Norwegian University of
Science and Technology

Automating Behaviour Modelling for Computer Generated Forces

Evolving Behaviour Trees with Observational Learning

Gabriel Berthling-Hansen
Eivind Morch

Master of Science in Informatics
Submission date: July 2018
Supervisor: Odd Erik Gundersen, IDI
Co-supervisor: Rikke Amilde Løvlid, Forsvarets forskningsinstitutt

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Computer generated forces are simulated entities that are used in simulation based training and decision support in the military. The behaviour of these simulated entities should be as realistic as possible, so that the lessons learned while simulating are applicable in real situations. However, it is time consuming and difficult to build behaviour models manually, and there has been an increasing interest in automating this process using machine learning.

In this thesis, we investigate whether behaviour modelling for computer generated forces in complex, realistic simulation environments can be automated by using genetic programming to generate behaviour trees based on example behaviour. A system for evolving behaviour trees was implemented in order to evaluate the investigated method. The system is able to connect to a complex simulation system over high level architecture, and to use it for recording example behaviour and simulating behaviour trees. With the use of this system, we were able to generate behaviour trees that are close to identical to simple observed behaviour. The used example behaviour was recorded from a simulated entity which was controlled by a manually created behaviour tree. The results suggest that the investigated method works for modelling simple behaviour, but further work is required to evaluate how it performs with more complex behaviour.

The research presented in this thesis has three contributions: (i) proof that genetic programming and behaviour trees can be used to mimic recorded, simple behaviour in complex, realistic simulations; (ii) a proposed set of methods for mutating behaviour trees; and (iii) a modular system for using genetic programming to evolve behaviour trees through observational learning with a complex, realistic simulation system over high level architecture.

Sammendrag

Datagenererte styrker er simulerte enheter som er brukt i simuleringsbasert trening og beslutsningsstøtte i militæret. Oppførselen til disse simulerte enhetene bør være så realistisk som mulig, slik at lærdommen fra disse simuleringene kan brukes i virkelige situasjoner. Det er tidkrevende og vanskelig å bygge oppførselsmodeller manuelt, og det har vært økende interesse for å automatisere denne prosessen ved hjelp av maskinlæring.

I denne oppgaven undersøker vi om oppførselsmodellering for datagenererte styrker i komplekse, realistiske simuleringsmiljøer kan automatiseres ved å bruke genetisk programmering for å generere oppførselstrær ut fra eksempeloppførsel. Det ble laget et system for å utvikle oppførselstrær gjennom evolusjon for å kunne evaluere den undersøkte metoden. Systemet kan kobles til et komplekst simuleringssystem over high level architecture, og bruke det til å ta opp eksempeloppførsel og simulere oppførselstrær. Ved å bruke dette systemet fikk vi til å generere oppførselstrær som er nær identiske med enkel observert oppførsel. Eksempeloppførselen som ble brukt ble tatt opp fra en simulert enhet som ble styrt av et manuelt bygget oppførselstre. Resultatene antyder at den undersøkte metoden fungerer for modellering av enkel oppførsel, men det er nødvendig med ytterligere arbeid for å vurdere hvor godt den fungerer med mer komplisert oppførsel.

Forskningen som presenteres i denne oppgaven har tre bidrag: (i) bevis på at genetisk programmering og oppførselstrær kan brukes til å imitere observert, enkel oppførsel i kompliserte, realistiske simuleringer; (ii) et foreslått sett med metoder for å mutere oppførselstrær; og (iii) et modulært system for bruk av genetisk programmering til å utvikle oppførselstrær ved læring gjennom observasjon med et komplekst, realistisk simuleringssystem over high level architecture.

Preface

This thesis has been authored by students at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU). The research was conducted in cooperation with the Norwegian Defence Research Establishment (FFI), as part of a larger project where the intent is to use the contributions of this project for further research on automating behaviour modelling. The thesis was supervised by Odd Erik Gundersen, an Associate Professor at NTNU, Rikke A. (Løvlid) Seehuus, a senior scientist at FFI, and Martin Asprusten, a scientist at FFI. MÄK provided us with student licenses for the simulation software that was used during the research. This includes a simulation engine, a simulation GUI, a Run-Time Infrastructure (RTI) and VR-Engage, among others.

Gabriel Berthling-Hansen
Eivind Morch

Oslo, July 15, 2018

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Problem Outline . . . 2
1.3 Hypothesis and Research Questions . . . 4
1.4 Research Approach . . . 5
1.5 Research Contributions . . . 6
1.6 Thesis Structure . . . 7

2 Background **9**
2.1 Data-Driven Behaviour Modelling . . . 9
2.2 Behaviour Trees . . . 10
2.3 Genetic Algorithms and Genetic Programming . . . 12
2.4 High Level Architecture . . . 13
2.4.1 Time Management . . . 13
2.5 MÄK Simulation Systems . . . 15
2.5.1 VR-Forces . . . 15
2.5.2 VR-Engage . . . 16

3 State of the Art **19**
3.1 Observational Learning with Toy-Problems . . . 19
3.2 Behaviour Evaluation in Complex Simulations . . . 20
3.3 Observational Learning in Complex Simulations . . . 21
3.4 Evolving Behaviour Trees . . . 22
3.5 Bloat-Control with Genetic Programming and Behaviour Trees . . . 23
3.6 Summary . . . 24

4 Methods **27**
4.1 System Overview . . . 27
4.1.1 Recording Data from Example Behaviour . . . 29
4.1.2 Training Behaviour Trees . . . 29
4.2 Simulation Environment . . . 29
4.3 Data and Feature Extraction . . . 30

4.4 Behaviour Tree Representation 31
4.5 Evolving Behaviour Trees 31
4.5.1 NSGA-II 31
4.5.2 Genetic Operators 32
4.5.3 Behaviour Tree Evaluation and Bloat Control 35
4.6 Summary 37

5 Implementation **39**
5.1 System Interface 39
5.1.1 Console Logging Output 39
5.1.2 Training Progress Window 40
5.2 System Components 40
5.2.1 Simulation Package 46
5.2.2 Unit Package 47
5.2.3 Data Package 49
5.2.4 Training Package 50
5.2.5 Behaviour Tree Package 52
5.2.6 Visualisation Package 55
5.2.7 Experiments Package 55
5.3 Setting up an Experiment 56
5.4 System Processes 58
5.4.1 Sequence Diagram Explanation 58
5.4.2 Initiation Process 59
5.4.3 Simulation Process 59
5.4.4 Recording Process 60
5.4.5 Training Process 61
5.5 Settings 70
5.5.1 System Settings 70
5.5.2 Simulation Settings 70
5.5.3 Behaviour Tree Operations Settings 70
5.5.4 Training Settings 71
5.6 System Logging 71
5.7 Libraries Overview 72
5.8 Summary 72

6 Experiments and Results **73**
6.1 Experimental Plan 73
6.2 Experiment 1 74
6.2.1 Data Extraction and Processing 74
6.2.2 Behaviour Tree Nodes 75
6.2.3 Scenarios 75
6.2.4 Fitness Evaluation 77
6.2.5 Settings 78
6.3 Experiment 2 78

6.4 Results 79
6.4.1 Experiment 1 79
6.4.2 Experiment 2 79

7 Evaluation **85**
7.1 Evaluation of the Experiments 85
7.1.1 Evaluation of Experiment 1 85
7.1.2 Evaluation of Experiment 2 87
7.2 Evaluation of the System 89
7.3 Research Questions Revisited 90
7.4 Evaluation of the Contributions 92

8 Conclusion and Future Work **95**
8.1 Conclusion 95
8.2 Future Work 96

Bibliography **99**

Glossary **103**

Appendices 105

A CogSIMA2018 Article and Poster **107**

B Single-Page Figure Versions **117**

C VR-Forces Settings **123**

D Literature Review Notes **125**

List of Tables

1.1 RQ2 system requirements 5

5.1 Entity colour-coding in system diagrams (Figures 5.4, 5.5 and 5.7 to 5.10) 43
5.2 Sequence diagram fragment explanations (used in Figures 5.7 to 5.10) 59
5.3 Simulation settings used internally in the system 70
5.4 Mutation settings 71

6.1 NSGA-II settings used for Experiment 1 78
6.2 Comparison of the Scenario 1 fitness values in Experiment 1 and Experiment 2 80
6.3 Comparison of the Scenario 2 fitness values in Experiment 1 and Experiment 2 80

List of Figures

2.1 Graphical representation of a sequence node with N children 11
2.2 Visualisation of sequence node execution 11
2.3 Graphical representation of a selector node with N children 11
2.4 Visualisation of selector node execution 11
2.5 Graphical representation of behaviour tree leaf node types 12
2.6 Scenario showing one of the issues related to time management in distributed simulation 14
2.7 Virtual terrain simulated in VR-Forces 15
2.8 VR-Forces GUI with 2D view of a simulation scenario 16

4.1 Process flow of the complete process of recording example behaviour and generating behaviour models through observational learning . . 28
4.2 HLA federation overview shown in MÄK RTI Assistant 30
4.3 Crossover 32
4.4 Add Random Subtree 34
4.5 Randomise Variables of Random Node 34
4.6 Remove Random Subtree 35
4.7 Replace Random Node With Node of Same Type 35
4.8 Replace Tree With Subtree 36
4.9 Switch Positions of Random Sibling Nodes 36

5.1 System console logging output during training 41
5.2 Training visualisation window with "Old population" tab open . . . 42
5.3 Training visualisation window with "Fitness history" tab open 42
5.4 Architecture overview of the system connected to MÄK VR-Forces . 43
5.5 Class diagram of the core system and the experiment package 45
5.6 Communication sequence diagram of HLA communication per system tick during simulation 61
5.7 Sequence diagram of the system initiation process 62
5.8 Sequence diagram of the simulation process 64
5.9 Sequence diagram of the recording process. References Figures 5.7 and 5.8 as subprocesses. 66

5.10 Sequence diagram of the training process. References Figures 5.7 and 5.8 as subprocesses. 68

6.1 Visualisation of approaching angle calculation used in Experiment 1 76
6.2 2D view of scenario terrain and target path for the scenarios used in Experiment1. 77
6.3 Manually made behaviour tree 78
6.4 Experiment 1 fitness development over 90 epochs 81
6.5 Zoomed view of Experiment 1 fitness development on Scenario 2 over 30 epochs 82
6.6 Graphical representations of two of the resulting behaviour trees in Experiment 1 after 30 epochs 82
6.7 Experiment 2 fitness development over 148 epochs 83
6.8 Zoomed view of Experiment 2 fitness development over 148 epochs . 84
6.9 Graphical representation of the overall best generated behaviour tree in Experiment 2 84

B.1 Single-page version of Figure 5.5 118
B.2 Single-page version of Figure 5.7 119
B.3 Single-page version of Figure 5.8 120
B.4 Single-page version of Figure 5.9 121
B.5 Single-page version of Figure 5.10 122

Chapter 1

Introduction

This thesis is about automating behaviour modelling for computer-controlled entities in military simulations. It documents our experiments with using genetic programming (GP) to generate behaviour models – specifically, behaviour trees – that represent specific, human-like behaviour. As part of the research, a poster article was submitted and accepted to the 2018 Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA 2018), and will be published in the IEEE Xplore Digital Library. The article and poster is included in Appendix A. In this first chapter, the motivation and context of the research, the hypothesis and research questions, and the research approach and contributions are introduced.

1.1 Background and Motivation

Computer generated forces (CGFs) are autonomous or semi-autonomous entities that represent military units, such as tanks, soldiers and combat aircrafts, in simulation software for military operations. CGFs are similar to non-player characters in computer games and are used in military simulation-based training and decision support applications. CGFs enable simulating large military operations as one operator is able to control several military units. The behaviour of the CGFs, e.g. how they move, where they look, when they shoot, should represent the behaviour of corresponding human soldiers or manned systems as accurately as possible. Ideally, a soldier training with a virtual simulator should not notice whether his teammates or opponents are human controlled entities or CGFs. Realistic CGF behaviour also makes it possible to simulate various plans or courses of actions to improve the situation awareness and get a good understanding of how a situation could play out [1]. Simulations can for instance help build and train the mental model of the trainee by practising situation comprehension and projection, situation awareness level two and three in Endsley's model of situation awareness [2]. In aviation, around 20% of the errors are related to problems with the mental model according to Endsley and Garland [3]. Given that errors made by soldiers in a stress

situation are similar to those made in aviation, improving their mental model is of high importance. This requires the CGFs to behave in a natural way, as their behaviour affects the situation comprehension and projection of the trainee.

There are several ways to represent the behaviour of CGFs. The most common way is to use state machines that describe different states that the CGFs can be in and the actions they can perform in every state. Lately, however, behaviour trees have grown popular [4, 5, 6, 7]. In any case, the behaviour models are typically made manually. This means that military experts first have to tell programmers how they want the CGFs to behave, and that the programmers then have to translate those descriptions into code. This is a difficult and time consuming process [8], and requires the presence of domain experts.

Lately, there has been an increasing interest in automating this process using machine learning [9]. The use of machine learning to model behaviour is also referred to as data-driven behaviour modeling (DDBM), and can be done by e.g. first recording data from a demonstration of desired behaviour and then using machine learning to train an agent based on the recorded data. As emphasised by Luotsinen et al. [10] and Løvlid et al. [11], there are significant challenges with automating this process, but the potential payoff in increased training efficiency and reduced costs is significant. Observational learning, a machine learning technique with inspiration from how humans and other mammals learn from observing others, has been argued to be especially suited for learning tactical knowledge – knowledge on how to act given the current situation [12]. Observational Learning has been successfully used to learn personal behaviour traits from the observed actors, which again can be used to create more unpredictable and realistic behaviour models [13, 14]. In any case, successfully applying machine learning for automating behaviour modelling for real-world problems can reduce the amount of resources and number of domain experts required to support simulated exercises. For instance, it might enable military end-users to create CGF behaviour by demonstrating the desired behaviour themselves [11].

1.2 Problem Outline

Manual modelling of CGF behaviour is expensive, time consuming, and requires the engagement of domain experts. Automating this process has several significant benefits, such as lowered costs and increased availability of behaviour models. These benefits may again result in better training efficiency, reduce the number of people required to create and manage simulated exercises, and make running simulated exercises with larger forces more feasible.

Modelling realistic human-like behaviour is also difficult. When using human experts to identify the behaviour and then communicate it to programmers, the resulting model is often logical, but also lacking of personal characteristics, and carry little resemblance to real human behaviour [14]. Realistic CGF behaviour is important for providing realistic simulated environments for training and analysis

of possible situational outcomes. DDBM is believed to be more efficient at creating objective and realistic behaviour models [11].

In this thesis, the focus is on using GP to generate behaviour trees based on example behaviour, as a way of automating behaviour modelling for CGFs in complex simulation environments. GP is a type of machine learning that is based on the Darwinian principle of survival of the fittest, where candidate solutions are selected for breeding and survival based on their performance. They are not typically used for observational learning, and to our knowledge, there is no existing software that enables using GP to generate behaviour trees from observing behaviour in complex simulations.

The conducted research is part of a larger project at the Norwegian Defence Research Establishment (FFI) on automating CGF behaviour modelling with DDBM. FFI requested that any communication with simulation systems is done over High Level Architecture (HLA) – an interoperability standard for distributed simulation. In order to conduct experiments with the proposed method for automating behaviour modelling, it was therefore necessary to develop a new system that would be able to extract simulation data from a simulation system over HLA, generate behaviour models in the form of behaviour trees from the extracted data, and then evaluate the generated models by controlling simulated entities in the simulation system by using the HLA infrastructure. Developing a system that has the ability to connect to distributed simulation systems over HLA enables it to use a large variety of simulation engines and equipment for controlling simulated entities. It also enables the system to be used with the equipment that is already used for simulated military exercises today. However, it introduces challenges with time management, data extraction, controlling the actions of the simulated units, among others. For our research, we used a real, military simulation system called VR-forces[1] from MÄK.

A potential problem when creating behaviour models with machine learning, is that the model often becomes opaque, meaning it becomes hard to interpret the represented behaviour. Løvlid et al. [11] present this as maybe the most significant drawback of generating CGF behaviour models with machine learning, and it has been addressed in initiatives such as the Explainable Artificial Intelligence (XAI)[2]. This problem is e.g. often experienced with artificial neural networks (ANNs), which quickly turn into black-boxes as the complexity of their non-symbolic knowledge representations increases. In our work, we focus on trying to learn behaviour trees, i.e. the same type of model that can be used to model the behaviour manually. By using behaviour trees, the learned model for a CGF is explicit, which enables and simplifies explaining the behaviour. However, despite being used for manual behaviour modelling, complex behaviour trees are not necessarily easy to analyse. This is especially true for computer generated trees, that may seem arbitrarily structured to humans.

Running simulations in complex, realistic environments is often computation-

[1]https://www.mak.com/products/simulate/vr-forces

[2]https://www.darpa.mil/program/explainable-artificial-intelligence

ally heavy. In fact, some of the systems used today, such as Virtual Battlespace 3 (VBS3)[3], are only able to run simulations in realtime [10]. GP typically depends on frequent evaluations of candidate solutions, and using a GP to generate behaviour models that need to be tested in complex simulations can quickly result in simulation becoming a bottleneck. It is therefore necessary to consider the number of behaviour evaluations that can be done during training depending on the available processing power.

1.3 Hypothesis and Research Questions

The hypothesis (HYP) underlying this thesis is:

HYP: *The process of creating behaviour models for CGFs can be automated by replacing manual behaviour analysis and programming with GP that generates behaviour trees from observing examples.*

Currently, behaviour models for CGFs are manually designed and programmed. This is hard and time consuming. We have investigated the use of GP and behaviour trees to automate this process.

The hypothesis is divided into two research questions (RQ). The first research question, RQ1, relates to the research on how GP and behaviour trees perform in imitating observed behaviour. The second research question, RQ2, relates to the development of a completely new system for connecting to third-party distributed simulation systems and using them for data extraction and behaviour training.

RQ1: *How do behaviour trees generated with GP perform in imitating observed behaviour in complex, realistic environments?*

The goal of this thesis is to investigate whether the CGF behaviour modelling process can be automated by using GP to evolve behaviour trees from observations of example behaviour. CGFs are used in military simulations, with highly complex and realistic simulated environments. Therefore, in order to answer the hypothesis, we need to research the feasibility of using GP for generating behaviour trees through observational learning in complex, realistic environments.

RQ2: *How should a system for generating behaviour trees with GP be designed to be used with an external simulation system?*

In order to answer RQ1, we needed to develop a system that would enable us to run experiments using GP to generate behaviour trees based on observed behaviour in complex simulations. The system had to satisfy three requirements, shown in Table 1.1. FFI requested that the system should use HLA for communication with the simulation system, forming the first requirement. They also required

[3] https://bisimulations.com/products/virtual-battlespace

that it should be able to extract data from the simulation system and then use the simulation system for evaluating generated behaviour models, forming the second requirement. The system was intended to be used for multiple experiments, resulting in the third requirement.

Table 1.1: RQ2 system requirements

Requirements
1. The system should use HLA for communication with an external simulation system.
2. The system should be able to extract data from the simulation system, and then use the simulation system for evaluating generated behaviour models.
3. The system should support running different experiments with different scenarios, different simulated entities with different possible actions, and different types of data.

1.4 Research Approach

The research documented in this thesis has been done in three phases:

1. **Literature review:** First, a literature review was performed. During the literature review, we focused on researching observational learning with both toy-problems and complex simulations, evaluating behaviour in complex simulations, evolving behaviour trees, and bloat-control with genetic programming and behaviour trees. The literature review was done in order to become familiar with the state of the art, as well as to help identify how the system should be designed and what methods that should be applied for evolving behaviour trees. Notes from the literature review is included in Appendix D.

2. **System design and implementation:** After the literature review was finished, the process of designing and implementing the system started. This phase followed a design and creation methodology [15, pp. 108–124], where the system for running experiments with GP and behaviour trees in complex simulations was developed. During this phase, we had frequent contact with our external supervisors at FFI, who assisted with issues with connecting to and controlling the simulation system over HLA. The goal of this phase was to design and implement a modular system that can be used for running different experiments, both in this and future projects.

3. **Experiments:** Finally, in order to both evaluate the system and address the hypothesis, we performed experiments. The first experiment identified issues with the extraction of data used for behaviour tree evaluation during training. It was therefore necessary to go back to the system implementation phase and fix the identified issues before continuing. However, the experiment showed that the proposed method works for imitating simple behaviour. This phase was planned to include multiple experiments, but was cut short due to the time required to fix the identified issues with the system. After the issues with data extraction had been addressed, the first experiment was re-done for a new evaluation of the system.

1.5 Research Contributions

The research presented in this thesis has three contributions, which are introduced in this section. An evaluation of the contributions is given in Section 7.4.

C1: *Proof that GP and behaviour trees can be used to mimic recorded, simple behaviour in complex simulations.*

The results from our experiments show that GP and behaviour trees can be used to successfully mimic simple behaviour in a complex simulated environment. While we can not guarantee that our solution will work for more complex behaviour, the results are promising. The results work as a proof of concept, and motivates further research on the subject.

C2: *A proposed set of methods for mutating behaviour trees.*

In this thesis, we propose a set of methods for performing mutations on behaviour trees. A mutation is a genetic operation which alters an existing candidate solution. To our knowledge, most of the methods for mutating behaviour trees described in this thesis have not been discussed in current research. Although several authors have used mutations for evolving behaviour trees, they typically use one or two mutations [5, 4]. In our research, we propose multiple independent behaviour tree mutations, as well as a technique for combining the use of different mutations when evolving behaviour trees. With inspiration from simulated annealing (SA) [16, p. 128], this technique uses scaling probability weights to select which mutation method should be used, favouring mutations that make smaller changes as time passes.

C3: *A modular system for using GP to evolve behaviour trees through observational learning with a complex, realistic simulation system over HLA.*

The system can be used for running different experiments, with different types of data, different types of simulated entities, different algorithms and potentially with

different simulation systems. The system has been primarily designed for observational learning, but should also be able to support experiential learning due to its modular structure. The system is intended to be used for further research on automating behaviour modelling for CGF as part of a larger project at FFI. The system is published as open source at `https://github.com/eivinmor/msc-gbh-em` under the MIT license[4].

1.6 Thesis Structure

This thesis is divided into eight chapters. Chapter 1 motivates the research, outlines the problem, and introduces the hypothesis, research questions and contributions. Chapter 2 covers relevant background theory. Chapter 3 describes related work and the state of the art. Chapter 4 covers the methods used in order to to answer the hypothesis and research questions of this thesis. Chapter 5 provides a detailed description of the implemented system. Chapter 6 covers the experimental plan, the conducted experiments and the results. Chapter 7 contains the evaluation of the experiments, implemented system, research questions and contributions. And finally, Chapter 8 presents the conclusion and suggestions for future work.

[4] `https://opensource.org/licenses/MIT`

Chapter 2

Background

This chapter contains an overview of the most important topics discussed in this thesis. The following chapters have been written with the assumption that the reader is familiar with the information that is presented here. The topics covered in this chapter are DDBM, behaviour trees, genetic algorithms (GAs) and GP, HLA, and MÄK simulation systems.

2.1 Data-Driven Behaviour Modelling

DDBM refers to the use of machine learning to automate the creation of behaviour from recorded data [17]. It is believed that replacing the manual work of human domain experts with DDBM will result in more objective and realistic behaviour models [11]. The learning techniques used in DDBM are usually divided into observational learning, experiential learning and a hybrid learning approach.

Observational learning is the process of learning from observing the behaviour of another agent, and is often used to create behaviour models that imitate human behaviour. The learning agent is trained to behave as similar as possible to the observed agent. The behaviour is trained on the same activities as the observed agent, and its performance is evaluated based on how much it resembles that of the observed agent. Observational learning is also commonly referred to as learning from examples, learning from imitation, and learning from demonstration.

Experiential learning is the process of learning from experience. This method does not use any example of the "correct" behaviour, but evaluates the performed actions based on experiences of their consequences. Experiential learning is typically used for finding the optimal way of achieving a specific goal or minimising a specific evaluation metric. Human behaviour is influenced by subjective bias, and is therefore often different from the objectively optimal behaviour learned through experiential learning. Behaviour learned through experiential learning may therefore seem unnatural or unrealistic if used for CGF.

The hybrid approach uses both observational and experiential learning. It uses

observational learning to learn the observed behaviour, which may include the personal behaviour traits of a human, and then uses experiential learning to improve the learnt behaviour to be more optimal based on specific evaluation metrics, or to generalise the behaviour for different environments and problems.

2.2 Behaviour Trees

We use behaviour trees to represent the CGF behaviour. Behaviour trees are trees of hierarchical nodes that control decision making and task execution, and have been popularly used for modelling the behaviour of computer-controlled units in video games [4]. They provide a scalable and modular solution for representing complex behaviour without the exponential complexity of Finite State Machines (FSM) and reusability-problem of Hierarchical Finite State Machines (HSFM) [5]. The modular structure of behaviour trees allows for easy manipulation of the represented behaviour, and the ability to use one behaviour tree as a subtree of another makes them highly reusable. Behaviour trees are also human-readable, giving the opportunity for visual analysis of the represented behaviour.

Behaviour tree nodes can return one of three statuses: *running*, *success* or *failure*. Running means that the node is currently active, has not completed its tasks and needs more time to finish. Success is returned when a node is finished executing and its task was successfully completed, and failure is returned when the task finished unsuccessfully.

Behaviour trees are traversed from the root node and down. If all visited nodes are finished, the tree will be traversed from the root and down again on the next timestep. However, if one of the nodes return running, the tree will keep running that node every timestep until it returns either failure or success. Once the node is done, the tree will continue traversing from the position of the node.

