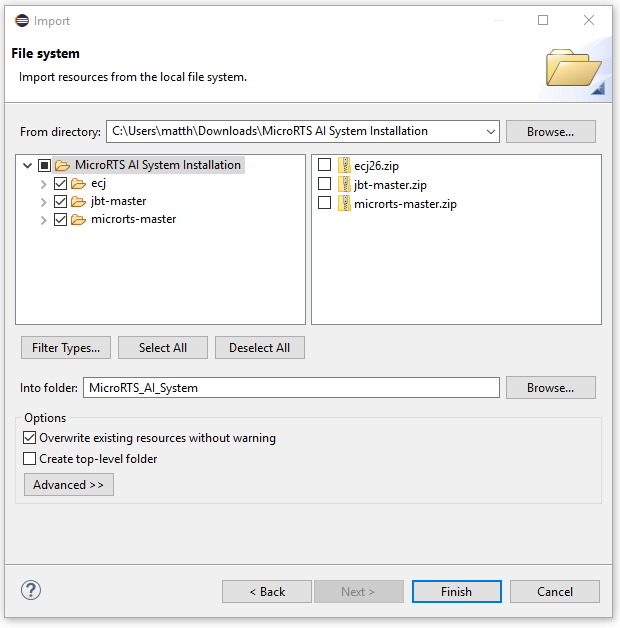
*Figure 5 – Installation - Work with current Eclipse IDE version*



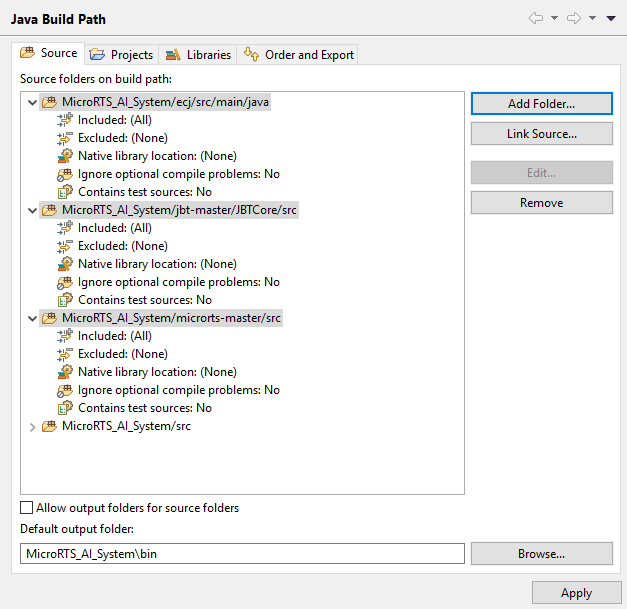
*Figure 6 – Installation - Select Swing and WindowBuilder plugins*



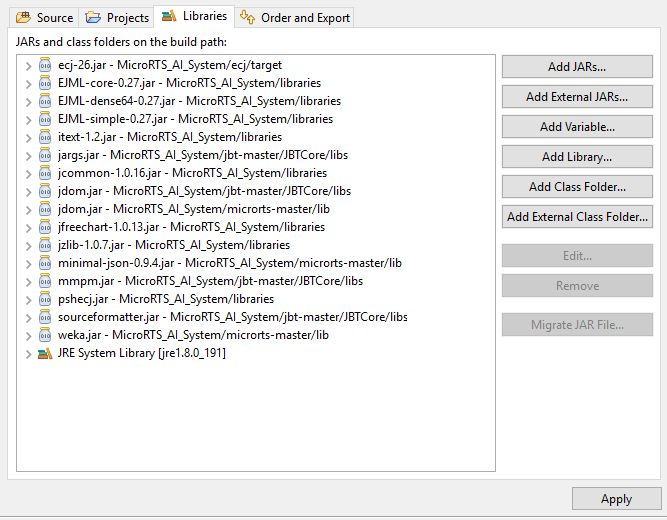
*Figure 7 – Installation - Create a Java Project*



*Figure 8 – Installation - Import Existing Module File Systems*



*Figure 9 – Installation - Java Build Path Source Files*



*Figure 10 – Installation - Java Build Path Libraries JAR Files*