Composite Nodes

A composite node is used to group nodes into a higher level task [5]. The type of the composite node dictates in which order it will execute children nodes, when to stop, and what status to return. The system described in this article uses two types of composite nodes, *sequence* and *selector*.

A sequence node, shown in Figure 2.1, executes its children from left to right until one of the children returns failure or all return success. If a child returns failure, then the sequence will stop and return failure. If all its children return success, it will return success. Figure 2.2 shows a visualisation of the sequence node execution.

A selector node, shown in Figure 2.3, executes its children from left to right until one of the children returns success or all children return failure. If a child returns success, the selector will stop and return success. If all its children return failure, the selector will return failure. Figure 2.4 shows a visualisation of the selector node execution.

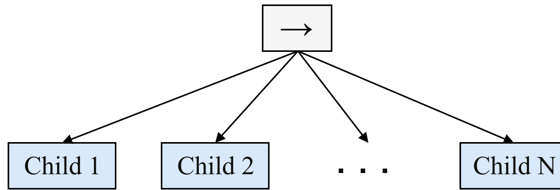


Figure 2.1: Graphical representation of a sequence node with N children

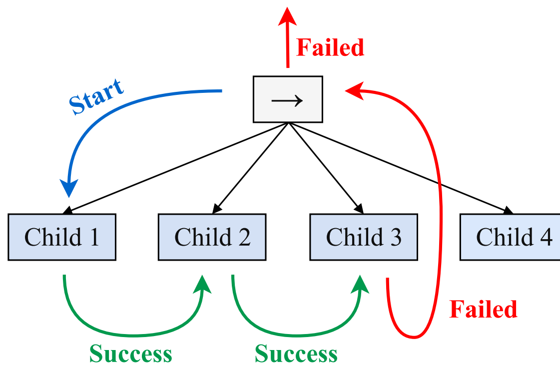


Figure 2.2: Visualisation of sequence node execution

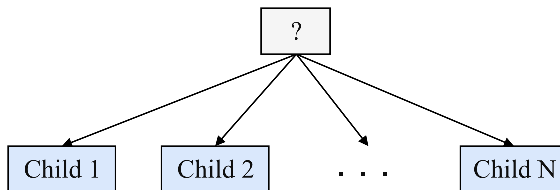


Figure 2.3: Graphical representation of a selector node with N children

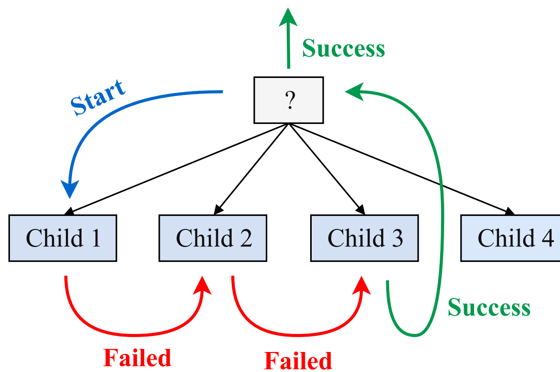


Figure 2.4: Visualisation of selector node execution

(a) Action node **(b)** Condition node

Figure 2.5: Graphical representation of behaviour tree leaf node types

Leaf Nodes

A leaf node has no children, and is either an action node or a condition node. An action node is used to perform a specific low-level action, e.g. move to a certain location. A condition node returns success or failure based on some condition, e.g. whether an object is within a specific distance or not. Figure 2.5 shows the graphical representation of the action node and the condition node.

Blackboard

A blackboard contains data that is accessible for all the nodes of the behaviour tree. Nodes may also alter data inside the blackboard. A blackboard is an important feature of a behaviour tree as it enables nodes to share and alter the same state representation, avoiding an exponential state complexity such as in FSMs.

2.3 Genetic Algorithms and Genetic Programming

GAs are stochastic search algorithms inspired by evolution. A GA generates and evolves a population of chromosomes, where each chromosome is a candidate solution for solving a problem. Chromosomes are assigned a value representing how well they solve the problem, called fitness. For each iterative step of the algorithm, called an epoch, a GA typically produces a new generation of chromosomes through crossover, mutation and selection. Crossover is the creation of a new chromosome by combining the traits of two existing chromosomes to produce a hybrid solution. The chromosomes can also be randomly mutated, making direct changes to existing solutions. Selection is the process of deciding which chromosomes should be used for crossovers and which chromosomes that should survive to the next generation, and is usually done by comparing fitness values. See [16, pp. 129–132] for more information on GAs.

Multi objective genetic algorithms (MOGAs) are GAs that optimise candidate solutions for multiple objectives by using multiple fitness values simultaneously. For comparing candidate solutions, MOGAs often use the concept of dominance, where a solution is said to dominate another solution if it is equal or better on all fitness objectives. The set of non-dominated solutions form the Pareto-front [18].

In 1992, Koza [19] introduced GP as an extension of the GA, where the evolved candidate solutions are computer programs. In this thesis, we define GP as the use of GAs to evolve computer programs represented as tree structures, where behaviour trees are defined as a type of computer program. A common problem

when working with GP, is that that the generated trees tend to grow in size without significant return in terms of fitness, called *bloat* [20]. This can cause several issues, including poor generalisation and reduced effectiveness of crossover and mutation operators, and should be addressed when working with GP.

2.4 High Level Architecture

HLA is a communication standard for distributed simulation systems, originally developed by the US Department of Defence [21]. It provides a general purpose architecture for interoperation between simulation engines, user interfaces, passive data collectors, and other entities used in simulation networks. A network of entities communicating over HLA is called a *federation*, where each connected entity is called a *federate*. All federates in a federation must agree on data and data formats to exchange. This is formalised in a Federation Object Model (FOM). All interactions between the federates go through a Run-Time Infrastructure (RTI), which provides a set of services to the federates. Among these services are management of the federates' ownership of simulated objects, management of publishing and subscribing to data updates and interactions, data distribution management and time management.

2.4.1 Time Management

In distributed simulations, there are three different types of time: *physical time* – the time in the simulated scenario, *simulation time* – the logical time used by the simulator to represent the physical time, and *wallclock time* – the real-world time when the simulation is executed [22]. Uncoordinated time advancement in the simulation engines and uncoordinated sending of events between them can result in temporal anomalies. An example of this is shown in Figure 2.6. A unit in Simulator A shoots and destroys a target in Simulator B. Both the "shoot event" and the "target destroyed event" are published to the other simulators in the network as they occur in the simulations. However, the "shoot event" is delayed as it travels through the network from Simulator A to Simulator C, and Simulator C is therefore notified that the target in Simulator B has been destroyed before it receives the "shoot event" from simulator A.

In HLA, each federation has a global simulation time which is used to manage communication and simulation advancement between the federates. Messages between the federates are controlled and distributed by the RTI, and can be sent both with and without a timestamp. The RTI also controls the time advancement of the federates, with the purpose of ensuring that no federate will receive messages with a timestamp less than its current logical time. In order to ensure this, the RTI needs to calculate what the smallest possible timestamp is for future messages, and then prevent any federate that receives timestamped messages from advancing past that time.

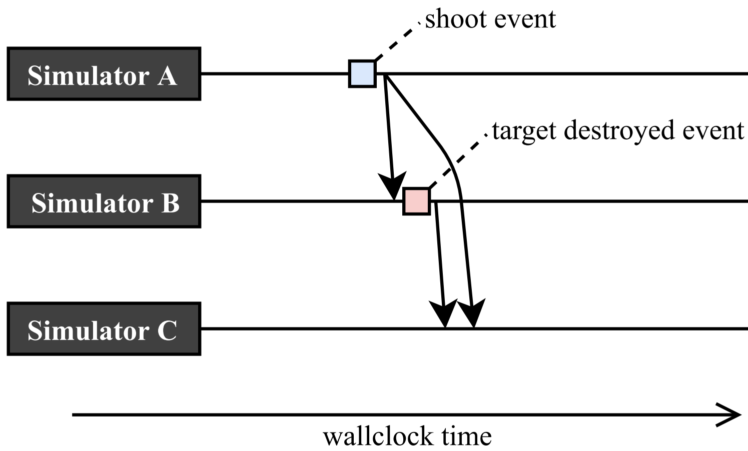


Figure 2.6: Scenario showing one of the issues related to time management in distributed simulation. Simulator C receives is notified of the consequential event before the causing event due to lack of time management.

Lookahead

Federates have two boolean flags related to message handling with time management – *time regulating* and *time constrained*. A time regulating federate is a federate that is able to send timestamped messages, and a time constrained federate is a federate that depends on receiving messages in timestamp order. In order for the RTI to calculate the smallest possible timestamp of the messages the time regulating federates can produce, each time regulating federate is required to register a value called *lookahead* with the RTI. Lookahead is the minimum amount of time between the federates current logical time and the timestamp used for its outgoing messages. E.g., a federate with a current logical time of 20 and a lookahead of 5 is not allowed to produce messages with a timestamp lower than 25. The time constrained federates have to request permission from the RTI to advance its logical time. For federates that are not marked as time constrained, the RTI will deliver timestamped messages as if they were sent without a timestamp. To summarise, the RTI uses the logical time and lookahead of the time regulating federates to calculate how far the time constrained federates can advance their logical time before they risk receiving messages with a timestamp that is smaller than their current logical time.

For a more detailed explanation of HLA time management, see [22].

Figure 2.7: Virtual terrain simulated in VR-Forces

2.5 MÄK Simulation Systems

VT MÄK is a company that develops modelling and simulation software. Their simulation products include software for setting up and managing simulation networks, creating scenarios and entity models, and running and visualising simulations. For the research covered in this thesis, we used VR-Forces Engine, VR-Forces GUI and MÄK RTI. VR-Engaged is also mentioned as a tool for controlling simulated entities. Following is an overview of of these applications.

2.5.1 VR-Forces

VR-Forces is MÄKs complete simulation solution, including a simulation engine and a graphical user interface (GUI) which is used to create scenarios and observe and control simulations. It also includes a set of terrains, unit models and simple behaviour functions such as move to location or waypoint, patrol along route, etc. These behaviour functions can be combined and scheduled in a behaviour plan. Figure 2.7 shows an example of a virtual terrain in VR-Forces. Both the Simulation Engine and the GUI supports HLA.

VR-Forces Simulation Engine

The VR-Forces Simulation Engine handles the calculations needed to run the simulation. It creates new world states and publishes event and data updates on the

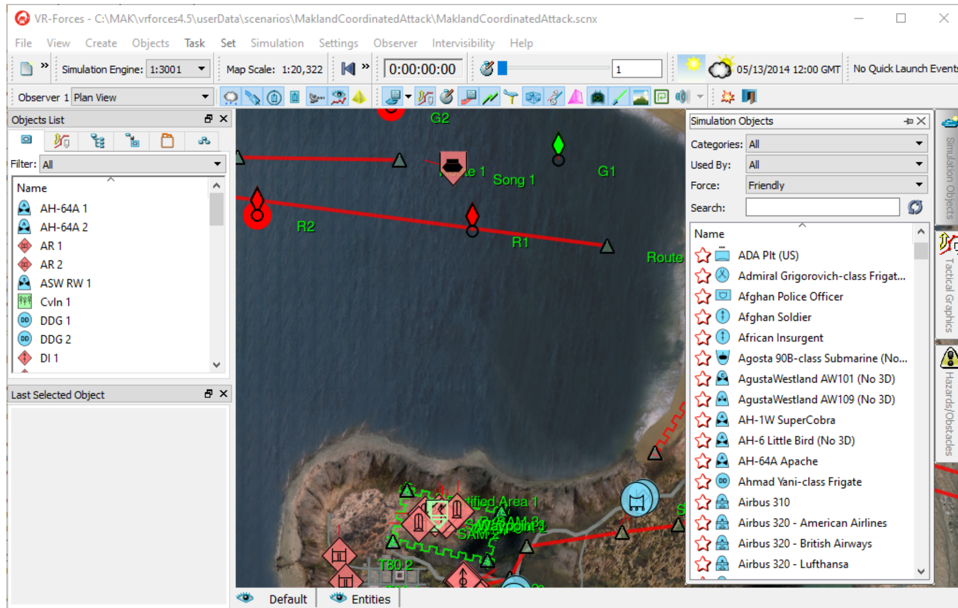


Figure 2.8: VR-Forces GUI with 2D view of a simulation scenario

simulation network every frame. The simulation engine also supports scripted entity behaviour, e.g. in the form of Lua[1] scripts.

VR-Forces GUI

VR-Forces GUI is an application that connects to the VR-Forces engine to display the simulation to the user. The observer can choose to render the simulation in several different ways, including different 2D and 3D visualisations. This application is optional and is not needed to run simulations, but is a useful tool for the user to view the simulation and ensure that the agents perform the tasks correctly. The VR-Forces GUI can also be used to create scenarios. That includes choosing a terrain and placing entities and objects – such as soldiers, tanks, planes, waypoints and roadblocks – and modelling behaviour by creating behaviour plans, movement routes, etc. Figure 2.8 shows the GUI when used to display a simulated scenario in 2D.

2.5.2 VR-Engage

VR-Engage[2] allows a user to take control of a simulated agent in first-person mode. It connects to a simulation engine and can either create a new entity to control or be

[1] `https://www.lua.org/`
[2] `https://www.mak.com/products/simulate/vr-engage`

used to control entity that already exist in a scenario. VR-Forces Engage can be used with keyboard and mouse, joystick or other peripherals and must be connected to a simulation engine, i.e. the VR-Forces Simulation Engine. The visualisation in VR-Engage is similar to the 3D view in the VR-Forces GUI.

Chapter 3

State of the Art

In order to evaluate the hypothesis and contributions, the state of the art of observational learning with toy-problems, behaviour evaluation in complex simulations, observational learning in complex simulations, evolving behaviour trees, and bloat-control with genetic programming and behaviour trees has been reviewed. Finally, a summary of the most important takeaways and their relations to the RQs and contributions is given.

3.1 Observational Learning with Toy-Problems

Here, we look at experiences and results from working with observational learning with toy-problems. The motivation behind this thesis is related to automating modelling of human behaviour, and the work on using observational learning on human behaviour is therefore especially interesting. The reviewed work includes the use of GP, as well as other learning techniques.

Fernlund et al. [14] show that observational learning with the use of GP can be successfully applied to learn personal behaviour traits from observed human behaviour. They attempt to learn human driving behaviour, and use driving simulations for both recording example behaviour and for evaluating generated behaviour models. By using a combination of GP and case based reasoning (CBR), the researchers were able to generate behaviour models that were found to perform at least as good as behaviour models that were manually developed by proficient engineers.

Luotsinen et al. [23] were able to use learning from observation to train an ANN and decision tree models to perform a specific task of passing a hockey puck as part of a toy-problem. The trained agents were able to learn the desired behaviour in three different exercises with small observation data sets. The researchers discuss how the training method would work in more realistic environments, and go on to list a set of problems that must be addressed in order to use the approach in real-world applications. Some of these are issues with data, such as insufficient,

incomplete or noisy data, that arise from using a real-world application.

Stein et al. [24] were able to imitate human behaviour on three separate problems, each with a different goal and domain. The researchers used observational learning to learn from recorded human behaviour, experiential learning to learn optimal behaviour, and a hybrid approach where behaviour learned through observational learning was optimised with experiential learning. For these experiments, they created an algorithm called PIGEON-Alternate which is a combination of NeuroEvolution of Augmenting Topologies [25] and Particle Swarm Optimization [26]. With observational learning, Stein et al. were able to train behaviour models that successfully imitate the observed human behaviour. With experiential learning, they were able to train more optimal behaviour, however, lacking of human behavioural traits. With a combination of using the human-like behaviour trained through observational learning and then optimising it with experiential learning, they found that the behaviour model became more proficient at its designated tasks, while it had retained most of its human-like qualities.

3.2 Behaviour Evaluation in Complex Simulations

Using complex simulation systems for behaviour evaluation can introduce new challenges. In this section, we look at the experiences and discussions related to these potential issues in the literature.

Luotsinen et al. [10] discuss potential issues of using complex simulation systems for behaviour evaluation. In their experiments, they use GP for experiential learning in a toy-problem, and go on to discuss three important issues with running similar experiments in a more complex simulation system, specifically VBS3.

The first issue Luotsinen et al. discuss is that when using simulation systems or conducting exercises that are restricted to running in real-time, the evaluation of generated behaviour is done in real-time as well. This limits the possible number of evaluations that can be done during training. The simulation system we have used for our experiments – VR-Forces from MÄK – is not restricted to simulating in real-time. However, due to the heavy computational load of calculating new world states in realistic simulated environments, the issue with long simulation times and limited possible evaluations is still relevant. The researchers mention that a common solution to this problem is scaling the algorithm, either vertically (high performance computing) or horizontally (clusters). The second issue is that defining behaviour in a single fitness function can be a difficult process. The third and final issue discussed by the authors is that the level of complexity in VBS3 will increase the search space for the GP algorithm.

Lim et al. [5] also discuss the limitations related to long simulation times and limited number of evaluations. They used GP to evolve a behaviour tree which was able to win the majority of games played versus the default AI-bot on the DEFCON[1] multiplayer real-time strategy game. The behaviour trees were generated

[1] Official DEFCON website: `http://www.introversion.co.uk/defcon/`

through experiential learning. The authors emphasise the time required to process the simulations as an important limitation. In their experiment, they trained a population of 100 individuals over 100 generations, a process they estimated would take approximately 41 days to complete using a single simulation engine. However, they were able to reduce the processing time down to approximately two days per experiment by distributing the load over 20 computers. The issue of computationally heavy behaviour evaluation is especially relevant when using GP that require frequent evaluations of a large population of candidate solutions.

3.3 Observational Learning in Complex Simulations

In this section, we look at the results and experiences of other authors from using observational learning in complex simulations.

Using machine learning to generate behaviour models for CGFs has been discussed in the NATO Research Task Group IST-121 RTG-060 "Machine Learning Techniques for Autonomous Computer Generated Entities" [27]. The paper refers to different case studies performed by the participating nations. Worth mentioning is the work done by Totalförsvarets forskningsinstitut (FOI), who used machine learning to create autonomous agents that learn a tactical movement called bounding overwatch, for dismounted infantry [17]. The researchers were able to learn bounding overwatch behaviour by using observational learning in realistic simulations. Although, for our experiments we will use a different simulation system than the one used in this approach, it is still interesting that the researchers were able to create bounding overwatch behaviour by using a realistic, complex simulation system.

Kamrani et al. [17] trained four behaviour models for each agent in the simulation. The four models determine the actions of the agent – whether it should it move or not, what stance it should be in, if the the agent should turn to correct the direction of its weapon and what waypoint it should move to. They used decision trees for the first, second and fourth model, and an ANN for the third model. All four models were trained separately. The models using decision trees were trained by using observed example behaviour. The four models were then manually combined to create desired behaviour, such as bounding overwatch. The experiments were all performed on a simulation system called VBS3[2], a game based military simulation system. An important note here is that the generated behaviour model required a route of waypoints specifying where the agents are to stop and change roles. In other words, it required the scenario to be specifically designed to use this behaviour model. In contrast to the approach described in this paper, our approach will not involve any manual steps in creating the behaviour tree.

Floyd and Esfandiari [28] propose a framework for modelling behaviour for CGFs in realistic, complex environments. The framework uses CBR that learns from observation, and is designed to be domain agnostic, although still allowing a

[2]https://bisimulations.com/virtual-battlespace-3

designer to make the framework domain biased in order to optimise performance. The authors emphasise benefits of complete separation between the central reasoning system and any domain-specific information – allowing agents to learn and act in a variety of domains. They also emphasise the importance of good quality observations, including coverage of problem-space and data noise, as it greatly affects the learning performance of the agent.

Zhang et al. [29] propose another CBR-based learning framework, which can be used to train CGF behaviour in military simulation based training and analysing applications. The proposed framework learns both from observational and experiential learning and uses behaviour trees to model the CGF behaviour. Instead of evolving a behaviour tree from a randomly generated starting point, they suggest generating an initial tree that is maximally specific to the observed behaviour. Then, they suggest pruning the tree by merging common subsequences in the initial tree. Finally, they suggest storing the selector nodes in the tree as separate case catalogues in a knowledge base, containing the actions of the respective selector node as cases. The authors also emphasise the potential computational expensiveness of using GAs to evolve behaviour trees in an exploratory process.

3.4 Evolving Behaviour Trees

Here, we look at how other authors have approached the process of evolving behaviour trees, including choice and evaluation of genetic operators. The discussed articles cover the use of experiential learning to optimise agent behaviour, but their approaches and experiences with evolving behaviour trees are relevant for use with observational learning as well.

Colledanchise et al. [4] used experiential learning to generate behaviour trees that control Mario in the open source benchmark Mario AI [30]. The researchers used a combination of a greedy heuristic algorithm and GP to evolve the behaviour trees to be able to obtain as many points in the Mario AI benchmark as possible. The researchers describe their results as "encouraging and comparable to the state-of-the-art". For genetic operations, they used a two-point crossover and a single mutation. The crossover is done by swapping a random subtree from one parent with a subtree from the other parent. The mutation replaces a single node in the behaviour tree with a node of the same type (composite or leaf node). They used SA [16, p. 128] for gradually reducing the mutation rate for each generation, focusing on local search as the algorithm converges.

As previously described, Lim et al. [5] used GP to evolve behaviour trees for playing the real-time strategy game, DEFCON. They were able to create behaviour models that won the majority of games played against the default game AI. For evolving the behaviour trees, they used a two-point crossover, where a random subtree from the first parent is replaced by a random subtree from the second parent, and vice versa. They also used two types of mutations – one which adds a new action node to an existing behaviour tree, and one which changes the variables

of an existing action node. They use action nodes with variables that adjust how the represented action is performed, e.g. the coordinates of where a fleet is to be placed. The authors conclude that their approach is feasible for developing AI for commercial games, but emphasise that the mean fitness reaches a plateau, possibly indicating the need for supplementing the evolutionary learning with other learning techniques.

Perez et al. [6] used grammatical evolution to evolve behaviour trees using experiential learning. Grammatical evolution is a specialisation of GP, where a specified context-free grammar is used to map candidate solutions to syntactically correct solutions. Like Colledanchise et al. [4], they also used the Mario AI benchmark in their experiments. They submitted a generated behaviour model to a competition where they placed fourth out of seven contenders, and the authors conclude that GP approaches are serious contenders to other more traditional artificial intelligence (AI) algorithms. For genetic operations, they also used a two-point crossover, where they switch the positions of two blocks of the binary strings. The blocks are marked with grammar symbols, indicating possible crossover points. They also allowed the crossover operation to be done with the same chromosome as both parents – effectively creating an operation which clones the selected parent and switches the positions of two sub-trees.

3.5 Bloat-Control with Genetic Programming and Behaviour Trees

As mentioned in Section 2.3, bloated solution representations is a common issue with GP. There are several proposed techniques for controlling or reducing the effects of bloating. In this section, we look at some of these techniques, and how they affect the training performance with GP.

Poli [31] present the *Tarpeian* method for controlling bloat in GP where they randomly penalise a selection of the offspring that are larger than a specified size limit, specifically setting their fitness to zero. This creates fitness "holes" that discourages search in the vicinity of larger solutions. The method is proven to be effective at controlling bloat, and the authors emphasise the benefit of not needing to evaluate the solutions whose fitness is automatically set to zero.

Another approach, as suggested by Bleuler et al. [32], is the use of solution size as a minimisation objective in multi-objective GP techniques. A problem with using this method is that the non-dominated individuals of the population can tend to cluster at the edges of the pareto-front. A way of handling this problem is by introducing diversity-control into the selection process [33], discouraging clustering. SPEA2 [34] is a multi-objective evolutionary algorithm (MOEA) that does exactly this. Bleuler et al. test the approach with SPEA2 and conclude that it is a promising approach for reducing bloat in GP that will most likely work well with other MOEAs, a conclusion supported by [35].

Another approach is to use a fixed limit for the depth or size of the solutions.

Koza [19] popularised the technique of limiting depth size, a technique commonly used for bloat control in GP. Crane and McPhee [36] investigated the effects of using both depth and size limits. From the results of their experiments, they observed that using a depth limit introduced more bias toward smaller solutions in the specified search space compared to using a size limit. They also observed that there were more inconsistencies in how much of the search space that was covered, when using depth limits. The authors emphasise that these observations will probably, but not necessarily, generalise to other problems than those used in the experiments.

Luke and Panait [37] compared some of the most popular methods for controlling bloat in GP, including the Tarpeian method and the multi-objective method. They concluded that when combined with the use of limited tree depth, both techniques are effective in reducing bloat. They discovered that all the evaluated methods performed the same or better when augmented with depth limitations.

Colledanchise et al. [4] uses a retroactive approach to bloat-control in observational learning with GP. They do not use any bloat-controlling techniques when evolving behaviour trees, however, once a behaviour tree satisfies the specified goal, they search for and remove ineffective subtrees. This is done by selecting subtrees in breadth-first order, simulating the tree without the selected subtree, and then deciding whether the selected subtree should be kept based on the fitness score. If the fitness score is better or the same as the original behaviour tree, then the subtree is permanently removed, and a new subtree then considered for removal. The researchers mention that bloat-control is most often performed by the genetic operations or selection mechanism in GP, such as the previously discussed methods.

3.6 Summary

In this chapter we have looked at the state of the art of observational learning with toy-problems, behaviour evaluation in complex simulations, observational learning in complex simulations, evolving behaviour trees, and bloat-control with genetic programming and behaviour trees. Observational learning has been successfully used to learn personal behaviour traits from the observed actors, and has been shown to be at least as good as proficient engineers at modelling human behaviour [13]. Several authors have discussed the potential challenges of using complex simulation systems for evaluation of behaviour models, emphasising the issues with heavy computational load and long simulation times when used with GP. We found little research on the use of GP with observational learning. Kamrani et al. [17] have shown that observational learning in complex simulations is possible, but their work does not include GP or behaviour trees. While GP has been used to evolve behaviour trees through experiential learning in complex simulations [5], we have not found any research on using GP to evolve behaviour trees through observational learning in complex simulations. These findings relate to C1 and

help answer RQ1.

In current literature, there are proposed frameworks for using observational learning in complex environments. However, both the discussed frameworks are CBR-based, and are not suitable for our research. Kamrani et al. [17] developed a system for running observational learning in complex environments, but their solution is not designed to use HLA. The design of such a system that uses HLA is related to RQ2 and C3.

For evolving behaviour trees, the reviewed research uses two-point crossovers with one or two mutations. The mutations include adding a single node, replacing a node with a node of the same type, and tuning the variables of a single node. Lim et al. [5] use variables in their action nodes to tune the specific action. This is an interesting approach, which can be useful for tuning condition nodes as well. Research on how to evolve behaviour trees is relevant for C1 and C2, which help answer RQ1.

Bloat is a common issue with GP, and there are several well-documented approaches to reducing bloat during training. We have looked at three techniques, the Tarpeian method, minimisation of solution size as an objective in a MOEA, and the Koza-method of setting a hard depth limit for the generated solution trees. Luke and Panait [37] conclude that both the Tarpeian method and the multi-objective method both work well when combined with the depth limit. Crane and McPhee [36] argue that limiting the total size of the tree instead of the depth is a better alternative as it results in better coverage of the search space. Knowing how to manage bloat control with GP is related to C1, that helps answer RQ1.

Chapter 4

Methods

This chapter covers the methods used in order to answer the hypothesis and research questions of this thesis. During the literature review, we found no existing system that would allow us to run the experiments required to answer RQ1. We therefore created RQ2, related to how a system should be designed to be able to run different experiments with observational learning in complex environments by connecting to a simulation engine over HLA. Included in RQ2 was a set of requirements for the system, shown in table 1.1.

Section 4.1 provides an overview of the developed system. Section 4.2 describes the used simulation environment and how our system communicates with it. Section 4.3 describes how the system handles data and feature extraction. Section 4.4 describes the used behaviour tree representation. Section 4.5 covers the methods used for evolving behaviour trees. And finally, a summary which relates the covered methods to the RQs and contributions is given in Section 4.6. For a detailed description of the system implementation, see Chapter 5.

4.1 System Overview

We developed a system for using GP to generate behaviour trees with observational learning in complex simulations within a HLA federation network. The system is used to record example behaviour and to train behaviour trees that imitate the recorded examples. The system handles data extraction, data processing, behaviour tree generation and behaviour tree visualisation, but the actual simulation of the behaviour is done in an external simulation system. Figure 4.1 shows an overview of the complete process, including recording example behaviour and training behaviour trees.

All communication between the system and the external simulation system is done exclusively over HLA. This includes publishing of instructions to the simulation engine and simulated entities, and subscribing to simulation data updates. By restricting all communication to go over HLA, the system has the ability to connect

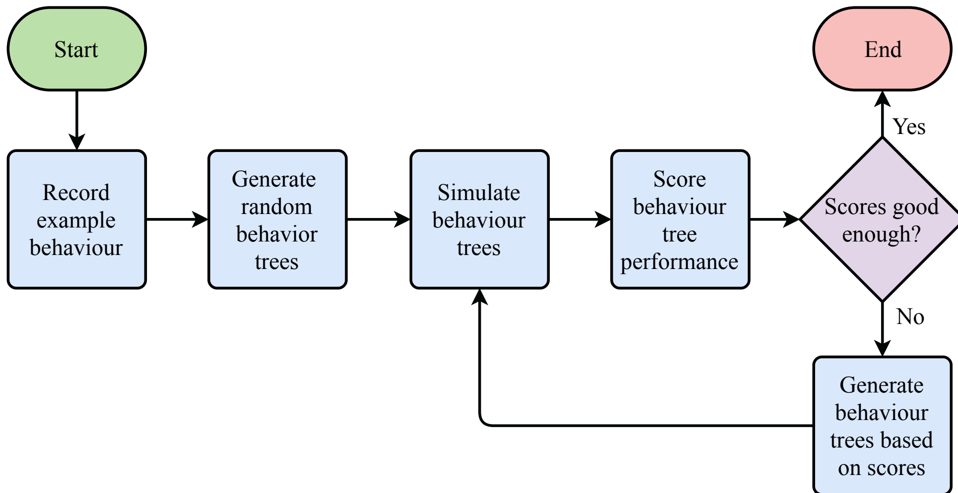


Figure 4.1: Process flow of the complete process of recording example behaviour and generating behaviour models through observational learning

to a variety of simulation systems.

CGFs include a large variety of different types of simulated entities, and as specified in the RQ2 requirements, the system had to be compatible with training behaviour for different types of entities. It is reasonable to assume that training different entities with different behaviour would require different data from the simulation, and that entities have different types of actions that they typically perform. This led to maybe the most important design decision for the system: that it should be compatible with any type of simulated entity as long as it is specified in the HLA federation FOM used for the simulation network. As long as the simulation data related to the entity is published on the HLA federation network, the system is able to subscribe, process and store the data based on the requirements of the specific experiment. For controlling entities in the simulation, the system uses a FOM extension module called Low Level Behaviour Markup Language (LLBML), developed by Netherlands Organisation for Applied Scientific Research (TNO) and FFI [38, 39]. LLBML allows sending of low-level instructions to the simulated entities. To enable full modularity, the core system provides a framework for representing entities, simulation data and entity instructions, and it is required that the user implements any specialised functionality by using this framework.

The system can be used for both recording example behaviour and for training behaviour models based on previously recorded examples. The data that should be used depends on the type of behaviour that is to be recorded or trained, and is specified by the user when setting up an experiment. Following are descriptions of how the system is used for recording and training.

4.1.1 Recording Data from Example Behaviour

When recording data from example behaviour, the system connects to the HLA federation, and subscribes to data updates on specific types of simulated entities. The user specifies which entities' updates that should be recorded as example behaviour. These observed entities may e.g. be controlled by humans with VR-Engage. As the simulation is running, the system works only as a passive data collector, saving the received data updates to comma-separated values (CSV) files on a specified simulation time interval. When the data from the desired behaviour has been successfully stored in the CSV file, the file is then manually marked with the scenario it was recorded in. The file is now ready to be used as example behaviour.

4.1.2 Training Behaviour Trees

For training the behaviour trees, the system uses a GA implementation to generate a random initial set of behaviour trees, and then evolve these trees to better imitate the desired behaviour. One or more examples of desired behaviour is loaded from CSV files that are generated as described in Subsection 4.1.1. Specifics on the used GA implementation and methods for evolving the behaviour trees are provided in Section 4.5. In order to evaluate the generated behaviour trees, the system has to simulate and record their performance in the same environment (scenario) as the example behaviour. The system first instructs the simulation system to load and run the scenario specified in the example file. It then controls the specified simulated entity with the generated behaviour tree that is being evaluated, while recording the entity data updates from the simulation system. Finally, the system evaluates how much the behaviour represented in the generated behaviour tree resembles the desired example behaviour. This evaluation score is then used to guide the generation of new behaviour trees.

4.2 Simulation Environment

For our experiments we used a real, military simulation system called VR-Forces from MÄK. The virtual terrain, physical simulation of entities, etc. are simulated in this system. Figure 2.7 shows an example of a virtual terrain in VR-Forces. Our system communicates with VR-Forces using HLA – a standard for distributed simulation that is commonly used in military simulation systems [40]. Other CGF systems that support HLA could be used in place of VR-Forces. An overview of the HLA federation, as visualised in the MÄK RTI Assistant, is shown in Figure 4.2. Our system is named no.ffi.msc.gbhem.

Systems that communicate over HLA must agree on data and data formats to exchange. This is formalised in a federation object model (FOM), and we have used the Real-time Platform-level Reference Federation Object Model (RPR FOM), which is a standard FOM that many military simulation systems support [41, 42].

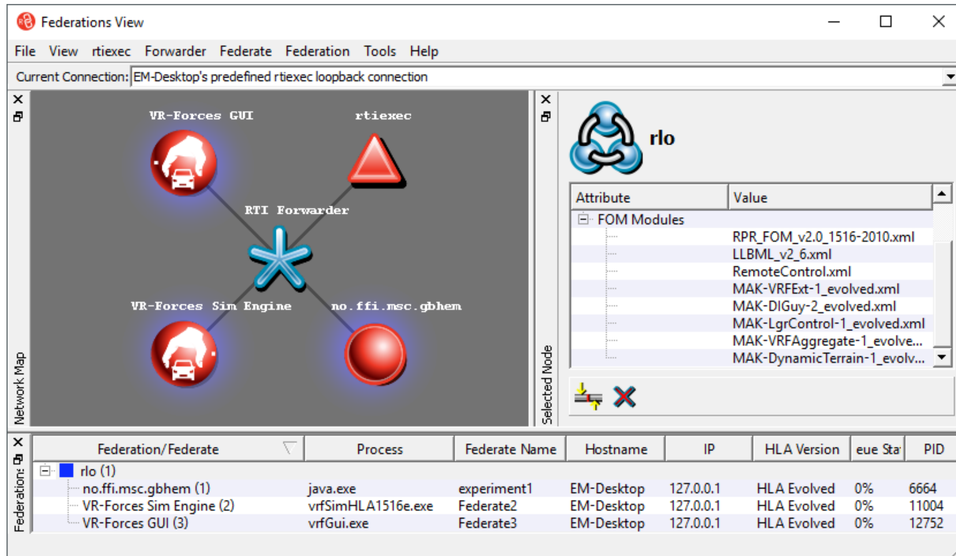


Figure 4.2: HLA federation overview shown in MÄK RTI Assistant

However, this FOM does not include commands or the perceived truth of the CGF entities. For this we use the LLBML module, which is made as an extension to the RPR FOM. Bruvoll et al. describe using a multiagent system to control a CGF system in a similar manner [1, 43].

4.3 Data and Feature Extraction

Each experiment may require different data from the simulation. Data is used by the action and condition nodes in the behaviour tree – nodes that are specific to the entities controlled in the experiment, and for evaluating the behaviour trees total performance – done by experiment-specific evaluation methods. Both the behaviour tree nodes and the evaluation methods may need specific features extracted from the simulation data. The simulation system publishes data updates to the HLA federation when calculating new world states. Our system subscribes to data updates on specific types of simulated entities that are formalised in the HLA federation FOM. To select what data that should be used and how it should be processed, the system uses experiment-specific system entity for handling simulation data, called *DataRows*. The DataRows are used to store raw data, and for extracting features from the raw data and storing those. This data is then available for the behaviour tree nodes and for behaviour tree evaluation.

4.4 Behaviour Tree Representation

For representing behaviour trees used during simulation, we use gdxAI[1], a Java library for AI techniques used in computer games that is actively maintained and has an active community. Behaviour trees represented with the gdxAI behaviour tree framework has limited structural modularity once created. In order to allow easier manipulation of the behaviour trees during evolution, we created a simple hierarchical representation that works as a blueprint for how the executable trees should be built once needed for simulation. The blueprint representations are suitable for performing genetic operations. Functionality for sending instructions to the simulated entities is implemented in the gdxAI-based leaf nodes.

We use selector and sequence composite nodes, and experiment-specific action and condition nodes for our experiments. In our implementation, both condition and action nodes can be augmented with internal variables that change how they function. E.g., for a condition node which checks if two agents are within x meters of each other, the variable x may be changed to different values. Lim et al. [5] used variable action nodes in their work. In the graphical representations of the behaviour trees, the variables of a variable node are shown in square brackets after the name of the node.

4.5 Evolving Behaviour Trees

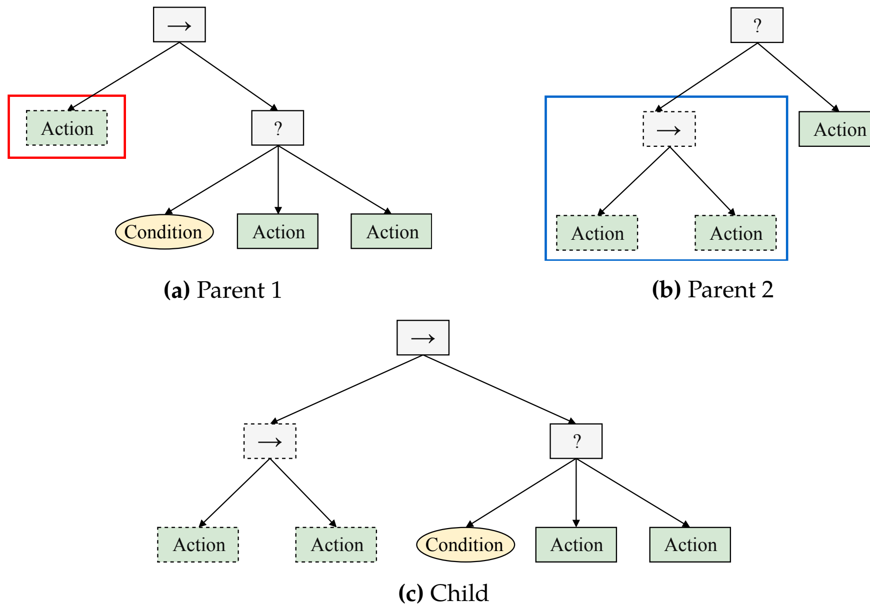
We use GP to evolve behaviour trees. The developed system supports the use of different algorithms. For our experiments we implemented and used a GA called Non Sorting Genetic Algorithm II (NSGA-II). Following are descriptions of the algorithm and the genetic operators used for evolving behaviour trees.

4.5.1 NSGA-II

NSGA-II is a MOGA which uses a non-domination ranking system and implements diversity control using of crowing distance [44]. For each epoch, the algorithm generates an offspring population based on the existing population, then combines the offspring population with the existing population, rank all candidate solutions in the combined populations, and then selects a new population based on their non-domination ranks and crowding distances.

The NSGA-II implementation has four settings: population size, crossover rate, minimum tree size and maximum tree size. The population size determines the size of the population and offspring populations used. In our implementation, when generating an offspring population, the new behaviour trees are created either through crossover or mutation. The crossover rate determines whether each new candidate solution should be created by crossover or by cloning and mutating an existing candidate solution. That means that with a crossover rate of 0.6, there

[1]https://github.com/libgdx/gdx-ai



(a) Parent 1

(b) Parent 2

(c) Child

Figure 4.3: Crossover

is a 60% chance that the offspring candidate solution is created through crossover, and 40% chance that it is created through mutation. Selection of parents and cloned candidate solutions is done with a binary tournament comparing non-domination ranks and crowding distance, in that order. The minimum and maximum tree size settings restrict the genetic operators from creating trees that are outside the specified limits. This is further discussed in Subsection 4.5.3.

4.5.2 Genetic Operators

Following is a description of the crossover and mutation operators used in the system.

Crossover

The crossover operator takes two behaviour trees as parents. First it chooses a random subtree in each of the parents, as illustrated in Figures 4.3a and 4.3b. Then, a clone of parent 1 is created and the random subtree in parent 2 is inserted at the position of the random subtree in parent 1. Figure 4.3c displays the final tree after crossover. This is the same crossover operator as is used in [4]. When selecting the parents, the crossover operator is allowed to select the same behaviour tree twice. This will result in a clone of the parent, where two randomly selected subtrees switches position. This is a technique also used by Perez et al. [6].

Mutation

The mutation operator uses seven different mutations with varying level of impact on the behaviour trees. In our NSGA-II implementation, each offspring is either created by the crossover operator, or by cloning and mutating a selected parent from the existing population. We created a probabilistic method for choosing between multiple mutations.

Each mutation is given a starting weight. When the system chooses which mutation to use, each mutation has the probability to be chosen relative to its weight divided by the total weight of all the mutations. This means that a mutation with a weight of two is twice as likely to be chosen as a mutation with a weight of one. The equation for calculating the mutation probability is shown in Equation (4.1), where m is the single mutation and M is the set of all mutations.

$$prob(m) = \frac{m_{weight}}{\sum\limits_{x \epsilon M} x_{weight}} \tag{4.1}$$

In addition, the weights of the mutations can also be scaled based on the number of epochs the training algorithm has been running. In order to do this, each mutation is given a factor base in addition to the starting weight. When calculating a mutation's weight given the current epoch, the mutation's starting weight is multiplied with the base factor to the power of the epoch number. This equation is shown in Equation (4.2), where $weight_0$ is the mutation's starting weight.

$$weight_{epoch} = weight_0 \times (factor_base^{epoch}) \tag{4.2}$$

With a factor base of 1.1, the mutation weight would increase by 10% for each epoch, and with a factor base of 0.9, the mutation weight would decrease by 10% for each epoch. With this technique, we are able to use mutations that result in drastic alterations to the behaviour trees more frequently in the early stages of the training, and use the mutations that make smaller changes more as the training goes on. This creates a SA effect, which helps prevent the algorithm from getting stuck in local minima while still allowing localised search as the algorithm converges [16, p. 128].

Following are descriptions of the seven different mutations that are used by the mutation operator.

Add Random Subtree This mutation generates a random subtree with a specified minimum and maximum number of nodes, and inserts it at a random position in the behaviour tree. Figure 4.4 shows the insertion of a tree with three nodes, marked with a dotted line.

Add Random Leaf Node This mutation is identical to the Add Random Subtree mutation, except that it only adds a leaf node. This mutation is also used in [5].

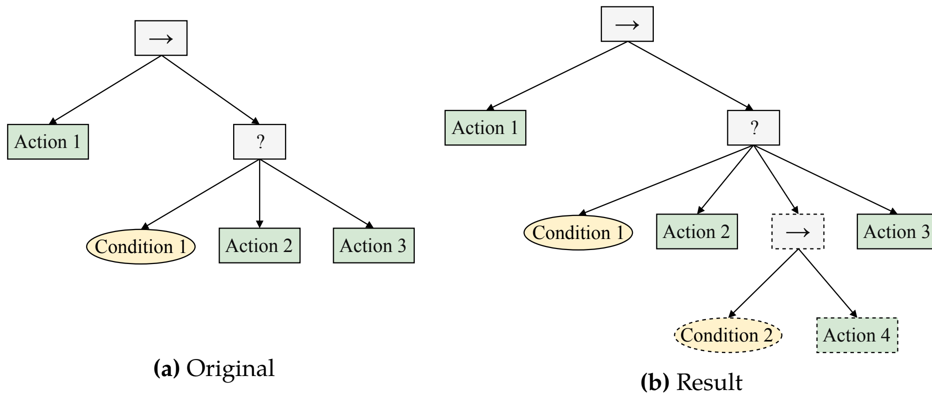


Figure 4.4: Add Random Subtree

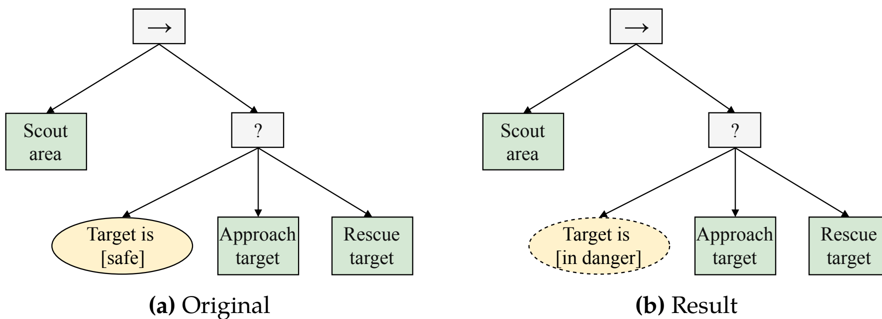
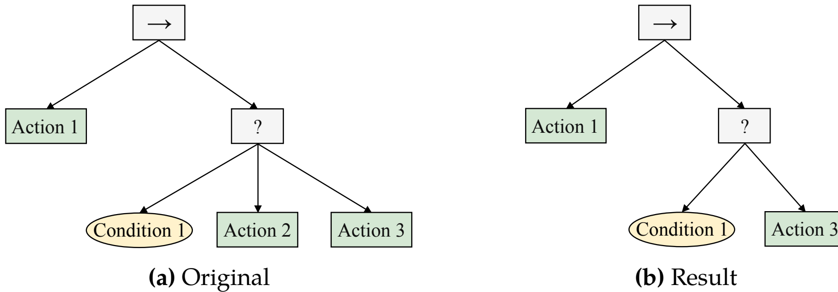


Figure 4.5: Randomise Variables of Random Node

Randomise Variables of Random Node Both action nodes and condition nodes can have variables that affect their functionality. This reduces the total number of nodes a developer has to make and also allows the system to fine-tune the behaviour of the behaviour tree. E.g., for a condition node that checks if the distance between two units is lower than a certain value, the value can be altered during training to check for different distances. This mutation randomises one or multiple variables in an action or condition node. In Figure 4.5 the value of the *"Target is ..."* condition node is changed from *safe* to *in danger*. Lim et al. [5] uses a similar mutation in their research.

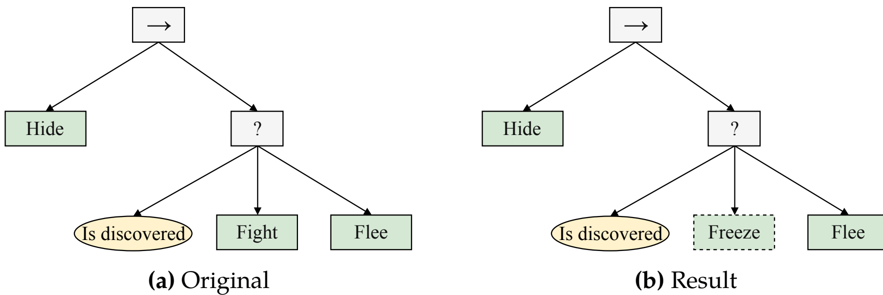
Remove Random Subtree This mutation removes a random subtree from a behaviour tree. Figure 4.6 shows the removal of the *Action 2* node.

Replace Random Node With Node of Same Type This mutation replaces a random node with another random node of the same type. This means that a composite node can be replaced with another composite node (e.g. sequence to selector) or



(a) Original **(b)** Result

Figure 4.6: Remove Random Subtree



(a) Original **(b)** Result

Figure 4.7: Replace Random Node With Node of Same Type

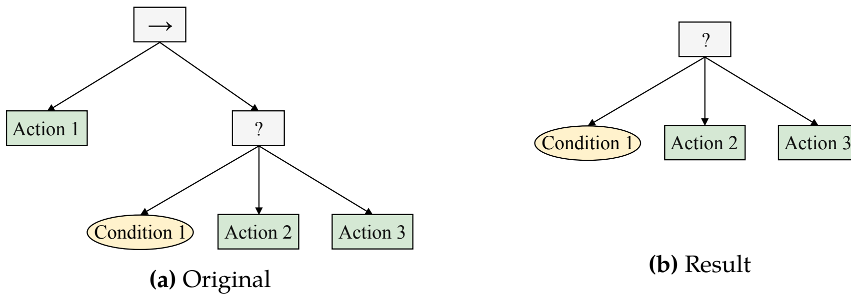
that a leaf node is replaced with another leaf node. Condition and action nodes are not treated differently, and may be replaced by any other leaf node. In Figure 4.7, the *Fight* action node is changed to a *Freeze* action node, marked with a stippled outline. This mutation is also used in [4].

Replace Tree With Subtree This mutation replaces the entire tree with a random subtree of that tree. In Figure 4.8 the entire tree is replaced by the selector subtree.

Switch Positions of Random Sibling Nodes This mutation switches the position of two random sibling nodes, including both leaf and composite nodes. In Figure 4.9 the *Condition 1* and *Action 2* nodes have switched places.

4.5.3 Behaviour Tree Evaluation and Bloat Control

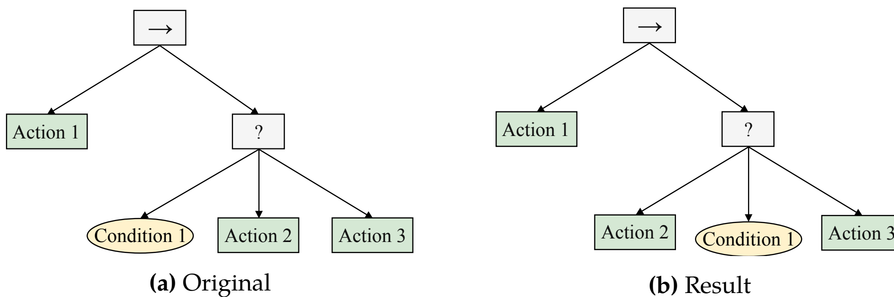
Evaluation metrics for behaviour trees and their performance are experiment-dependent, and it is therefore required that the evaluation is defined per experiment. However, as described in Section 3.5, evolving behaviour trees with GP can often result in bloated trees, with subtrees that does not affect the represented behaviour. To combat this, we applied three countermeasures, where the first is part of the behaviour tree evaluation process.



(a) Original

(b) Result

Figure 4.8: Replace Tree With Subtree



(a) Original

(b) Result

Figure 4.9: Switch Positions of Random Sibling Nodes

First, we added size as a minimisation objective for the NSGA-II, as suggested by Bleuler et al. [32]. This causes the NSGA-II algorithm to consider tree size when selecting solutions for creating offspring solutions and selecting the new population. NSGA-II implements diversity-control through the use of crowding distance, which should help reduce the issues with clustering among the non-dominated solutions related to this method of managing bloat.

Luke and Panait [37] concluded that the use of size as an objective worked well in combination with limiting tree depth. Crane and McPhee [36] argue that limiting the total size of the tree instead of the depth is a better alternative than using tree depth. In our NSGA-II implementation, we combine the use of size as one of the multiple fitness objectives with restricting minimum and maximum size of the generated behaviour trees. The exact sizes can be specified in the algorithm settings.

Finally, we apply a set of methods for removing unused and unnecessary parts of the behaviour trees after each crossover or mutation. This includes the removal of empty composite nodes, removal of unreachable nodes, and combining any nested composite nodes of the same type in order to reduce hierarchical complexity.

4.6 Summary

In this chapter, the methods used to help answer the hypothesis and research questions have been described. The system overview, simulation system communication, data and feature extraction, and behaviour tree representation are related to RQ2 and C3. The methods used for evolving behaviour trees are related to C1, which helps answer RQ1, and the proposed method for performing mutations forms C2.

Chapter 5

Implementation

This chapter contains a detailed description of the system implementation. The level of detail provided here is important for documenting how the system works, for future use at FFI. The system is written in Java, and is published as open source at `https://github.com/eivinmor/msc-gbh-em` under the MIT license[1].

Section 5.1 provides an overview of the console output and training visualisation of the system. Section 5.2 covers the most important components of the system implementation. Section 5.3 explains what is needed to set up an experiment. Section 5.4 describes the processes of initiating the system, running simulations, recording behaviour and training behaviour models. Section 5.5 covers the different settings used in the system. Section 5.6 covers the logging system used in the system. Section 5.7 provides an overview of the libraries that are used in the implementation. And finally, Section 5.8 summarises the chapter and relates it to the RQs and contributions.

5.1 System Interface

The system is started by initiating either the recording or training process. When the system is running, it provides the user with system status updates by logging to the system console, and with training progress visualisations during training. Following are descriptions of the console logging output and the training progress visualisation window.

5.1.1 Console Logging Output

All important events in the system are logged to both files and the system console. This includes events such as sending HLA messages, registering of simulated entities, starting new training epochs, starting and stopping the simulation, and general simulation and training progress, such as the evaluation of candidate solutions. All

[1] `https://opensource.org/licenses/MIT`

log outputs are timestamped and marked with the logging level (DEBUG, INFO, WARN, ERROR). The log outputs are colour-coded and indented to let the user more easily see what the output message is related to. Figure 5.1 shows an example of the console log output during training. For more information on the system logging implementation, see Section 5.6.

5.1.2 Training Progress Window

During training, the system will create a window which displays the evolved behaviour trees and the overall fitness development for the training session. This window is updated for every training epoch. When used with the included NSGA-II implementation, the window contains five different tabs. The first four tabs include graphical representations of the behaviour trees in the previous population, the offspring population, the new population and the non-dominated set of behaviour trees. The graphical representations of the behaviour trees use the same symbols, shapes and colours as explained in Section 2.2, with the exception that composite nodes are coloured blue instead of grey. Figure 5.2 shows window with the "Old population" tab open.

The last tab displays plots of the fitness history so far for the training session, as shown in Figure 5.3. Each fitness history plot contain three different data series, one for the best fitness, one for the average fitness, and one for the worst fitness. The fitness history plots are interactive, allowing the user to change axis ranges, point symbols, line styles, etc.

5.2 System Components

The system is composed of a large number of different entities with different responsibilities and functionality. Figure 5.4 shows a high level overview of the system architecture, with the most important components. The section provides a detailed description of the system components and the relationships between them. The system is divided into two main modules – the core system module and the experiment module.

The core system works as a framework for running different experiments with experiment-dependent types of data, experiment-dependent types of simulated entities with varying possible actions, and experiment-dependent evaluation metrics. It has also been designed to be used with different GAs, and to make the addition of new GAs as easy as possible. The only requirement for new GAs implementations is that they extend the Algorithm abstract class and are placed in the core.training.algorithms package.

The class diagram in Figure 5.5 shows an overview of the classes described in this section. It uses the same colour-coding as the architecture overview in Figure 5.4 and the sequence diagrams in Figures 5.7 to 5.10. An explanation of what the different colours means is given in Table 5.1. The beige packages are part

```
    NSGA2Chromosome@3ba0ae41{fitness={Scenario 0=12657.388015686629, Size=3.0}, behaviourTreeRoot=76fe6c
]
19:02:45 DEBUG NSGA2         @268  - RANKS
ArrayList@22a4e11b[
    5 [NSGA2Chromosome@7a7471ce{fitness={Scenario 0=266.56073470202784, Size=3.0}, behaviourTreeRoot=282
    6 [NSGA2Chromosome@62452cc9{fitness={Scenario 0=301.360024167032, Size=3.0}, behaviourTreeRoot=69418
    4 [NSGA2Chromosome@3ba0ae41{fitness={Scenario 0=12657.388015686629, Size=3.0}, behaviourTreeRoot=76f
    3 [NSGA2Chromosome@6d0be7ab{fitness={Scenario 0=7170.507739080215, Size=4.0}, behaviourTreeRoot=1d4f
    2 [NSGA2Chromosome@35c4e864{fitness={Scenario 0=12097.864806472238, Size=5.0}, behaviourTreeRoot=32a
]
19:02:45 DEBUG NSGA2         @269  - NEW POPULATION
Population@17e9bc9e{size=10}[
    NSGA2Chromosome@710d7aff{fitness={Scenario 0=154.37012217059063, Size=8.0}, behaviourTreeRoot=2d7e11
    NSGA2Chromosome@72725ee1{fitness={Scenario 0=161.22938831850394, Size=11.0}, behaviourTreeRoot=40e60
    NSGA2Chromosome@5434e40c{fitness={Scenario 0=161.83918940038828, Size=6.0}, behaviourTreeRoot=3b48e1
    NSGA2Chromosome@3ef41c66{fitness={Scenario 0=164.31681305079337, Size=5.0}, behaviourTreeRoot=6b7395
    NSGA2Chromosome@64a9d48c{fitness={Scenario 0=166.18800753134698, Size=8.0}, behaviourTreeRoot=365a6a
    NSGA2Chromosome@43a65cd8{fitness={Scenario 0=168.7978520163986, Size=5.0}, behaviourTreeRoot=3f1ef9d
    NSGA2Chromosome@60e9df3c{fitness={Scenario 0=168.79785202100075, Size=4.0}, behaviourTreeRoot=907f2b
    NSGA2Chromosome@3403e2ac{fitness={Scenario 0=181.01072926893946, Size=4.0}, behaviourTreeRoot=65b104
    NSGA2Chromosome@7a7471ce{fitness={Scenario 0=266.56073470202784, Size=3.0}, behaviourTreeRoot=28276e
    NSGA2Chromosome@62452cc9{fitness={Scenario 0=301.360024167032, Size=3.0}, behaviourTreeRoot=6941827a
]
19:02:45 INFO  Trainer       @61   - ================ EPOCH 19 ================
19:02:45 INFO  Trainer       @87   - Simulating population.
19:02:45 INFO  Trainer       @91   - Simulating chromosome 0: NSGA2Chromosome@723877dd{fitness=null, beh
19:02:45 INFO  SimController @181  -  Loading scenario C:\MAK\vrforces4.5\userData\scenarios\it3903\bro
19:02:45 INFO  HlaManager    @253  -    Holding time advancement.
19:02:45 INFO  UnitHandler   @134  -    Resetting unit storage.
19:02:45 INFO  UnitLogger    @45   -    Resetting UnitDataWriters.
19:02:45 INFO  HlaManager    @246  -    Sending CgfControlInteraction with command LoadScenario.
19:02:45 INFO  SimController @142  -  START Waiting for all (2) units to be discovered.
19:02:45 DEBUG HlaManager    @119  -    Removed event received: HlaObjectRemovedEvent[objectInstanceHan
19:02:45 DEBUG HlaManager    @119  -    Removed event received: HlaObjectRemovedEvent[objectInstanceHan
19:02:46 DEBUG HlaManager    @94   -    Remote object discovered: HLAobjectRoot.BaseEntity.PhysicalEnti
19:02:46 DEBUG HlaManager    @94   -    Remote object discovered: HLAobjectRoot.BaseEntity.PhysicalEnti
19:02:46 INFO  SimController @154  -  DONE All units discovered.
19:02:47 INFO  SimController @159  -  START Waiting for all (2) units to be updated with values. Reques
19:02:47 DEBUG HlaManager    @269  -    Requesting flush of RTI message queue.
19:02:47 DEBUG HlaManager    @108  -    Updated event received: HlaObjectUpdatedEvent@62ece180{logicalT
19:02:47 INFO  UnitHandler   @41   -    FollowerUnit F1-W1[c] was scheduled for later initialisation.
19:02:47 DEBUG HlaManager    @108  -    Updated event received: HlaObjectUpdatedEvent@1499e219{logicalT
19:02:47 INFO  UnitHandler   @61   -    Unit added: Experiment1Unit@67344bdd{marking=W1, identifier=W1,
19:02:47 INFO  UnitLogger    @26   -    Unit registered for logging: Experiment1Unit@67344bdd{marking=W
19:02:47 INFO  UnitHandler   @61   -    Unit added: FollowerUnit@53264547{marking=F1-W1[c], identifier=
19:02:47 INFO  UnitLogger    @26   -    Unit registered for logging: FollowerUnit@53264547{marking=F1-W
19:02:47 INFO  UnitHandler   @69   -    Controlled unit added: ControlledUnit@8f4fc10 {unit=FollowerUni
19:02:48 INFO  SimController @164  -  DONE All units updated with values.
19:02:49 INFO  SimController @113  -  Playing scenario for 500 ticks.
19:02:49 INFO  HlaManager    @246  -    Sending CgfControlInteraction with command Play.
19:02:49 INFO  HlaManager    @258  -    Enabling time advancement.
19:02:49 DEBUG SimController @73   -  First tick
19:02:50 DEBUG SimController @67   -  Ticks:  100 | Average time per tick: 16ms
19:02:52 DEBUG SimController @67   -  Ticks:  200 | Average time per tick: 16ms
19:02:54 DEBUG SimController @67   -  Ticks:  300 | Average time per tick: 16ms
19:02:55 DEBUG SimController @67   -  Ticks:  400 | Average time per tick: 15ms
19:02:57 DEBUG SimController @67   -  Ticks:  500 | Average time per tick: 17ms
19:02:57 INFO  SimController @168  -  Pausing scenario.
19:02:57 INFO  HlaManager    @246  -    Sending CgfControlInteraction with command Pause.
19:02:57 INFO  HlaManager    @253  -    Holding time advancement.
19:02:57 INFO  Trainer       @91   - Simulating chromosome 1: NSGA2Chromosome@56cc9f29{fitness=null, beh
19:02:57 INFO  SimController @181  -  Loading scenario C:\MAK\vrforces4.5\userData\scenarios\it3903\bro
19:02:57 INFO  HlaManager    @253  -    Holding time advancement.
```

Figure 5.1: System console logging output during training

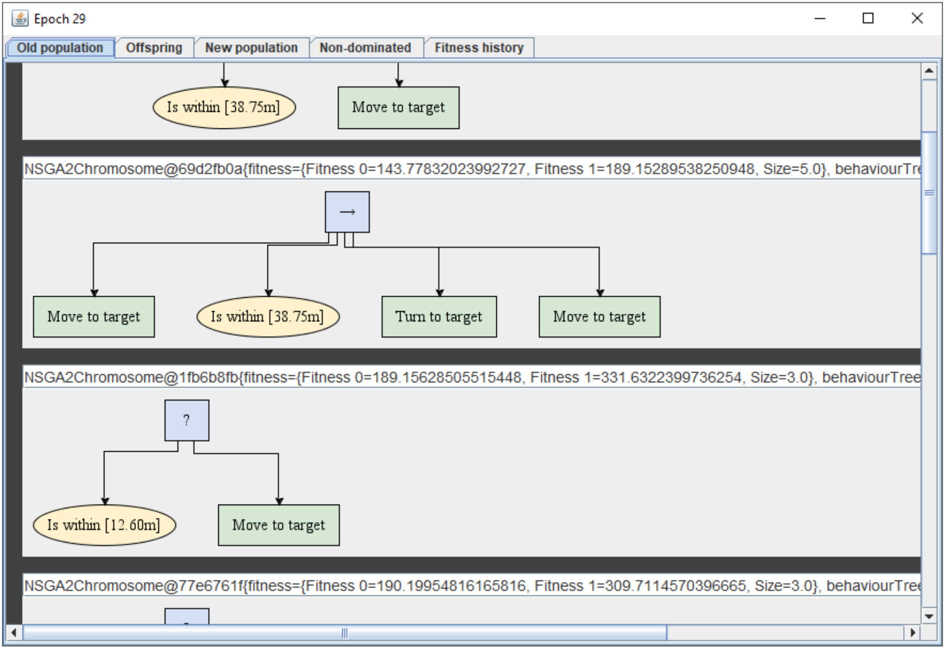


Figure 5.2: Training visualisation window with "Old population" tab open

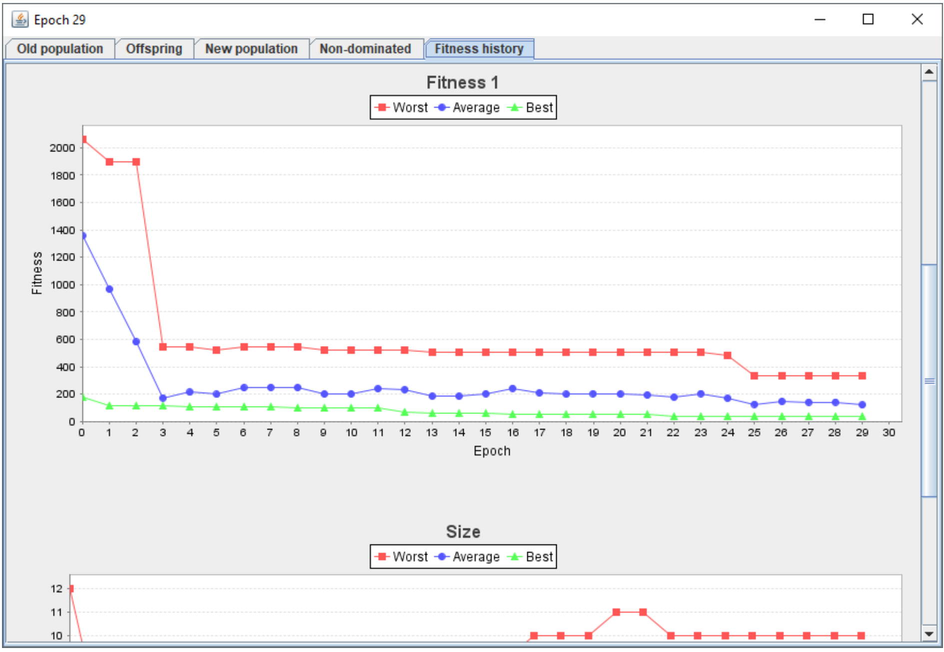


Figure 5.3: Training visualisation window with "Fitness history" tab open

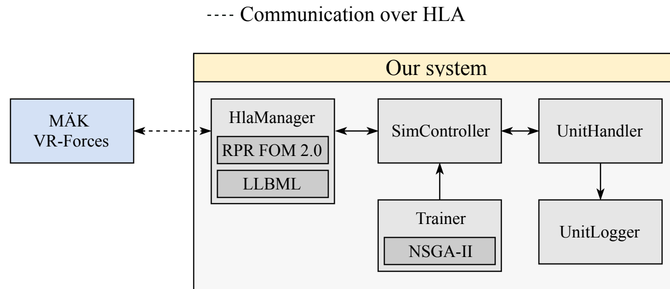


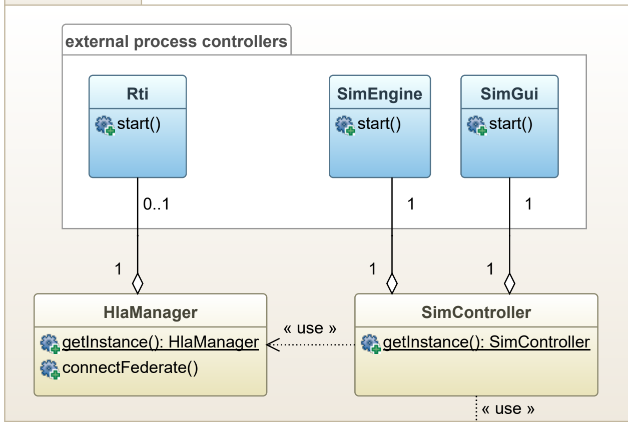
Figure 5.4: Architecture overview of the system connected to MÄK VR-Forces

Table 5.1: Entity colour-coding in system diagrams (Figures 5.4, 5.5 and 5.7 to 5.10)

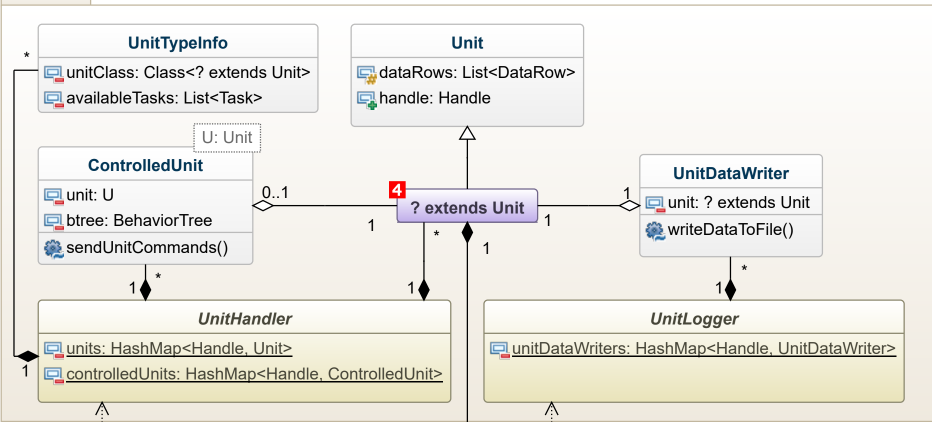
Colour	Relation
Yellow	Most important entities (those included in Figure 5.4)
Blue	Process controllers used to start VR-Forces and RTI
Green	Entities related to behaviour tree representation and operations
Pink	Entities related to algorithm implementations
Purple	Entities that are specified per experiment

of the core system, the pink package contains the included implementation of the NSGA-II algorithm, and the purple package is the experiments package that holds all experiment-specific classes. In the core system packages, there are several pink and purple coloured classes with names that start with "?", that have a red number box in the top left corner. These classes are used to indicate where and how the classes from the experiment module and algorithm implementations are used in the system. In other words, these classes indicate the experiment- and algorithm-specific classes that are required to run the core system. These classes will hereby be referred to as *wildcard entities*. Each wildcard entity in the core system is marked with an ID that is shared with one of the classes in either the experiment package or the NSGA-II implementation package. These IDs indicate exactly which class from the experiment or algorithm implementation package that replaces the wildcard entity. It is important to note that NSGA-II is used as an example of an algorithm implementation, and can be replaced by any other implementation without any modifications to the core system.

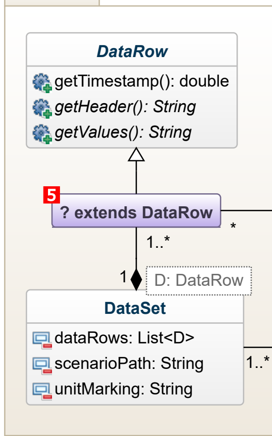
The green package, core.btree.task.template, shows the specialisation hierarchy of tasks used in the internal representation of behaviour trees. Task (green coloured class in the core system) is the highest abstraction level of the possible behaviour tree tasks, and the wildcard entity that extends it can be any of the subclasses shown in the core.btree.task.template package.



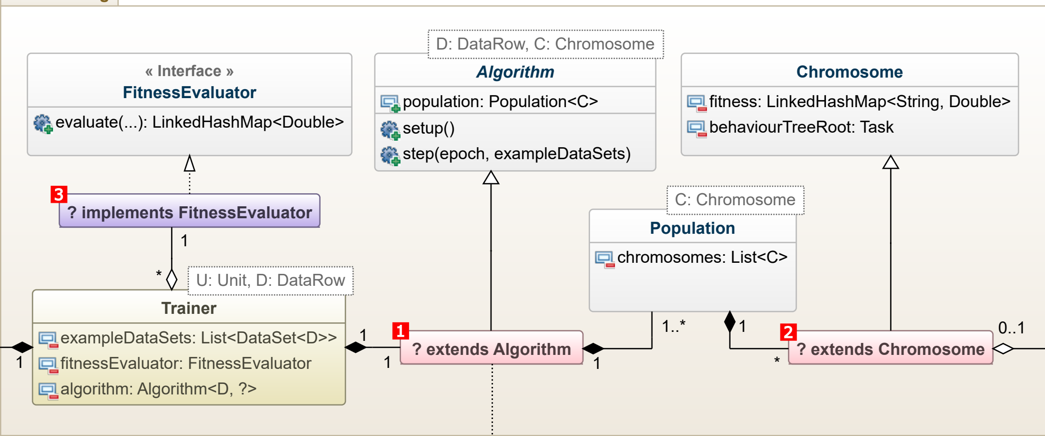
core.simulation
external process controllers
Rti
start()
SimEngine
start()
SimGui
start()
0..1
1
1
1
1
1
HlaManager
getInstance(): HlaManager
connectFederate()
« use »
SimController
getInstance(): SimController
« use »



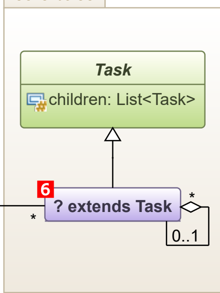
core.unit
UnitTypeInfo
unitClass: Class<? extends Unit>
availableTasks: List<Task>
Unit
dataRows: List<DataRow>
handle: Handle
U: Unit
ControlledUnit
unit: U
btree: BehaviorTree
sendUnitCommands()
4
? extends Unit
UnitDataWriter
unit: ? extends Unit
writeDataToFile()
UnitHandler
units: HashMap<Handle, Unit>
controlledUnits: HashMap<Handle, ControlledUnit>
UnitLogger
unitDataWriters: HashMap<Handle, UnitDataWriter>
*
0..1
1
1
1
*
1
1
*
1
1
1
*
1



core.data
DataRow
getTimestamp(): double
getHeader(): String
getValues(): String
5
? extends DataRow
*
1..*
1
D: DataRow
DataSet
dataRows: List<D>
scenarioPath: String
unitMarking: String
1..*



core.training
« Interface »
FitnessEvaluator
evaluate(...): LinkedHashMap<Double>
D: DataRow, C: Chromosome
Algorithm
population: Population<C>
setup()
step(epoch, exampleDataSets)
Chromosome
fitness: LinkedHashMap<String, Double>
behaviourTreeRoot: Task
3
? implements FitnessEvaluator
1
*
U: Unit, D: DataRow
Trainer
exampleDataSets: List<DataSet<D>>
fitnessEvaluator: FitnessEvaluator
algorithm: Algorithm<D, ?>
C: Chromosome
Population
chromosomes: List<C>
1
1
1
1
1..*
1
1
*
1
? extends Algorithm
2
? extends Chromosome
0..1
« use »



core.btree
Task
children: List<Task>
6
? extends Task
*
*
0..1

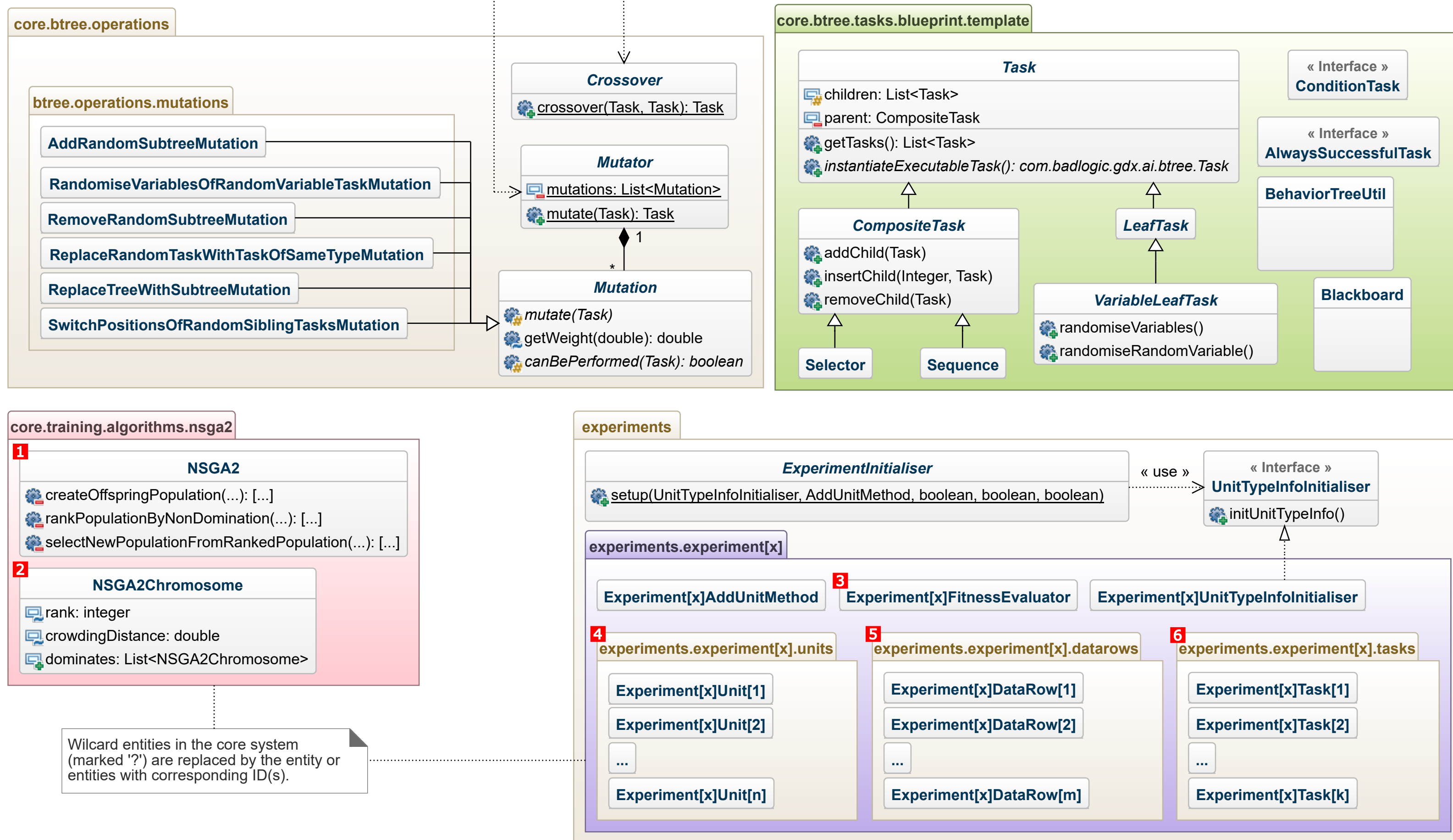


Figure 5.5: Class diagram of the core system and the experiment package. A single-page version for electronic viewing is shown in Figure B.1. The colouring of the system entities follows the system diagram colour-coding, specified in Table 5.1. The core system requires class implementations from both the experiment package and an algorithm implementation to work. These missing classes are shown as purple (experiment) and pink (algorithm) classes with names that start with "?" and have red number boxes in the top left corner. The numbers indicate where each class or set of classes from the experiment and algorithm packages are used in the core system.

5.2.1 Simulation Package

The simulation package contains the HlaManager, the SimController, and three external process controllers. HlaManager manages all incoming and outgoing HLA communication, the SimController is used for controlling the simulation, and the three process controllers are used to initiate the external simulation systems.

HlaManager

HlaManager manages all incoming and outgoing HLA communication. That includes restricting federation time advancement, requesting time advancement, advancing the system federate time, sending HLA messages and forwarding incoming data updates to any listener in the system. All other entities in the system use exposed methods in the HlaManager to communicate with the simulation engine.

SimController

The SimController (simulation controller) controls the simulation. It uses the HlaManager to send instructions to the simulation system, including instructions to play, pause or load a scenario, and instructions for what the simulated entities should do. The SimController is added as a "tick listener" in the HlaManager, meaning that it is notified every time the system federate time advances by a specified internal tick interval. When the SimController is notified that the time has advanced, it instructs the system entities responsible for updating internal data representations (UnitHandler), storing data to files (UnitLogger) and generating entity instructions (UnitHandler) to process the new data received from the simulation system. The SimController also controls how long the simulation should run, depending on a specified number of ticks. The methods for loading, playing, and pausing a scenario is exposed to the rest of the system, and is e.g. used by the Trainer entity.

External Process Controllers

There are three external process controllers, shown in the top left corner in Figure 5.5 under the simulation package. An external process controller is responsible for starting and maintaining external simulation software processes. They listen to the console output of the process, logs this output to our internal logging sysem, and can be set to listen to specific process console outputs. The RTI class can be used to start the MÄK RTI software, the SimEngine can be used to start the MÄK VR-Forces simulation engine, and the SimGUI can be used to start the MÄK VR-Forces GUI. Both the RTI and simulation engine must be running for our system to function as intended. However, they can be started manually instead of using the process controllers. The external process controllers are not able to shutdown or relay console output if the processes were manually started.

The SimEngine and SimGUI process have multiple settings that are tuned in the SimSettings class. These include application ID, HLA version, HLA federation FOM, FOM modules, plugins, etc.

5.2.2 Unit Package

Simulated entities are represented as "units" in our system. The package contains the Unit, ControlledUnit, UnitHandler, UnitLogger, UnitInfo and UnitDataWriter classes. The Unit class is a base class intended to be extended to create more advanced unit types. The ControlledUnit class is used to control a specific unit in the simulation. The UnitHandler is responsible for storing and managing the active units in the system. The UnitLogger is responsible for storing unit data to CSV. The UnitTypeInfo is used to store information about the different types of simulated entities, how they should be represented as units, and what types of nodes that can be used in their behaviour trees.

Unit

The Unit class is the base class that all other units types are built on. It is an abstract class and can therefore not be instantiated. It defines several parts of the unit that must be present: a *marking*, an *identifier*, a *handle* and a list of DataRows. See 5.2.3 for more information on the DataRow base class. The Unit class also contains a set of methods, including an abstract method for updating the unit's DataRows, and getters for the unit variables. Any class that implements the Unit class must implement the updateData method.

A simulated entity in HLA has a marking field which is a text field that the user can fill in when creating the entity. The marking should make it easier for a human to identify different entities in the simulation. In order for our system to determine certain properties of the entity, it requires a special syntax in the marking text. A string representing the identifier of the agent, e.g. "F1", must be present for the system to know which unit type the entity is. The identifiers for different unit types are registered as UnitTypeInfo instances in our system. The identifier is stored in the unit instance and should be unique to each simulated entity in the scenario. This can be followed by a goal separator, the "−" character, which is followed by a string that represents the agents goal. The goal can e.g. be the identifier of another unit that the unit should follow. This is followed by a "[" character to define properties of the agent and closed by a "]" character, e.g. "F1[c]" or "F1-W1[c]". The system uses the "c" character (short for controlled) to define that the agent should be controlled by our system. If no such option is used, then the system will not send instructions to the unit. The user may insert additional text that is ignored by the system by using the "(" character followed some text and closed by a ")" character. An important note is that the entity marking is limited to eleven 8-bit characters in HLA, and it should be investigated if there are other more suitable fields to store this information.

The handle is the simulation unique identifier given to every entity in the simulation. The handle is required when sending commands to a specific unit.

ControlledUnit

The ControlledUnit class holds a Unit instance and a behaviour tree. The Unit instance is the internal representation of a single simulated entity. When the SimController is notified of a new tick in the simulation, it instructs the UnitHandler to execute all ControlledUnits. When a ControlledUnit is executed, it will execute its behaviour tree, which then sends instructions to the simulated entity related to its Unit. The behaviour tree uses data stored in the Unit class instance when executing its nodes.

UnitHandler

The UnitHandler holds all Unit instances in the system. It handles registration and removal of UnitTypeInfo, Units and ControlledUnits. It also updates the internal simulation data representations (DataRows) of the Units and instructs ControlledUnits to execute behaviour trees in order to generate instructions for the simulated entities.

The UnitHandler requires the user to provide a method that handles adding units when the simulation entity is discovered. This is done by passing an implementation of the AddUnitMethod interface to the UnitHandler. The user must supply this method because the unit types are experiment-specific. The method that is supplied should use the UnitHandler's methods for extracting information from an agent's marking text. The UnitHandler exposes methods that return the identifier, options, and methods that can remove the comments and options from the marking string. Once a unit instance has been created, it should be added to the UnitHandler by calling the UnitHandler's putUnit method.

UnitLogger

The UnitLogger is responsible for writing all simulation data stored in the Unit instances to CSV files for use in training evaluation. Each Unit has one or multiple CSV files, depending on how many types of DataRows they contain. The UnitLogger creates a separate folder for each unit, which contains one file for each type of DataRow used by the Unit. The UnitHandler registers every created Unit with the UnitLogger, which then stores the header information and metadata in the CSV file. Metadata may be added before the header line of the CSV file, where each metadata line is prepended with a # symbol followed by the metadata key, the : character and the metadata value. The UnitLogger stores the system start time, the scenario path and the unit marking in the metadata. When the SimController is notified of a simulation tick, it will call the logAllRegisteredUnits method in the UnitLogger, and the UnitLogger will store the updated data stored in the Units to the respective data files.

UnitDataWriter

The UnitDataWriter class is an internal class of the UnitLogger, and is used to write the Unit's data to files. The UnitLogger maintains a list of all the UnitDataWriters, where there is one instance per DataRow per Unit.

UnitTypeInfo

The UnitTypeInfo is used to store information about the different types of simulated entities, how they should be represented as units, and what types of nodes that can be used in their behaviour trees. For each type of simulated entity, an instance of UnitTypeInfo is registered in the UnitHandler. Each instance contains a name, symbol, class type used for internal entity representation (Unit), and leaf tasks and composite tasks that can be used when generating behaviour trees for that type of simulated entity. The UnitHandler exposes two methods for extracting this information – one for extracting UnitTypeInfo instances by symbol, and one for extracting by Unit class.

5.2.3 Data Package

The data from the simulation is used for both generating unit commands during simulation, and stored for use in evaluating behaviour performance after the simulation has been completed. The data package contains entities related to data management. It provides a framework for creating specialised classes for processing and storing simulation data. It contains the DataRow class, an abstract class intended to be extended with specialised data fields, which is used to store data related to a specified timestamp in the simulation. It also contains the DataSet class, used to group multiple DataRows to a data set.

DataRow

The DataRow is the fundamental component for storing and interacting with data in the system. It is an abstract class and can therefore not be instantiated. The only field in the DataRow base class is the simulation timestamp for when the data was extracted. The intent is that the data fields are specified in the implementation extending the DataRow base class. All classes that extend the DataRow class must implement the getDataSetName, getHeader, setValues, and getValuesAsCsvString methods.

The getDataSetName method should return the name of the data set which is used to store the data row to a file with that name. The getHeader is used to get the header line that should be used when writing the data to a CSV file. The getValuesAsCsvString method should return all the values in the DataRow as a comma separated string. The order of the values in the string should match the order given by the getHeader method. The setValues method takes a list of strings as argument and is intended to be used to create a DataRow from a line in a CSV file. Each string in the list should be converted to the appropriate data format

before being stored. The order of the values in the argument should also match the order given by the getHeader method.

DataSet

The DataSet is used to group multiple DataRows to create a data set. A DataSet will only contain one type of DataRow. When constructed, the DataSet takes has two parameters. The first parameter is a class that implements the type of DataRow that the respective DataSet should contain. The second parameter is a path to a CSV file that the data should be loaded from. The constructor reads the scenario path and unit marking from the metadata of the file, but ignores the system time and header line. For each of the remaining lines it will create a DataRow instance, of the specified type, and use the setValue method to add the data values to the DataRow instance.

The DataSet exposes methods to retrieve the scenario path, all the DataRows, the unit marking, the DataSet name and the number of ticks. The number of ticks corresponds to the number of DataRows in the DataSet.

5.2.4 Training Package

The Training package contains entities that are used for training the behaviour trees. This includes the Trainer which controls the training process, and a set of classes that provide a framework for implementing GAs (Algorithm, Population, Chromosome) and experiment-specific evaluation methods (FitnessEvaluator).

Trainer

The Trainer class controls the training process. To make the Trainer experiment-independent, it requires a lot of the experiment specific implementations to run. When creating a Trainer instance, the constructor requires a set of parameters. It requires the Unit class that the behaviour trees should be trained for, the DataRow to use for evaluating the behaviour trees, an implementation of the FitnessEvaluator interface, an algorithm implementation to use when training, and a list paths to the example data files. To start the training, the user must call the train method with a specified number of epochs to run as argument. The Trainer will then, for each epoch, call the step method on the algorithm.

The Trainer exposes a set of methods that are used by the algorithm implementation. It provides a method that easily allows for simulation of the candidate solutions (Chromosome). The method takes a Population and a list of example DataSets as arguments. The example DataSets are used to retrieve the scenario to simulate and the duration to run the simulation. The method then simulates each of the Chromosomes in the Population before returning. The method locks the training thread while the simulation is running. This makes development of the algorithms easier but forces the algorithm to wait until all Chromosomes have been simulated until evaluating them. With the current use of the system, this

restriction has no significant effect on the training time as the time spent evaluating Chromosomes is insignificant compared to the time spent simulating.

The Trainer also exposes a method called setFitness. This method takes a Population and the current epoch number as inputs. It loads the DataSets generated from simulating the Chromosomes and passes the Chromosome, the DataSet from simulating the Chromosome and the example DataSet to the FitnessEvaluator implementation. The FitnessEvaluator implementation returns a set of fitness values which the Trainer sets as the Chromosomes fitness.

The Trainer tracks the fitness history of the training session. The Algorithm implementation can call the updateFitnessHistory method with the new Population of evaluated Chromosomes. When this method is called, the Trainer stores the best, average and worst scores out of all the Chromosomes for each fitness value. This method is typically called each training epoch. The stored fitness history is used to present the fitness development as graphical plots to the user.

Algorithm

The Algorithm class is essential in the training process. It is an abstract class, providing a skeleton for implementing GA algorithms for use during training. The Trainer requires an implementation of the Algorithm class to work. During training, the Algorithm implementation dictates how to generate a new Populations, what Chromosomes to simulate, and which Chromosomes that should be selected for the new Population.

The Algorithm base class requires the implementations to have a setup method, and a step method. The setup method should contain any actions that are required to prepare the algorithm for running the training, such as generating a starting Population. The step method is called for each epoch of the training process, and should contain all necessary actions for the algorithm to generate new Chromosomes, evaluate them and select a new Population.

Population

The Population class is used to hold and manipulate a list of Chromosomes. The Population has a method called selectionTournament that selects a Chromosome based on the number of contenders and a specified Comparator which is defined in the Chromosome implementation. Typically, different algorithms would use different Comparators. The Population class also provides a utility method called containsChromosomeWithEqualTree which takes a behaviour tree as argument and checks if a Chromosome containing an equivalent behaviour tree already exists in the Population. This is used to only keep unique individuals in the Population.

Chromosome

The Chromosome class holds a behaviour tree and fitness values, and is the representation used for candidate solutions of the GA. It works as a baseline implemen-

tation for a Chromosome, and can be extended and replaced with an algorithm-specific implementation that require more than the base functionality provided in this class. In the baseline implementation, the fitness values are stored in a map that uses the fitness name as the key and the fitness value as the value. The fitness name is a string representation of what the fitness value measures, for example "Scenario X" can be a fitness name which scores the Chromosome on how well it performs in a specific scenario.

The baseline class exposes getters and setters for behaviour trees and fitness, as well as two other methods – the singleObjectiveComparator method and the functionallyEquals method. The singleObjectiveComparator method returns a Comparator that can be used to sort a collection of Chromosomes based on a specific fitness value. The functionallyEquals method is called on a Chromosome instance and takes another Chromosome instance as an argument. It compares the two Chromosome instances and return true if all their fitness values are equal.

FitnessEvaluator

The FitnessEvaluator is an interface that needs to be implemented in order to evaluate the Chromosomes. What data and how it should be processed and scored will typically be specific to each experiment, and the implementation of the FitnessEvaluator interface should be provided per experiment. The interface specifies only one required method, the evaluate method, which takes a Chromosome instance, a list of example DataSet instances and a list of DataSet instances resulting from simulating the Chromosome as arguments. Each DataSet corresponds to data recorded from a scenario. The order of the DataSet lists are the same, so that index 0 in both the lists correspond to the same scenario. The evaluate method must return a map that has fitness names as keys and fitness values as values.

NSGA-II Algorithm Implementation

The only included implementation of the Algorithm abstract class is the NSGA2 class. We implemented the NSGA-II algorithm, with a few modifications, as described in Subsection 4.5.1. For use with the NSGA-II algorithm, we had to extend the functionality of the baseline Chromosome class. The NSGA2Chromosome class has extra functionality for comparing and storing non-domination ranks and crowding distance.

5.2.5 Behaviour Tree Package

For representing behaviour trees used during simulation, we use the gdxAI library. Behaviour trees represented with the gdxAI behaviour tree framework has limited structural modularity once created. In order to allow easier manipulation of the behaviour trees during evolution, we created a simple hierarchical representation that works as a blueprint for how the executable trees should be built once needed for simulation. The blueprint representations are suitable for performing genetic

operations. Therefore, it is required that each type of behaviour tree node used in our system has two implementations, one for the blueprint behaviour trees and one for the executable behaviour trees. The code for processing simulation data and producing instructions for the simulated entity are only implemented in the executable nodes.

The gdxAI library defines a tree of nodes as a *task*, where the nested children of the root node define sub-tasks. When referring to a node as a task, it includes all nodes in the tree spanning from that node. Our implementation of the blueprint behaviour tree representation follows the same naming convention, and as the nodes are typically referenced as the root of a tree, we use "task" instead on "node" in the class names.

Blueprint Task Template

The blueprint task template package is located in the behaviour tree package, and provides a framework for creating nodes for use in blueprint behaviour trees. The structure of the nodes mirrors the structure used in the gdxAI framework. The blueprint task template contains the Task class, the CompositeTask class and the LeafTask class, as well as the ConditionTask and VariableLeafTask interfaces.

Task Each node in a behaviour tree is an instance of some class that extends the Task class. The Task class is abstract and cannot be instantiated. An instance of the Task class may have a single parent and any number of children. Classes that implement the Task class must also implement three methods: getDisplayName, cloneTask and instantiateExecutableTask. The instantiateExecutableTask method must return the gdxAI Task class that this blueprint Task represents. For each blueprint Task class there must therefore be a gdxAI Task class with the implementation of the executable behaviour of that blueprint Task. The Task class also includes some static utility methods, e.g. methods for removing unnecessary nesting of composite nodes in a task (hierarchy of nodes).

CompositeTask Composite tasks are tasks that have one or more children. The CompositeTask class is also an abstract class which provide utility methods for the different types of composite tasks – *sequence* and *selector*. These utility methods include addChild, insertChild, removeChild, replaceChild, shuffleChildren, etc. These methods are used by the genetic operations to alter the behaviour tree in some way. See Subsection 4.5.2 for more information on the genetic operations.

The system includes two CompositeTask implementations: the Sequence task, and the Selector task. The Sequence and Selector implementations are not experiment-specific and is therefore part of the core system. For information on the concept of sequence and selector nodes in behaviour trees, see Section 2.2. The Sequence and Selector blueprint implementations extend the CompositeTask base class and implements the getDisplayName, cloneTask and instantiateExecutableTask abstract methods of Task base class.

LeafTask Leaf tasks are tasks that have no children. Any task in the system that is meant to be a leaf task must implement the LeafTask abstract class, or implement the VariableLeafTask abstract class which in turn extends the LeafTask. In the system there are two types of leaf tasks: the condition task and the action task. The condition tasks are nodes that do not alter the environment, but tests some condition on the current environment and returns Success or Failure based on that condition. The action tasks are used to instruct the controlled simulated entity to perform an action.

The VariableLeafTask should be used if the task requires some internal variable(s) for tuning the conditon or action. This can e.g. be meters a target should move or the quantity a condition node should check for. VariableLeafTask implementations need to implement a method randomising the values of the variable(s), within specified limits. It also requires a specialised method for checking if the node is equal to another node. This is because the VariableLeafTask nodes need to check the variable values in addition to the normal structural comparison used in the Task class.

If the task is a condition task then it should also implement the ConditionTask interface. This is primarily used for recognising condition tasks when creating graphical representations of the behaviour trees.

Crossover

The Crossover class is used for performing crossover on the behaviour trees. It provides a single static method, crossover, which takes two behaviour trees as arguments. The first argument is cloned and becomes the child of the crossover operation, a random subtree (Task) in the child is then replaced with a random subtree (Task) from the second argument. The child is then returned by the method. This method does not alter the behaviour trees given as arguments but returns a new behaviour tree instance.

Mutator

The Mutator class is used to select between the seven mutation operators used in the system, and to instruct the selected mutation operator to mutate a behaviour tree. The system calls the static mutate method on the Mutator class, which takes a Task, the Unit class that the behaviour trees are intended for and the current training epoch as arguments. The Unit class is needed to request the UnitTypeInfo object so that the mutation operator can get the types of tasks that can be used in the behaviour tree. The training epoch is used when getting the weight of a specific mutation, where the weights are used for a probabilistic selection between the different mutations , described in Subsection 4.5.2.

Mutation

The Mutation class is an abstract class that must be extended by all mutation operator implementations. The class has a constructor that takes a weight and a

factorBase as arguments. It also implements a getWeight method which returns the current weight of the mutation which takes the current epoch as its argument. Subsection 4.5.2 gives an explanation on how the weight for a specific epoch is calculated.

The class also defines an abstract method called canBePerformed which must be implemented by the mutation operators. The method takes a Task instance as its argument and should return true if the mutation can be performed on the Task instance, and false otherwise. For example, a mutation that removes two nodes from a behaviour tree should return false if there are only one node in the behaviour tree. The Mutation class also defines an abstract method called mutate which takes a Task instance and a Unit class as arguments. The method is used to actually perform the mutation operation on the Task instance. The method should return a new Task and not alter the Task instance given as argument.

All seven mutation implementations included in the system are described in Subsection 4.5.2.

5.2.6 Visualisation Package

Visualisation of behaviour trees and fitness history is handled by the Visualisation package. JGraphX[2] is used to generate a JScrollPane (Swing[3] component) that contains the graphical representation of a behaviour tree. The JScrollPane can be used in a JFrame to render the figure to the user. For visualising the fitness history, JFree[4] is used to plot historical fitness data, creating a JPanel (a Swing component) that can be rendered in the same JFrame as the graphical representations of the behaviour trees.

5.2.7 Experiments Package

For each experiment, the user has to create their package containing the classes required to run the experiment. These packages are intended to be placed in the Experiments package. The Experiments package contains two classes that are used for setting up an experiment – the ExperimentInitialiser and the UnitTypeInfoInitialiser. Following are description on these two classes. For more information on how the experiment-specific packages should contain, see Section 5.3.

ExperimentInitialiser

The experiments package includes the ExperimentInitialiser class which has a single static method called setup. The setup method takes a UnitTypeInfoInitialiser instance, a AddUnitMethod instance, a boolean called startRti, a boolean called startSimEngine and a boolean called startSimGui. The three boolean arguments

[2] `https://github.com/jgraph/jgraphx`
[3] `https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`
[4] `http://jfree.org`

tell the ExperimentInitialiser if it should start the RTI, simulation engine and simulation GUI. The method will first call the initUnitTypeInfo method on the UnitTypeInfoInitialiser instance. It will then register the AddUnitMethod instance in the UnitHandler. Then, it will start the RTI if specified, instruct the HlaManager to connect to the HLA federation, and add the SimController as a tick listener and physical entity update listener in the HlaManager. Finally, it will start the simulation engine and the simulation GUI if specified.

UnitTypeInfoInitialiser

The UnitTypeInfoInitialiser is an interface that the user should implement in their experiment package. The interface contains a single method called initUnitTypeInfo. The method is called by the ExperimentInitialiser and should add every UnitTypeInfo to the UnitHandler.

5.3 Setting up an Experiment

When creating a new experiment, the user has to create classes for all experiment specific system components. The necessary implementations to run an experiment include specialisations of the Unit, DataRow and Task classes, an implementation of the FitnessEvaluator interface, instances of UnitTypeInfo for registering the different types of Units that should be used, and an implementation of the AddUnitMethod interface for registering simulated entities as Units. The experiment also requires a main class which is the entry point of the system. The main class should contain methods for starting the recording or training process. For more information on how these processes are initiated, see Subsection 5.4.2.

To register the experiment-specific instances of UnitTypeInfo, the user must implement the UnitTypeInfoInitialiser interface. Listing 5.1 shows an example of how this can be done.

When the user wants to start the training or recording phase, they must first call the static setup method on the ExperimentInitialiser class located in the Experiment package. This class takes the UnitTypeInfoInitialiser, the AddUnitMethod and three boolean arguments. The three boolean arguments tell the ExperimentInitialiser if it should also start the RTI, the simulation engine and the simulation GUI.

If the user wants to start the training phase, they must first instantiate the Algorithm implementation that should be used with the algorithm-specific arguments. Then, an instance of the Trainer class can be created, with the class of the Unit type to train, the DataRow implementation to use for evaluation data, the FitnessEvaluator implementation to use for evaluating the Chromosomes, the created Algorithm instance, and paths to example data files. Finally, the train method must be called on the Trainer instance. An example of how to start the training process is shown in Listing 5.2.

Code 5.1: Example of how to register unit type information.

```
1
2 public class Experiment1UnitTypeInfoInitialiser implements
        UnitTypeInfoInitialiser {
3     @Override
4     public void initUnitTypeInfo() {
5         UnitHandler.addUnitTypeInfo(
6                 name: "Follower",
7                 symbol: "F",
8                 class: FollowerUnit.class,
9                 availableLeafTasks: Arrays.asList(
10                        MoveToTargetTask.class,
11                        WaitTask.class,
12                        IsApproachingTask.class,
13                        IsWithinTask.class,
14                        TurnToTargetTask.class
15                ),
16                availableCompositeTasks: Arrays.asList(
17                        Selector.class,
18                        Sequence.class
19                )
20        );
21        UnitHandler.addUnitTypeInfo(
22                name: "Wanderer",
23                symbol: "W",
24                class: Experiment1Unit.class,
25                availableLeafTasks: new ArrayList<>(),
26                availableCompositeTasks: new ArrayList<>()
27        );
28    }
29 }
```

Code 5.2: Example of how to initiate the training process.

```
1 private static void train() {
2         ExperimentInitialiser.setup(
3                 new Experiment1UnitTypeInfoInitialiser(),
4                 new Experiment1AddUnitMethod(),
5                 startRti: false,
6                 startSimEngine: true,
7                 startSimGUI: false
8         );
9
10        Algorithm<FollowerEvaluationDataRow, NSGA2Chromosome>
          algorithm = new NSGA2<>(
11                populationSize: 30,
12                crossoverRate: 0.5,
```

```
13                 minimumTreeSize: 3,
14                 maximumTreeSize: 12
15         );
16         String[] exampleFileNames = new String[]{
17                 "experiment1/brooklyn.csv",
18                 "experiment1/village.csv",
19                 "experiment1/makland.csv"
20         };
21         Trainer trainer = new Trainer<>(
22                 FollowerUnit.class,
23                 FollowerEvaluationDataRow.class,
24                 new Experiment1FitnessEvaluator(2),
25                 algorithm,
26                 exampleFileNames
27         );
28 
29         trainer.train(200);
30     }
```

5.4 System Processes

This section gives a high level description of four important system processes. These are the system initiation process, the simulation process, the recording process, and the training process. Each of the processes are documented in more detail in the sequence diagrams found in this section. An explanation of how to read the sequence diagrams is given in Subsection 5.4.1.

5.4.1 Sequence Diagram Explanation

Each sequence diagram has an ID which is shown in the top left corner of the figure. The sequence diagrams for system initiation and simulation (Figures 5.7 and 5.8) also have process parameters specified after the diagram ID. All diagrams can be referenced by their ID, and the ones with parameters can be referenced with different arguments that affect the sequence of events.

System entities are colour-coded according system diagram colour-coding standard we specified in Table 5.1. In addition, the sequence diagrams use a separate colour-coding for entity activity blocks (the rectangular shapes along the entity lifelines), where colours indicate different processing threads. Thread colour indications are consistent through the diagrams, meaning the blue active blocks in one diagram will be on the same thread as the blue active blocks in the other diagrams, etc.

The diagrams contain four different types of fragments, shown as frames with an green card in the top left corner which contains a fragment code. The fragment codes and their associated fragment names and meanings are shown in Table 5.2.

Table 5.2: Sequence diagram fragment explanations (used in Figures 5.7 to 5.10)

Code	Name	Explanation
Alt	Alternative	Indicates mutually exclusive choices, with the choices separated by a stippled line.
Opt	Option	Indicates an optional sequence, with potential conditions included in square brackets.
Loop	Loop	Indicates a repetitive sequence, with looping condition included in square brackets.
Ref	Reference	References a process shown in another diagram by diagram ID, with potential diagram parameter arguments.

For a more detailed description of the different types of fragments, see the IBM DeveloperWorks page on sequence diagrams[5]

5.4.2 Initiation Process

The system initiation process starts when the Experiment[x] is initialised. The complete system initiation process is shown in Figure 5.7. The Experiment[x] class will first instantiate the Experiment[X]UnitTypeInfoInitialiser and Experiment[x]AddUnitMethod classes. It will then call the setup method in the ExperimentInitialiser, with the instantiated classes as arguments in addition to three booleans that specify if the RTI, simulation engine and simulation GUI should be initiated as well. The ExperimentInitialiser will add all UnitTypeInfo objects from the Experiment[x]UnitTypeInfoInitialiser and specify to the UnitHandler that the Experiment[x]AddUnitMethod should be used when adding new simulation entities to the system. It will then start the RTI, if this is specified in the options, and connect the federate to the RTI. It will then add the SimController instance as a “tick listener” and “physical entity updated” listener. Finally, it will start the simulation engine and simulation GUI if this is specified.

5.4.3 Simulation Process

The entire process of running simulations is shown in Figure 5.8. All communication between our system and the simulation system goes through the HlaManager, the HlaLib library and the RTI, as shown in the diagram. As the process starts, the HlaManager thread which is responsible for advancing simulation time and notifying the system of new simulation ticks is waiting for the TIME_ADVANCE_LOCK to be opened. This means that our system will not allow time advancement in the federation simulation time until the TIME_ADVANCE_LOCK is opened.

The simulation process begins with the loading of a scenario. The user may manually instruct the MÄK simulation engine to load a scenario by using the

[5]https://www.ibm.com/developerworks/rational/library/3101.html

MÄK GUI application, or the system may automatically load the scenario during training. The choice between these is shown by the Alt fragment in the sequence diagram. When the system automatically loads a scenario, it will first reset the UnitHandler and the UnitLogger. This will clear any units already stored and close any file streams that are currently open. The SimController will then instruct the simulation engine to load the scenario that should be simulated by by calling the sendCgfLoadScenarioInteraction method on the HlaManager with the appropriate scenario path. During training, the scenario path is retrieved from the provided example file(s). The HlaManager will then forward the interaction to HlaLib, which sends the command to the RTI, which again forwards the command to the MÄK simulation engine to load the specified scenario.

Once a scenario has been loaded, the system will call the play method on the SimController. The SimController will then wait for the simulation entities to be discovered. Once the system has received all the simulation entities that it expects, it will wait for all the simulation entities to receive a data update. This ensures that the units have all information that is required, such as marking, location, velocity, etc. Once all the simulated entities have been discovered and updated, the system will send a play interaction to the RTI. This is done by calling the sendCgfPlayInteraction method on the HlaManager which sends the CgfInteraction to HlaLib. When the system has sent the play interaction, it will call the enableTimeAdvancemenet method on the HlaManager. This will open the TIME_ADVANCE_LOCK, which allows the system to tick the simulation forward.

For each tick in the simulation, our system will receive data updates from the simulation, update all internal units in the system with the new data, write any data updates to CSV files, and send instructions to the simulated entities. Once this is done, the HlaManager will then request a new time advance from the RTI. Figure 5.6 shows a simplified overview of the per-tick communication between the system, the RTI and the simulation engine. When the system has run the simulation for the specified number of ticks, it will pause the simulation and notify any listeners that the simulation has ended.

5.4.4 Recording Process

The process of recording behaviour using our system is shown in Figure 5.9. The user must first manually start both the RTI and the simulation engine, and manually load a scenario in the simulation engine. Once this is done, they may start the recording mode of the system, which is initiated by the record method call on the Experiment[x] class. The system will then begin the system initiation process, described in Subsection 5.4.2. Then, the system will call the play method on the SimController with the number of ticks to simulate set to unlimited. The system does not know how long the user wants to record behaviour for, and the user must therefore manually stop the system when they have finished recording the behaviour. The system starts the simulation process once the play method is called. The simulation process is described in Subsection 5.4.3. The system will record data in the background while the simulation is running and continuously write the data

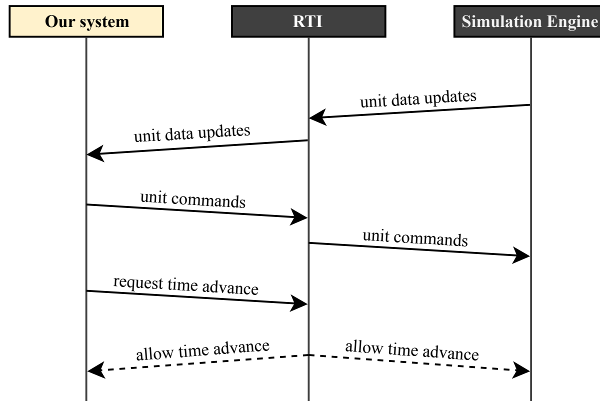


Figure 5.6: Communication sequence diagram of HLA communication per system tick during simulation

to CSV files. When the user has finished recording the behaviour, the user needs to place the generated file in the proper location to use it as an example file. The user must also fill in the scenario file path that was used when recording the example in the example file metadata.

5.4.5 Training Process

The training process is started with a call to the train method in the Experiment main class. The complete process is shown in Figure 5.10. The Experiment[X] class will first start the system initiation process, which is documented in Subsection 5.4.2. It continues by creating an Algorithm and a FitnessEvaluator instance. It then instantiates a Trainer instance with the Algorithm and FitnessEvaluator instances as part of the constructor arguments. 5.2 shows an example of how to instantiate the Algorithm and Trainer. The Trainer instance will call the setup method in the provided Algorithm instance and read the example data sets from the specified files. When the setup method is called on the Algorithm instance, it will perform any steps required before starting the training process. This can e.g. include generating a random starting population.

The Experiment[x] class will then call the train method on the Trainer instance, which starts the training process. The Trainer instance calls the step method on the Algorithm instance once for each epoch. When the step method is called on the Algorithm instance, it will simulate every candidate solution in the population and calculate the fitness for each based on the results of the simulations. See Subsection 5.4.3 for more information on the simulation process. Finally, the Algorithm instance will call the updateFitnessHistory method on the Trainer instance, which stores the fitness values to later show them to the user. The Trainer instance will then continue to the next epoch and will again call the step method on the Algorithm instance.

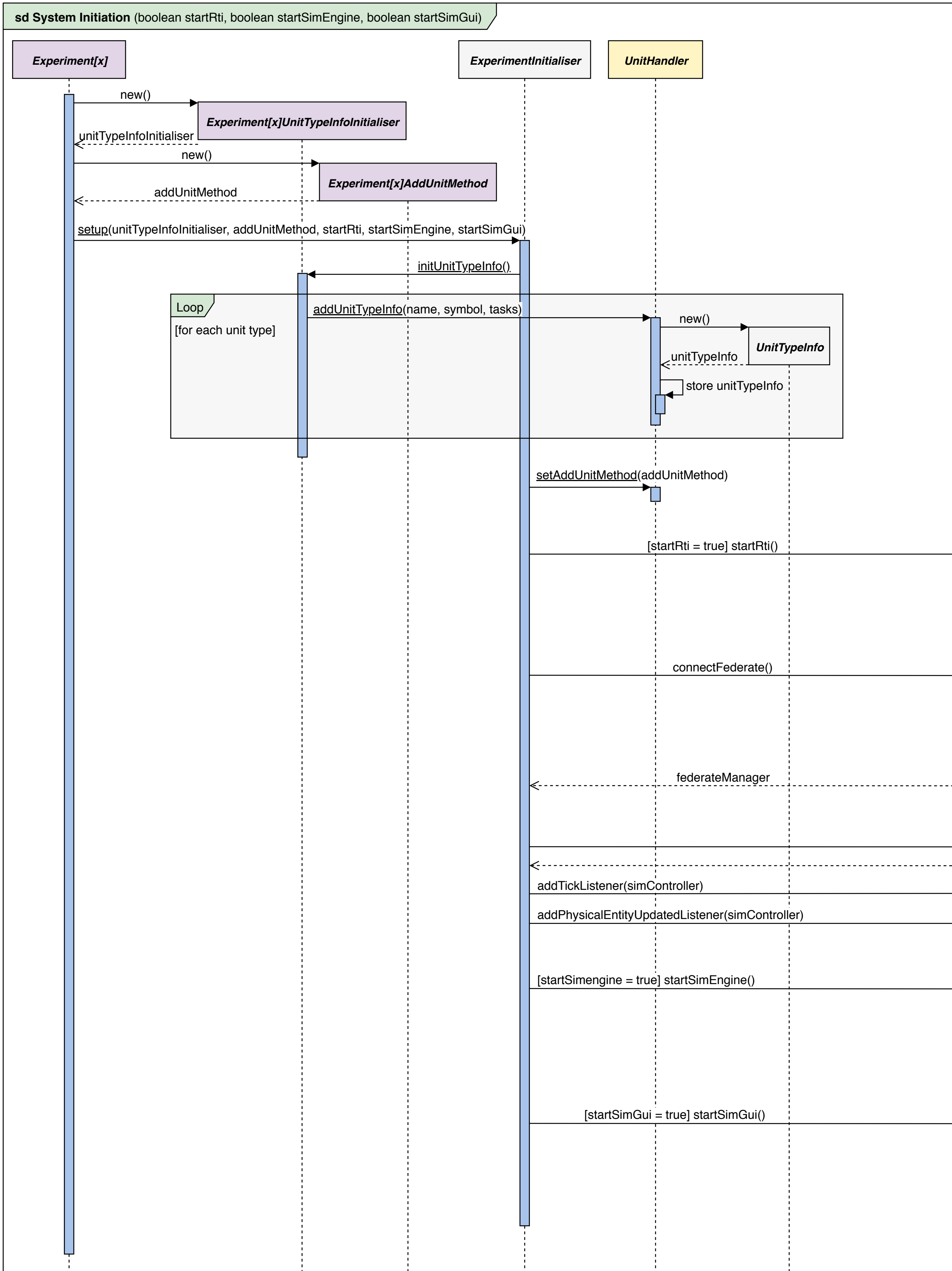
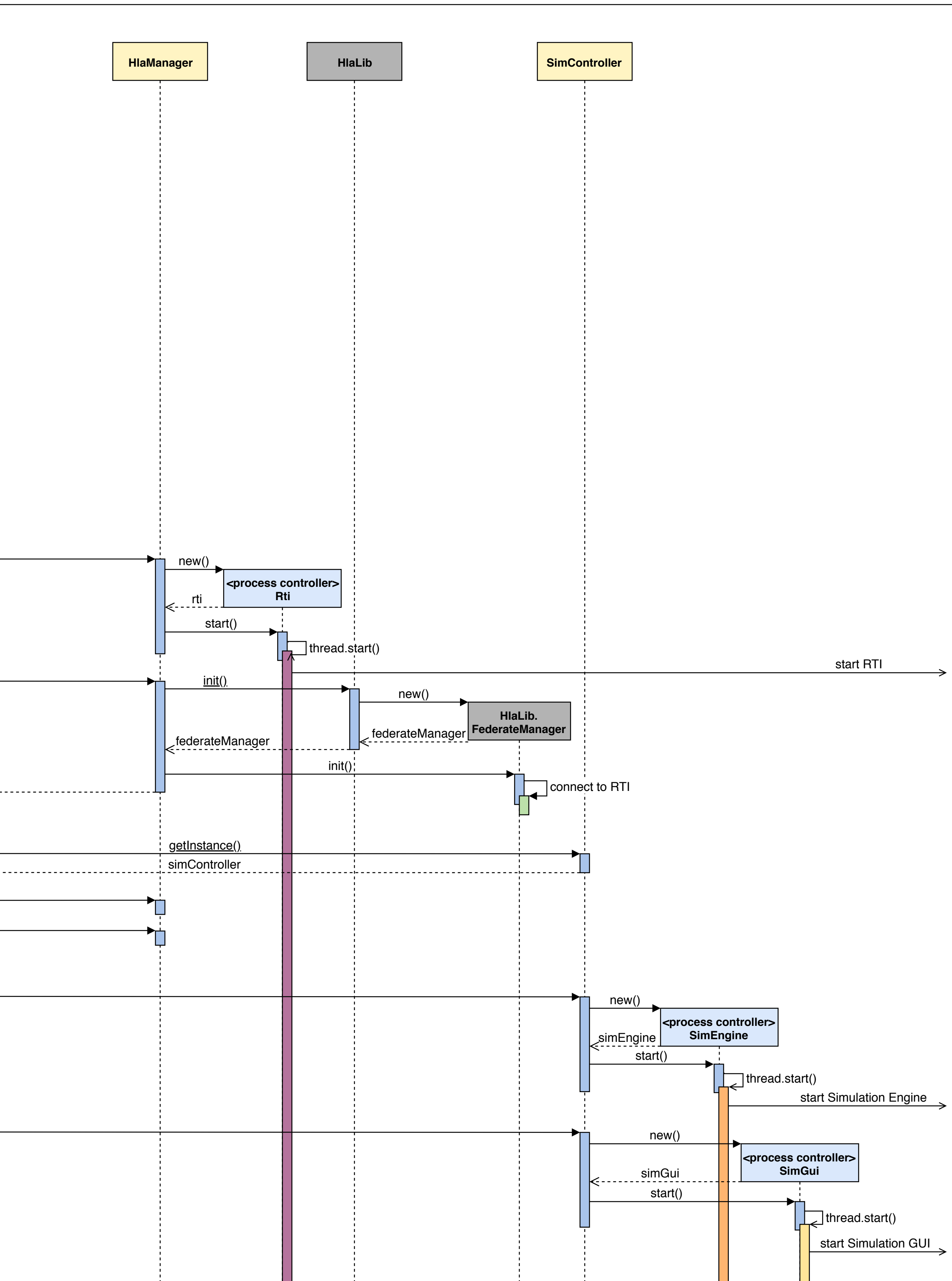


Figure 5.7: Sequence diagram of the system initiation process



Single-page version for electronic viewing is shown in Figure B.2.

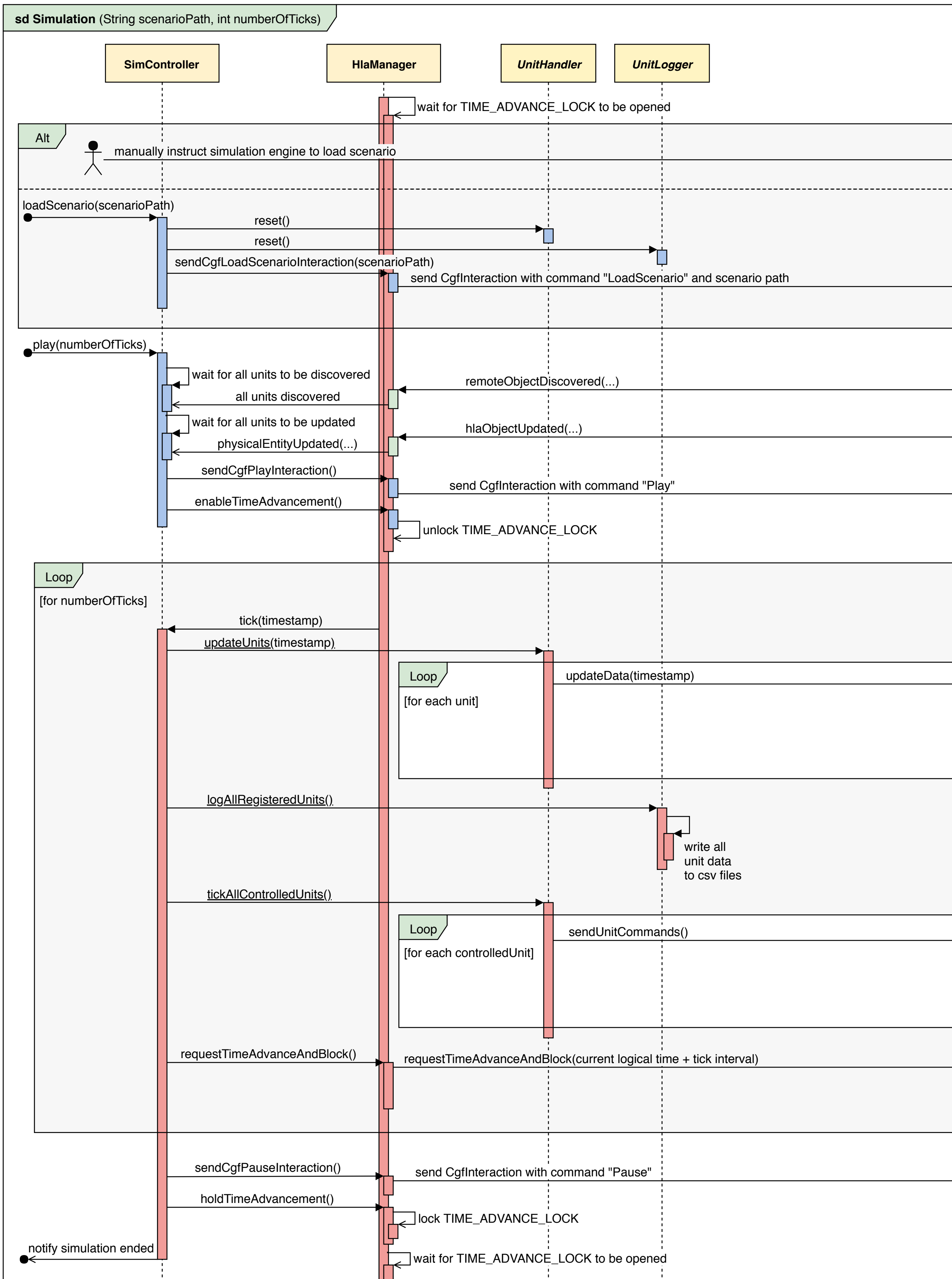
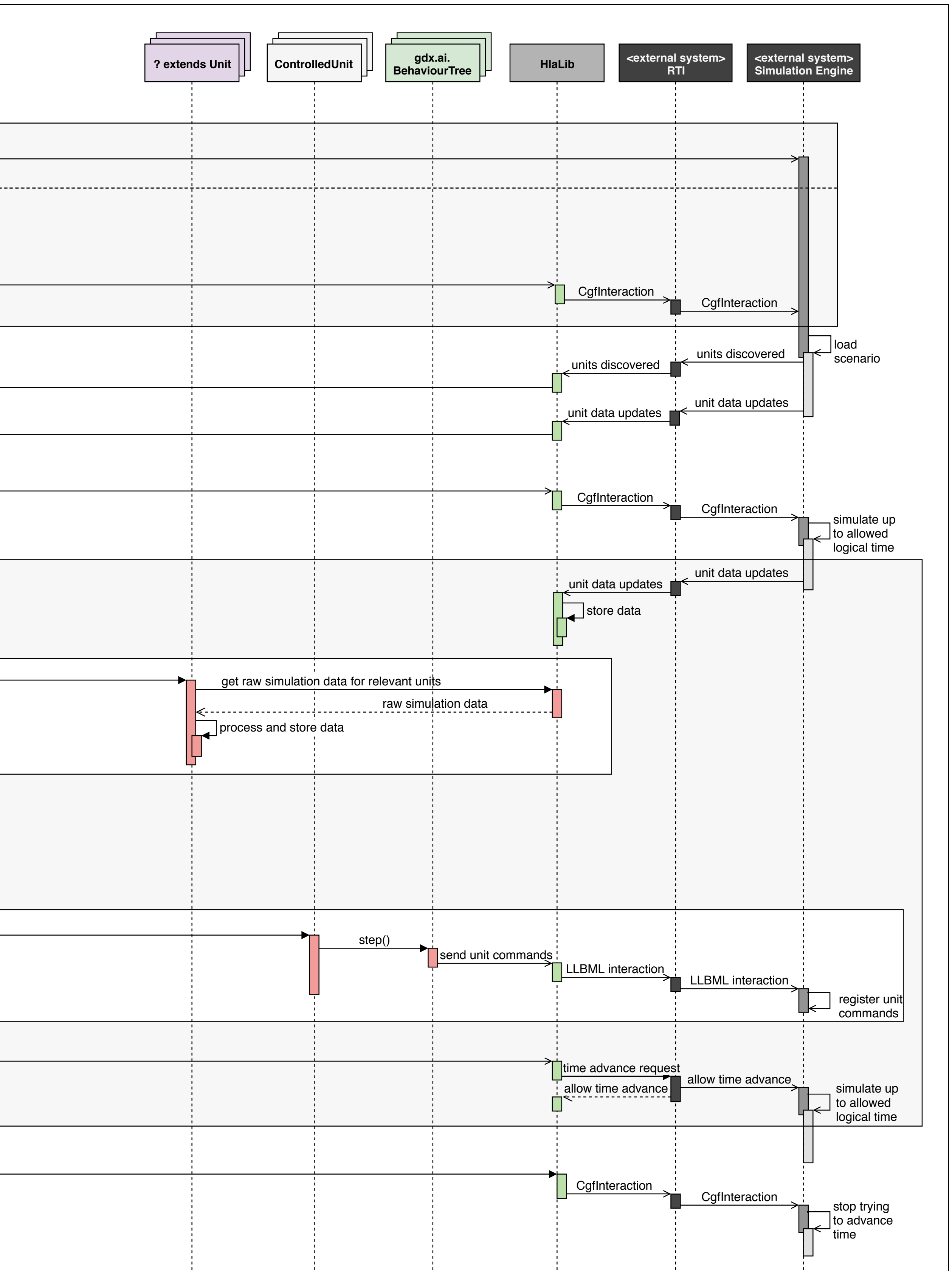


Figure 5.8: Sequence diagram of the simulation process



Single-page version for electronic viewing is shown in Figure B.3.

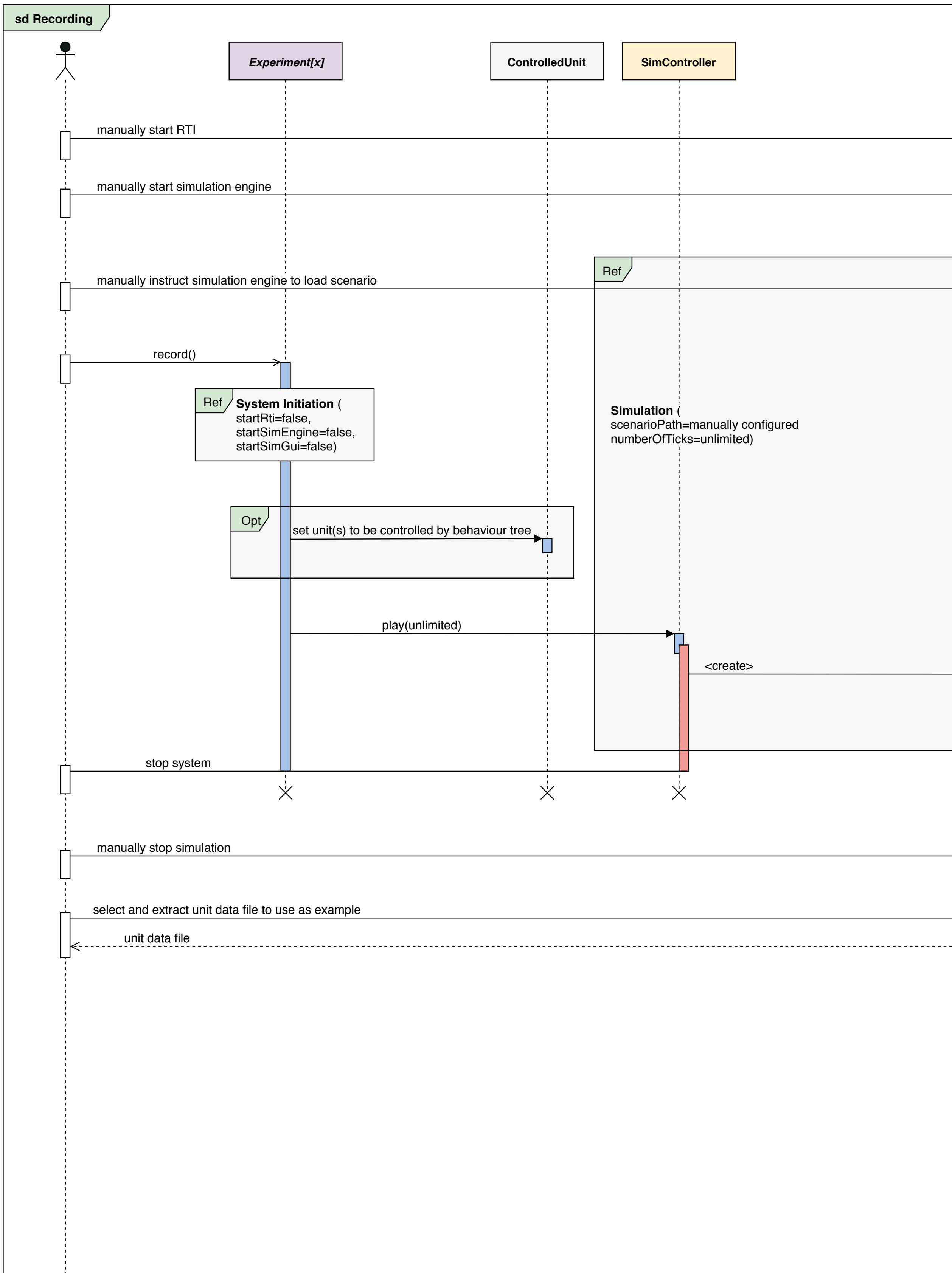
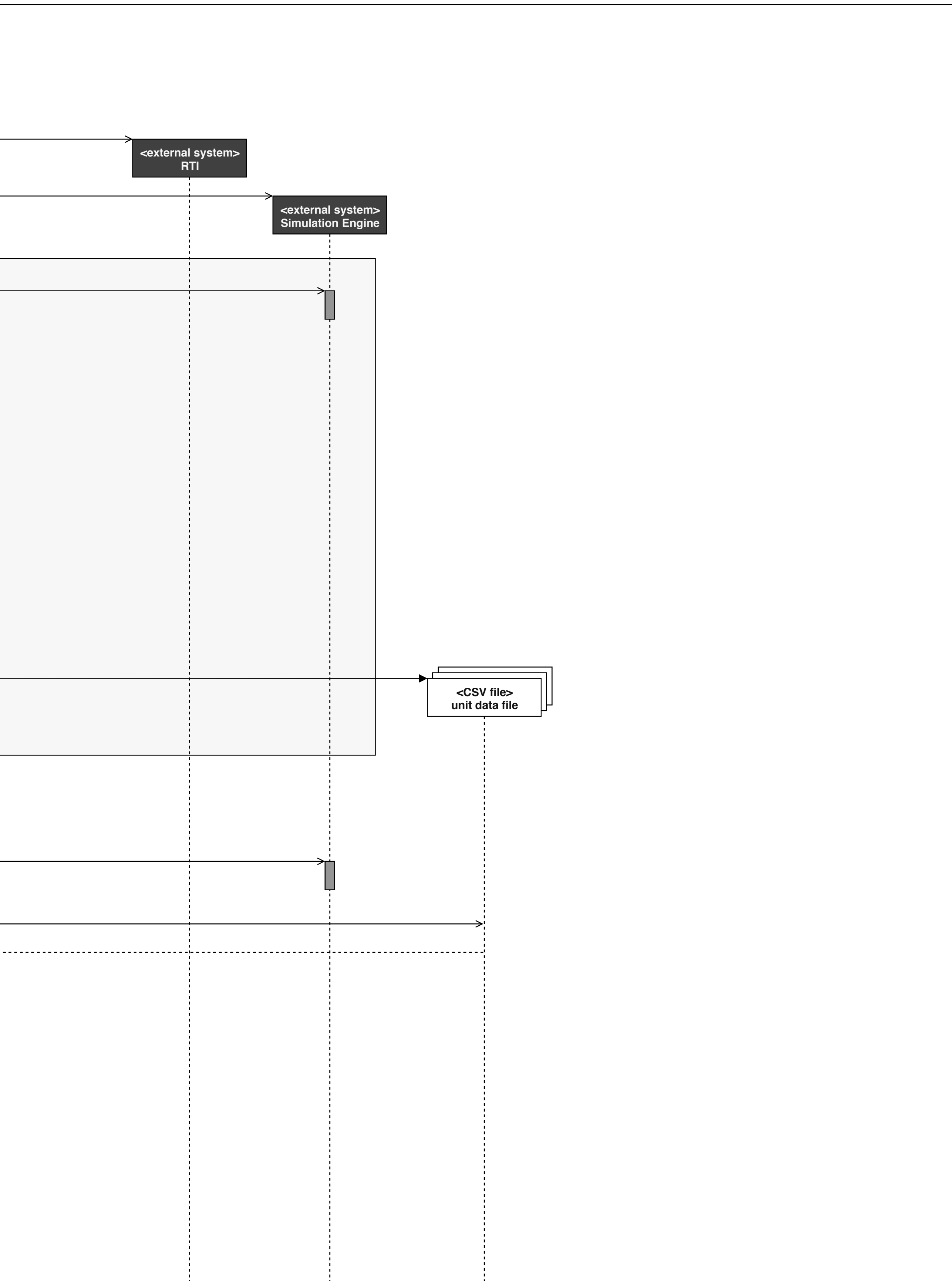


Figure 5.9: Sequence diagram of the recording process. References Figures 5.7 and 5.8 as subprocesses.



Single-page version for electronic viewing is shown in Figure B.4.

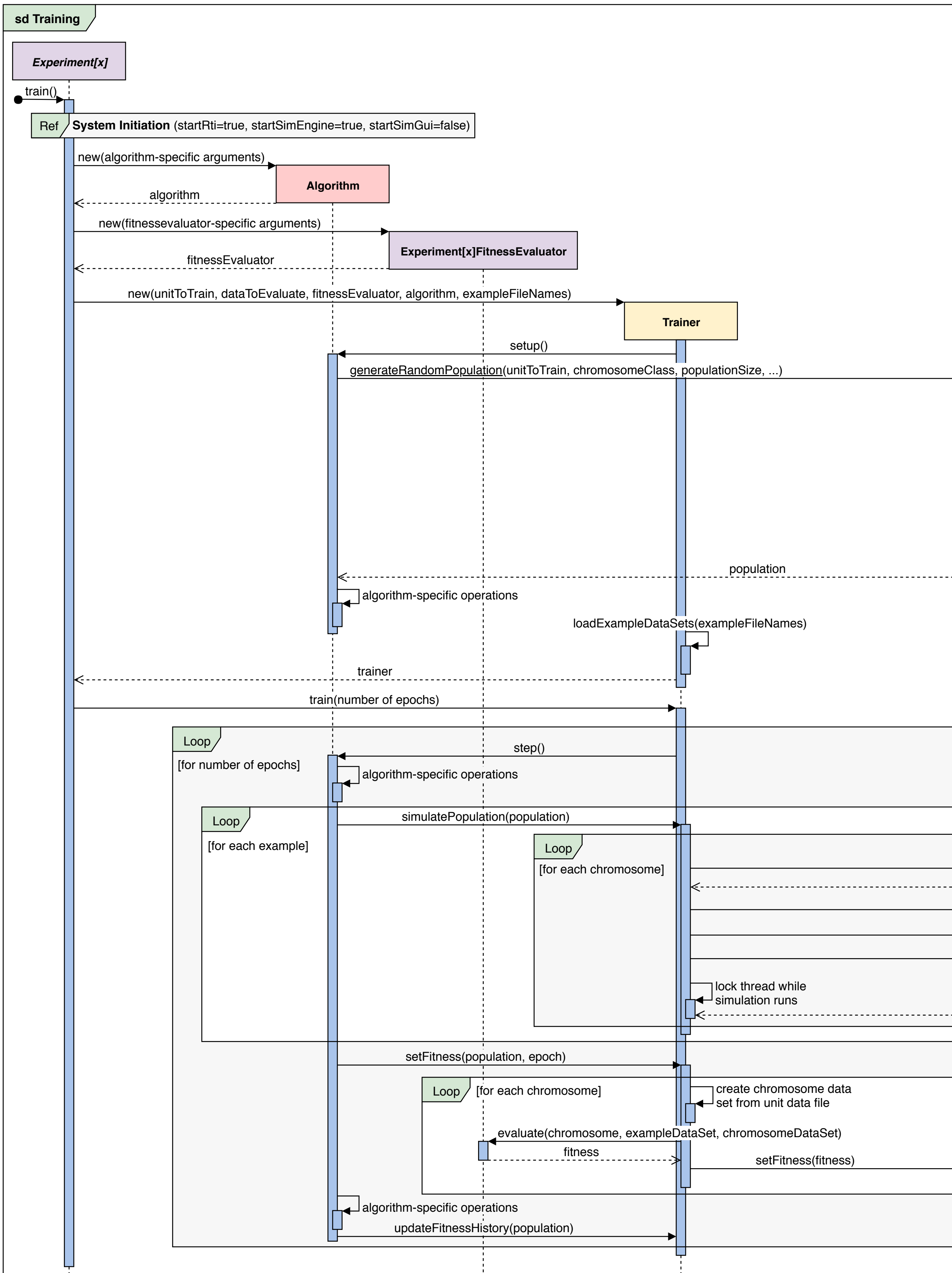
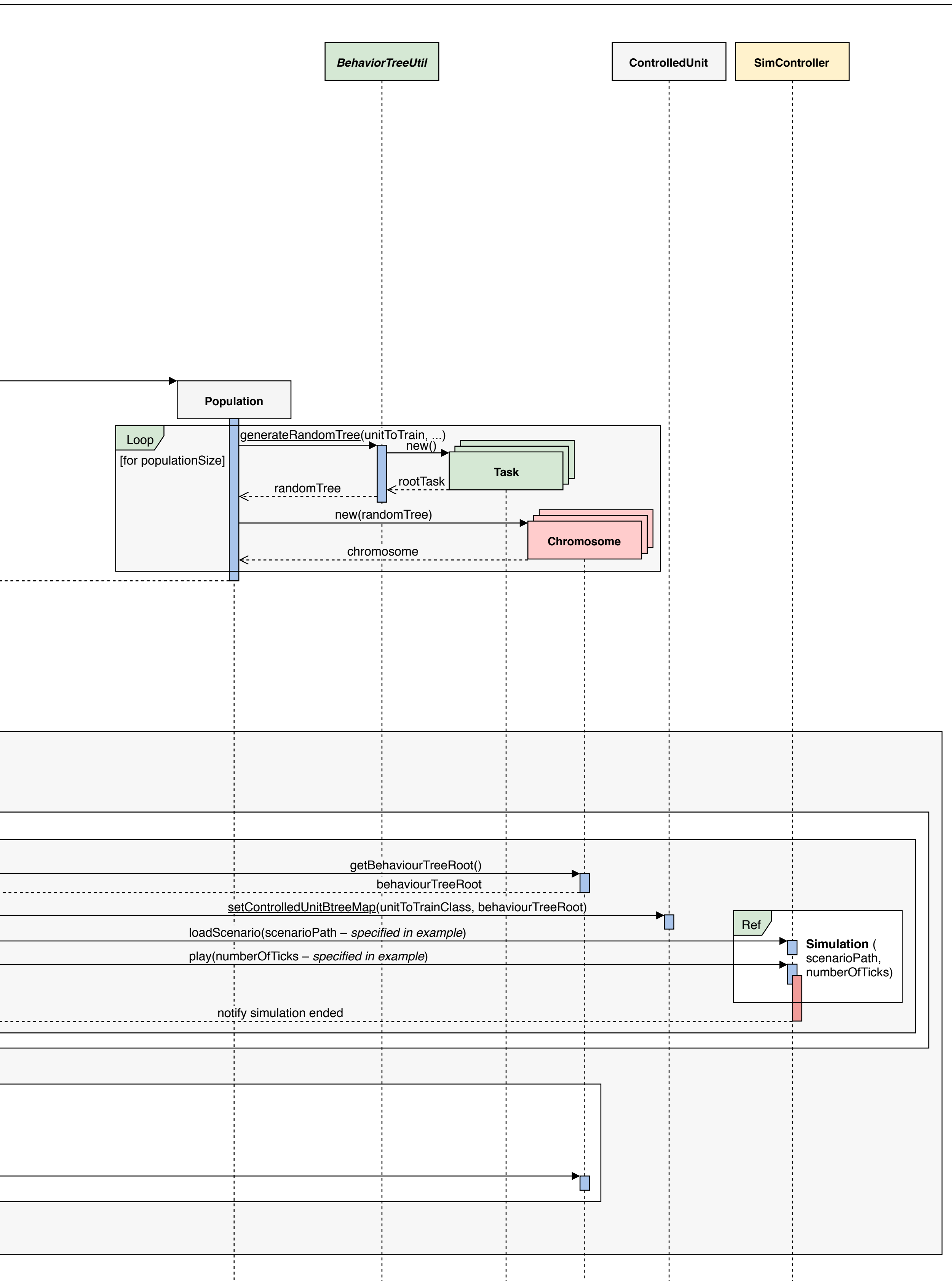


Figure 5.10: Sequence diagram of the training process. References Figures 5.7 and 5.8 as subprocesses.



Single-page version for electronic viewing is shown in Figure B.5.

Table 5.3: Simulation settings used internally in the system

Setting	Value
tickInterval	1.0
simulationTickDelayInMilliseconds	0.0
secondsToWaitForUnitsBeforeReload	10.0
numberOfTicksToCalculateAverageTickTime	100.0

5.5 Settings

This section gives an overview of important settings used in the system. The settings used in VR-Forces are included in Appendix C.

5.5.1 System Settings

The system settings are defined in a single file called SystemSettings. The system settings include settings for simulation data file storage directories and the date format used in the system.

5.5.2 Simulation Settings

All settings that is used when automatically starting the simulation engine and the simulation GUI is given in the SimSettings file, located in the Simulation package. The settings file contains settings that are needed to start the simulation systems and connect to the HLA federation. The file also contains some settings used internally in our system, shown in Table 5.3. The tickInterval controls the interval in logical simulation time between each internal system tick, where the system extracts data from the simulation and sends instructions. The simulationTick-DelayInMilliseconds setting is used to give an artificial delay after a tick, before the rest of the system continues. This setting can be used for slowing down the simulation to better observe the simulation in the MÄK Simulation GUI system. The secondsToWaitForUnitsBeforeReload is used when a new scenario is loaded, where the system will wait x number of seconds to receive units before it reloads the scenario. This is done to prevent the system getting stuck if the simulation entities in a scenario are not properly published to the HLA federation. And finally, numberOfTicksToCalculateAverageTickTime is used to set how often the system should calculate the average tick time that is displayed in the system console output during simulation.

5.5.3 Behaviour Tree Operations Settings

Every mutation in the system takes in two numbers which can be tuned when creating the mutation – weight and factor base. The weight is a number indicating the relative probability of choosing this mutation compared to the other mutations,

Table 5.4: Mutation settings

Mutation	Weight	Factor base
Add Random Leaf Task	1.0	0.995
Add Random Subtree	1.0	1.000
Remove Random Subtree	1.0	1.000
Switch Positions of Random Sibling Tasks	1.0	1.000
Replace Tree With Subtree	1.0	0.990
Replace Random Task With Task of Same Type	1.0	1.000
Randomise Variables of Random Variable Task	1.0	1.020

and the factor base is used to scale the mutation weight over the number of epochs that has been run. Table 5.4 shows an example of how the mutation weights and factor bases can be tuned.

5.5.4 Training Settings

When starting the training process, the user must provide the number of epochs for which the training should be run. In addition, every algorithm can have specific settings that are specified as arguments upon instantiation. The system only implements one algorithm, NSGA-II, which takes in six arguments: population size, crossover rate, minimum tree size and maximum tree size. Each of these settings should be set for every experiment. For an explanation on how the NSGA-II settings are used, see Subsection 4.5.1.

5.6 System Logging

The system logging is built on the Log4j 2[6] library, which can be used to log to the console, file or both. All logging is controlled from a properties file called log4j2.xml located in the system resources directory. The file defines each package or class that should be grouped, sets the log level of the group and where the log output should go (console, file or both). Log4j 2 has eight log levels, of which the system uses four: DEBUG, INFO, WARN and ERROR. Each class that wants to utilise the logger must define a logger instance to use. To create this the class must call the getLogger on the LoggerFactory class that is provided by the Log4j 2 library. The method takes a class as an argument which will appear as the origin of the log messages. An example of the log output is shown in Figure 5.1.

[6] https://logging.apache.org/log4j/2.x/

5.7 Libraries Overview

This section gives a short summary of the libraries used in the system.

- **HlaLib** – HlaLib is used for all communication over HLA, and is a library used internally at FFI. The library is used in the HlaManager class, which handles all HLA communication in our system.
- **LLBML** – The LLBML library is a HlaLib module, which allows the system to control units in the simulation engine that it does not own [38, 39].
- **GeographicLib**[7] – GeographicLib is used in our system for solving geodesic problems, such as finding the shortest vector between two units. The library is used for processing data from the simulation engine and used as a utility for the behaviour tree action tasks.
- **log4j 2**[8] – The log4j 2 library is used for logging system information to the console or log files.
- **gdxAI**[9] – The gdxAI library is used for behaviour tree execution. The library is used in the ControlledUnit class, and every blueprint behaviour tree task must have an executable version built on the gdxAI task framework.
- **JFree**[10] – The JFree library is used to visualise behaviour trees during training.
- **JGraphX**[11] – The JGraphX library is used to visualise the fitness history during training.
- **Batik**[12] – The Batik library is used to write the visualisations of the behaviour trees and the fitness history to the Scalable Vector Graphics (SVG) file format.

5.8 Summary

In this chapter, the details of the system implementation have been described. The implementation of the system is one of the three contributions of this thesis, C3, and the experience from implementing and testing the system has assisted the answering of RQ2.

[7] https://geographiclib.sourceforge.io
[8] https://logging.apache.org/log4j/2.x/
[9] https://github.com/libgdx/gdx-ai
[10] http://jfree.org
[11] https://github.com/jgraph/jgraphx
[12] https://xmlgraphics.apache.org/batik/

Chapter 6

Experiments and Results

This chapter covers the experimental plan, the conducted experiments, and the results. The conducted experiments and the results are evaluated in Section 7.1.

6.1 Experimental Plan

The initial experiment plan included three experiments. The first was a simple experiment designed to verify that the system is able to create behaviour models that imitate simple behaviour. The simple behaviour was to follow a specified moving target on foot. The plan was to manually create a behaviour tree that would represent the described behaviour, use that behaviour tree for controlling a simulated entity, and use the recording of that entity's behaviour as the example behaviour for our system. This way, we have a ground truth representation of the observed behaviour, which can be used for evaluating the generated models. This experiment was designed to help answer RQ1 – how behaviour trees generated with GP perform in imitating observed behaviour in complex, realistic environments. At the same time, the experiment could be used for evaluating the system design and implementation, related to RQ2, and help identify areas for improvement in the system. The results from this experiment were to be submitted to the CogSIMA 2018 conference.

The second planned experiment was to record the same type of behaviour as used in the first experiment, however, with a human controlling the following entity instead of a manually created behaviour tree. This was to introduce subjective personal behaviour traits, and investigate how well the system would be able to imitate these.

The third experiment was planned to involve the learning of a cooperative behaviour, called bounding overwatch, where agents must work together to advance forward. It was also planned to use human-controlled entities for recording example behaviour. The third experiment was designed to evaluate how the proposed methods work on more complex activities.

During the process of conducting the first experiment, we identified significant issues with the system, and fixing the identified issues were given a higher priority than to conduct the second and third planned experiments. After the identified issues had been addressed, we had time for one more experiment. For a new evaluation of the system and the performance of the behaviour trees it generates, we did a new, longer run of the first experiment, with a larger population size. This is hereby referred to as Experiment 2.

The experiments were performed on a system with an Intel i5-6600K @4.4GHz CPU, 16GB RAM and an NVIDIA GeForce GTX 1070 GPU.

6.2 Experiment 1

Experiment 1 revolves around a wanderer and a follower. The objective for the experiment is to have the follower agent follow the wanderer. The wanderer is pre-programmed to follow a specific plan, which involves going to different locations and waiting a given amount of time at certain positions. The example file used for training was recorded from a manually created behaviour tree, shown in Figure 6.3. Using this behaviour tree, the controlled agent would follow the specified target, with two conditions causing it to stop: (i) If the target is within 30 meters, and (ii) if the target is moving toward the controlled agent. This was an easy way to generate training data, and it also made it possible to compare the learned model with the "true" model. A poster article which briefly covers Experiment 1 was submitted and accepted to the CogSIMA 2018 conference, and both the article and the poster is included in Appendix A. This section provides a more detailed description of the experiment.

6.2.1 Data Extraction and Processing

For this experiment, we created three types of DataRows – an object used to hold and process data in our system. The first DataRow, called the raw DataRow, is used hold the raw simulation data for the simulated entities. This includes the latitude, longitude, altitude and movement angle of the entity. The movement angle is stored as an absolute bearing between the current and next position of the entity, and the next position is calculated by adding its velocity vector to its current position vector. This DataRow is used for both the wanderer and follower.

The second DataRow, called the follower processed DataRow, is used by some of the behaviour tree tasks. It stores the euclidean distance between the follower agent and a target agent, and an angle called the approaching angle. The approaching angle calculation is explained in more detail below. This DataRow uses the raw DataRows of both the wanderer agent and the follower agent to calculate the necessary values. This DataRow is only used for the follower.

The third DataRow, called the follower evaluation DataRow, only contains the euclidean distance between the follower and the target. This DataRow is used in

the evaluation of the behaviour trees during training, to determine the fitness of the behaviour tree. This DataRow is only used for the follower.

Approaching Angle

When following an agent, it seems reasonable that knowing to what degree the wanderer is approaching the follower would be valuable information. The follower may, for example, want to stop and wait for the wanderer or change direction based on the angle of approach the wanderer has relative to the follower. We therefore added a calculation that would tell the follower to what degree the target is approaching. Where 0° or 360° means that the target is headed directly towards the follower agent and 180° means that the target is heading directly away from the follower.

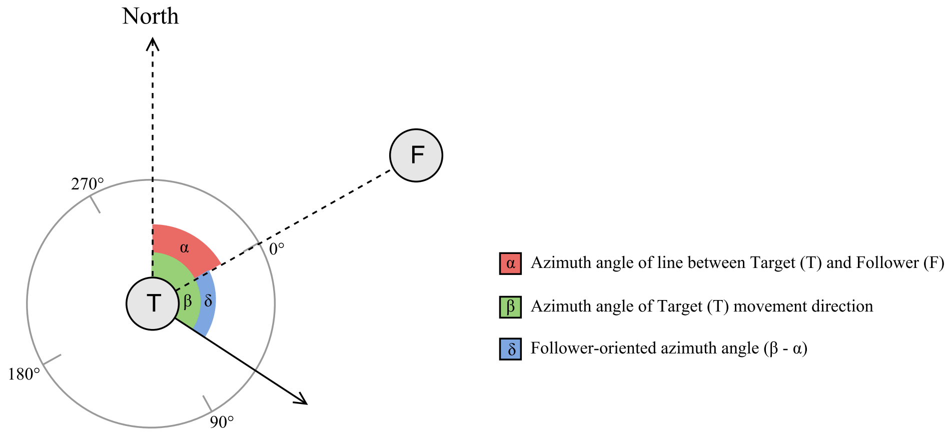
Figure 6.1 shows a visualisation of how the approaching angle is calculated. The first step is to calculate the line from the follower position to the target position and find the absolute bearing of this line. In Figures 6.1a and 6.1b, this angle is shown as the α angle. Then, we have to calculate the absolute bearing of the movement direction for the target. This has already been calculated in the raw DataRow. In Figures 6.1a and 6.1b this angle is shown as the β angle. Finally, we have to subtract α from β to produce the final angle. The final angle is shown in Figures 6.1a and 6.1b as δ. The α might be larger than β, as shown in Figure 6.1b. This results in a negative δ angle. When this happens, we subtract δ from 360° which results in a positive angle.

6.2.2 Behaviour Tree Nodes

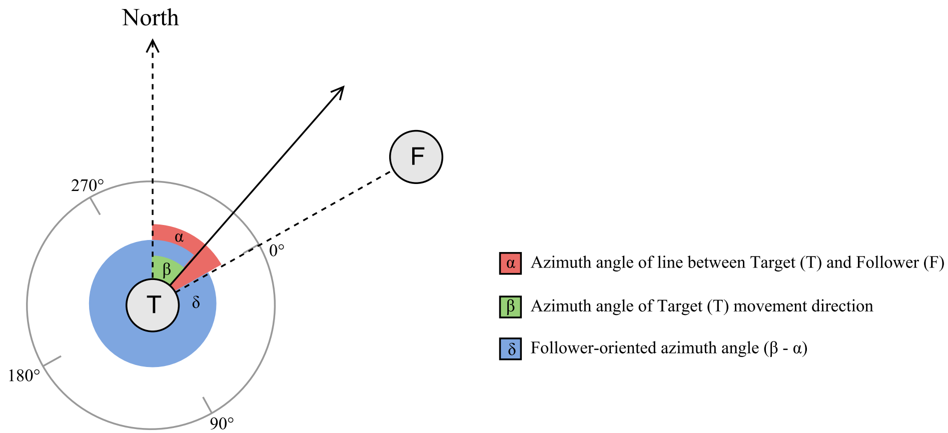
A behaviour tree for a follower unit can have five different types of leaf nodes: *Move to target*, which makes the follower move toward its target. *Turn to target*, which makes the follower turn towards its target. *Wait*, which makes the follower stand still. *Is within*, a variable condition node which checks whether the followers target is within a specified euclidean distance. *Is approaching*, a variable condition node which checks whether the angle between the movement vector of the target and the vector between the follower and the target is smaller than a specified threshold – effectively checking whether the followers target is moving towards the follower.

6.2.3 Scenarios

For this experiment, the training was done on two separate scenarios, using different terrains and wanderer paths. 2D overviews of the terrain and wanderer paths for both scenarios are shown in Figure 6.2. The first scenario simulates approximately 18 minutes of real-time over 1100 ticks, and the second approximately 12 minutes of real-time over 700 ticks.

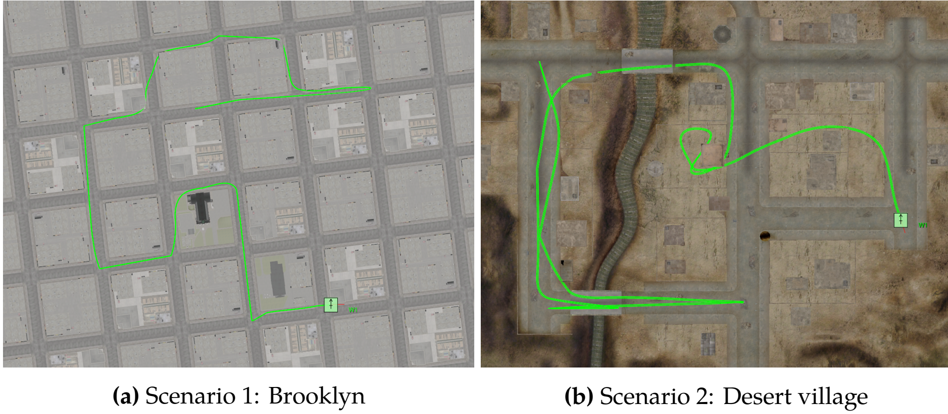


(a) Method for calculating approaching angle when α is smaller than β



(b) Method for calculating approaching angle when α is larger than β

Figure 6.1: Visualisation of approaching angle calculation used in Experiment 1



(a) Scenario 1: Brooklyn

(b) Scenario 2: Desert village

Figure 6.2: 2D view of scenario terrain and target path for the scenarios used in Experiment1.

6.2.4 Fitness Evaluation

The objective is to imitate the recorded behaviour. The fitness function therefore compares the behaviour produced by the behaviour tree with the recorded behaviour. The behaviour trees were evaluated by comparing the euclidean distance in each tick between follower and target position in the training data with the follower-target distance during behaviour tree simulation. All trees were tested on both scenarios. The equation for calculating the fitness value of a behaviour tree for a single scenario is shown in Equation (6.1), where n is the number of ticks that were simulated.

$$Fitness = \frac{1}{n} \times \sum_{t=0}^{n} \Big(dist(example_t) - dist(btree_t) \Big)^2 \tag{6.1}$$

For each of the recorded ticks we find the follower-target distance from the training data and the simulated behaviour tree. The difference of these two distances is then squared so that a large difference over a few ticks is worse than a small difference over a large number of ticks. The squared differences in euclidean distance are summed and normalised over the number of ticks (n) to make the different scenario fitness values more comparable. The fitness values should be minimised. We chose this fitness function, as it is a simple formula that captures the similarity of the example and evaluated behaviour.

During training in this experiment, NSGA-II was set to simultaneously minimise three fitness values: the fitness value from running scenario 1, the fitness value from running scenario 2, and the number of nodes in the behaviour tree. By adding the tree size as a minimisation objective, we prevented the algorithm from creating bloated behaviour trees with unnecessary subtrees that have no significant effect on the behaviour.

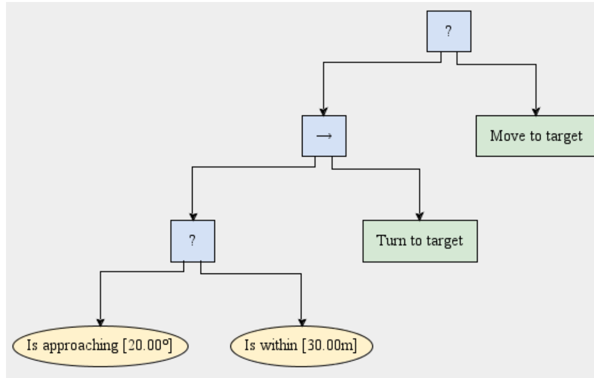


Figure 6.3: Manually made behaviour tree

Table 6.1: NSGA-II settings used for Experiment 1

Setting	Value
Population Size	10.0
Crossover Rate	0.5
Minimum Tree Size	3.0
Maximum Tree Size	12.0

6.2.5 Settings

Table 6.1 shows the NSGA-II settings used for this experiment. The results from this experiment were to be submitted to the CogSIMA 2018 conference, and we had limited time before the submission deadline. Due to the long time it takes to simulate a behaviour tree, we ran the experiment with a small population of 10.

The default mutation settings of the system, shown in Table 5.4, were used for this experiment. The mutations that change the behaviour trees drastically – *add random subtree* and *replace tree with subtree* – were given a factor base of less than 1, while *randomise variable of random variable node* was given a factor base higher than 1. This way, the algorithm prioritises local search over larger changes at later epochs.

In addition, there are a few simulation-specific and system-specific settings which were not changed from the default settings. An overview of these settings can be found in Section 5.5.

6.3 Experiment 2

After we had identified issues with the system during Experiment 1, several improvements were made to how it handles data extraction and entity control. We then had time to do one more experiment. The goal of this experiment was to

see if we could improve on the results from Experiment 1, related to RQ1, and to do a new evaluation of the system, related to RQ2. We used the same scenarios, same entities, same example behaviour and same settings as in Experiment1, but with a population size of 30 instead of 10. The small population size used in Experiment 1 was chosen in order to meet the submission deadline for the CogSIMA 2018 conference, and we wanted to see how the new system performs with a larger population.

6.4 Results

In this section, the results from Experiment 1 and Experiment 2 are presented.

6.4.1 Experiment 1

Experiment 1 ran for 90 epochs, taking approximately 4.5 hours. Figure 6.4 shows the fitness development for Scenario 1 and 2, as well as the size of the behaviour trees over the 90 epochs. Figure 6.5 shows a zoomed in view of the development of best fitness on Scenario 2 over the first 30 epochs. The results show that the generated behaviour trees improve over time. For both scenarios, the trend is that the best and average score is continually decreasing for the first 30 epochs, before stagnating. On Scenario 2, there are minor improvements to the average fitness after the first 30 epochs. A possible explanation for the increases in average fitness values for single fitness objectives is that it is caused by the multi-objective selection of the NSGA-II algorithm.

As can be seen in Figure 6.4a, the fitness development on Scenario 1 stagnated after the first 30 epochs, without approximating zero.

We have chosen two of the non-dominated behaviour trees from the population at epoch 30, shown in Figure 6.6 with fitness values included in the sub figure captions. The behaviour tree in Figure 6.6a has the smallest possible size following the size restrictions, and has the lowest fitness on Scenario 2 of all the trees of the same size. The larger one, shown in Figure 6.6b, performs better at Scenario 1 and Scenario 2, but has more than twice the number of nodes. It is important to note that the best fitness achieved on Scenario 1 and the best fitness achieved for Scenario 2 are not from the same behaviour tree.

6.4.2 Experiment 2

Experiment 2 ran for 148 epochs, taking approximately 39 hours. Figures 6.7 and 6.8 show the fitness development for Scenario 1 and 2 and the size of the behaviour trees over 148 epochs. During this experiment, we observed a more continuous fitness improvement than in Experiment 1. The results show that the generated behaviour trees improve over time. For both scenarios, the trend is that the best, average and worst fitness is continually decreasing as the algorithm is running.

Table 6.2: Comparison of the Scenario 1 fitness values in Experiment 1 and Experiment 2. The values for Experiment 1 are estimates from reading the plot in Figure 6.4a.

Epoch	**30**		**60**		**90**	
Fitness measure	**Average**	**Best**	**Average**	**Best**	**Average**	**Best**
Experiment 1	~150	~84	~150	~84	~142	~84
Experiment 2	114	14	100	1	102	1
Improvement	~24%	~83%	~33%	~99%	~28%	~99%

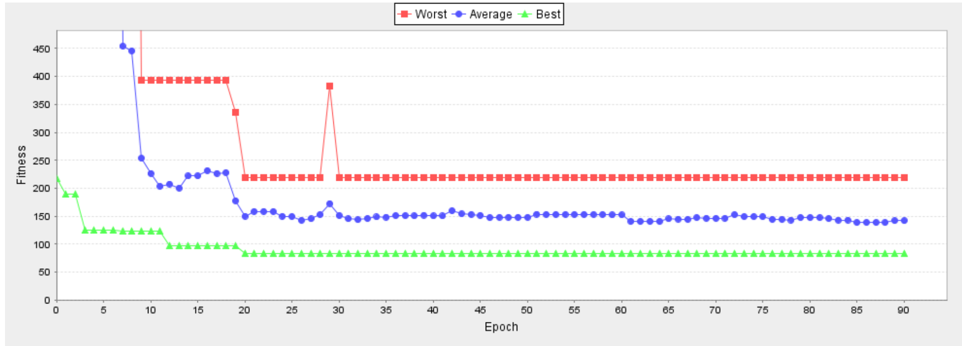
Table 6.3: Comparison of the Scenario 2 fitness values in Experiment 1 and Experiment 2. The values for Experiment 1 are estimates from reading the plot in Figure 6.4b.

Epoch	**30**		**60**		**90**	
Fitness measure	**Average**	**Best**	**Average**	**Best**	**Average**	**Best**
Experiment 1	~117	~1	~103	~1	~90	~1
Experiment 2	87	1.30	47	0.66	57	0.03
Improvement	~26%	~-30%	~54%	~34%	~37%	~97%

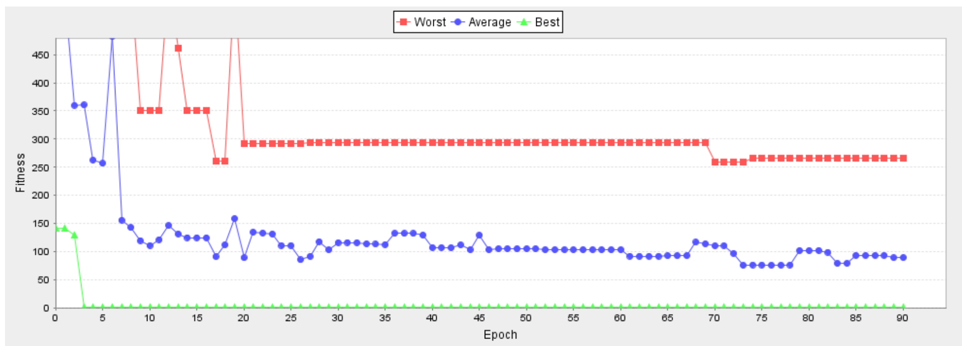
After 141 epochs, the best fitness for both scenarios is close to zero. Again, a possible explanation for the short-term and long-term increases in the average fitness values is the multi-objective selection of the NSGA-II algorithm.

Figure 6.9 shows a graphical representation of the best behaviour tree generated in Experiment 2, with the fitness scores included in the figure caption. Out of the generated behaviour trees, this behaviour tree performs the best on both scenarios.

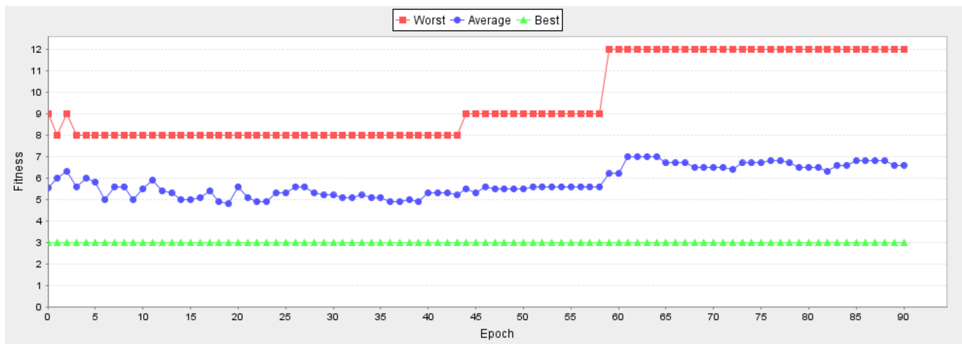
Tables 6.2 and 6.3 show comparisons of the average and best fitness values in Experiment 1 and Experiment 2 for both scenarios respectively at epoch 30, 60 and 90. It should be noted that the Experiment 1 fitness values used in the tables are estimates from reading the fitness history plots in Figure 6.4. This is because we did not have the required data from Experiment 1 to fill in precise values. The fitness values are consistently better in Experiment 2, except for the best fitness on Scenario 2 after 30 epochs, where the best fitness in Experiment 2 is ~30% worse than in Experiment 1. Overall, the fitness values of Experiment 2 are consistently better. The best fitness after 90 epochs is significantly better in Experiment 2 than Experiment 1, with an improvement of ~99% on Scenario 1 and ~97% on Scenario 2.



(a) Scenario 1



(b) Scenario 2



(c) Size

Figure 6.4: Experiment 1 fitness development over 90 epochs

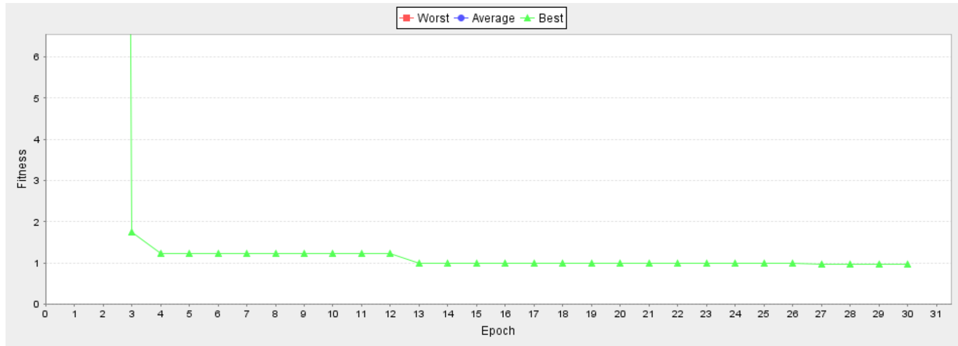
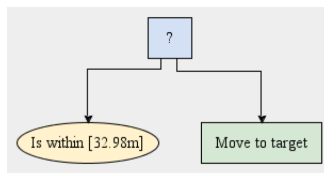
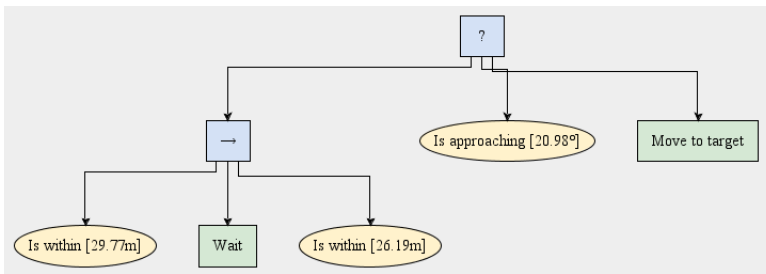


Figure 6.5: Zoomed view of Experiment 1 fitness development on Scenario 2 over 30 epochs

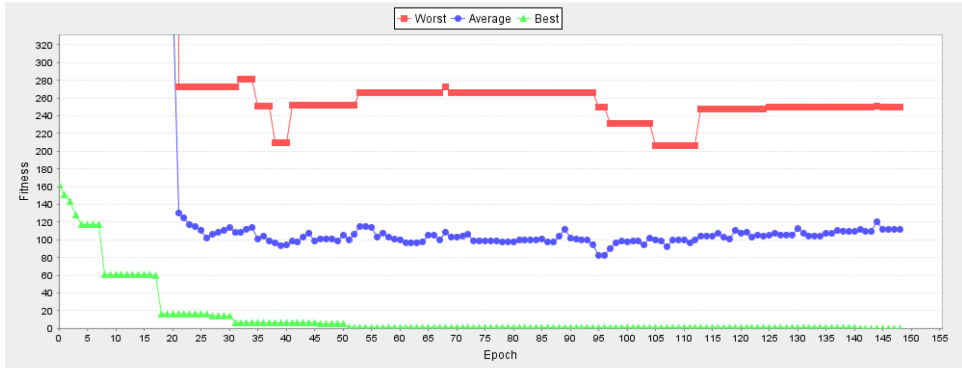


(a) Fitness: [size=3, scenario1=219, scenario2=114]

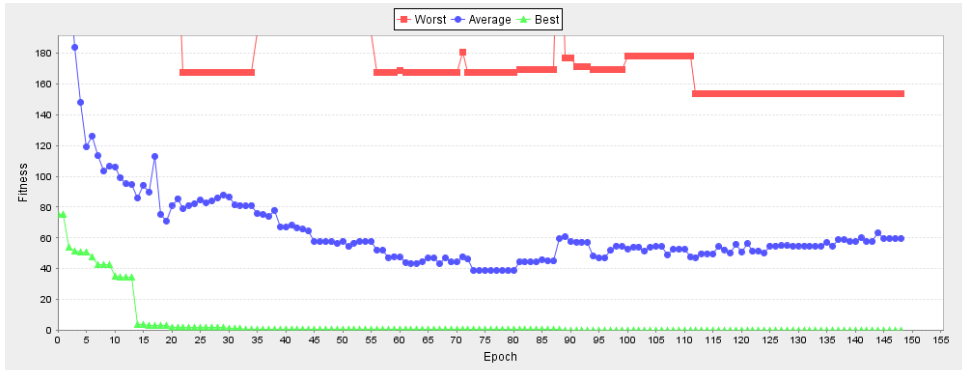


(b) Fitness: [size=7, scenario1=84, scenario2=5]

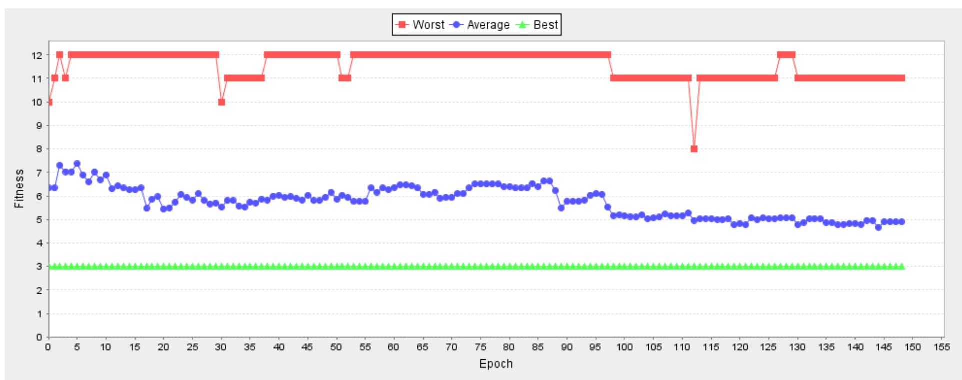
Figure 6.6: Graphical representations of two of the resulting behaviour trees in Experiment 1 after 30 epochs



(a) Scenario 1

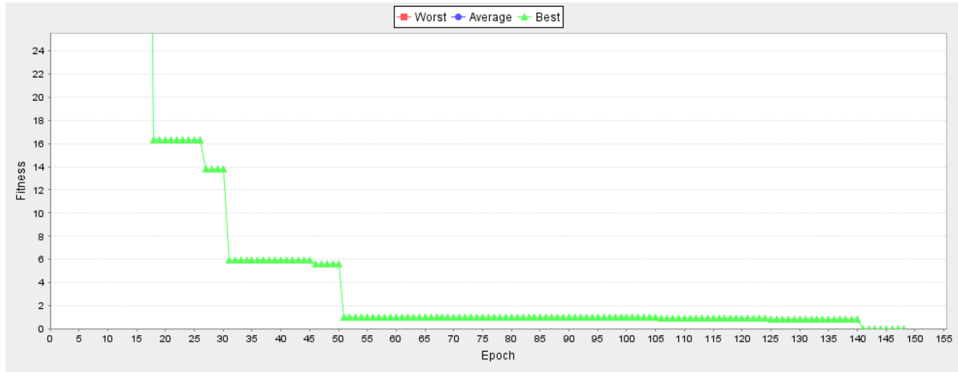


(b) Scenario 2

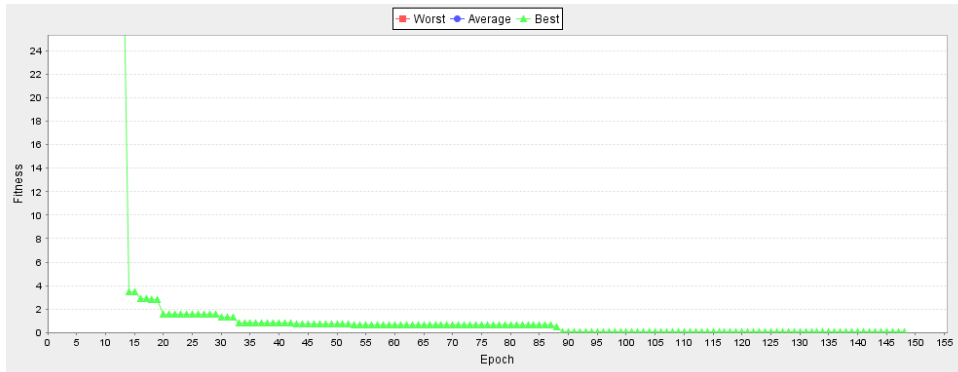


(c) Size

Figure 6.7: Experiment 2 fitness development over 148 epochs



(a) Scenario 1



(b) Scenario 2

Figure 6.8: Zoomed view of Experiment 2 fitness development over 148 epochs

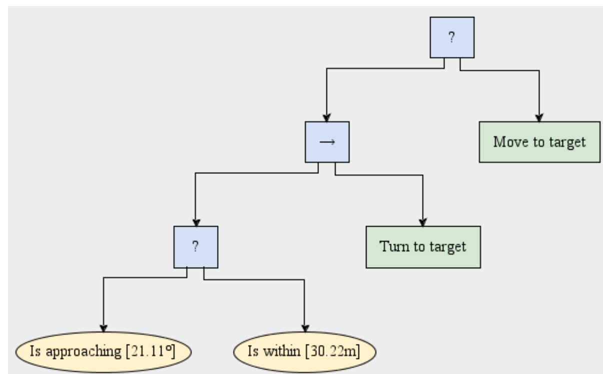


Figure 6.9: Graphical representation of the overall best generated behaviour tree in Experiment 2. Fitness: [size=7, scenario1=0.012, scenario2=0.030]

Chapter 7

Evaluation

In this chapter, the experiments, system, research questions and contributions are evaluated.

7.1 Evaluation of the Experiments

It is important to note that the results from Experiment 1 and Experiment 2 are from single runs of a probabilistic training algorithm with small population sizes. However, they are representative of the general trend in fitness development that we observed while testing the system in the same period as the respective experiment was conducted. Ideally, the performance should have been measured over a large number of runs for each experiment, however, this was not possible due to long simulation times.

The example behaviour used in the two conducted experiments is very simple. It involves only two simple tasks – move and wait – and two conditions – checking distance and relative movement angle of the target. Therefore, the results are only representative of how the system performs with simple behaviour. In addition, the follower's behaviour model is fed ground truth information about the target from the simulation system. This means that the follower always knows the current position of the target, even if the follower unit is unable to observe the targets position and velocity. For a more realistic experiment, the follower behaviour model should only have access to perceived truth, as supported by the LLBML module introduced in Section 4.2.

7.1.1 Evaluation of Experiment 1

In Experiment 1, we used a manually created behaviour tree, shown in Figure 6.3, to record the example file used in the training phase. While running the experiment, we observed different outcomes of the simulations with identical inputs. Running multiple simulations with the same behaviour tree controlling the follower resulted

in slightly different pathing, and therefore different fitness values. When simulating the same behaviour tree that was used for recording the example data, the fitness on Scenario 1 varied between 0 and 40, and on Scenario 2 the fitness varied between 0 and 9.

We initially suspected that the simulation engine had internal inconsistencies for when received unit tasks were executed, and that this was caused by the internal path planning of the simulation engine taking different lengths of time to complete depending on the available processing power. We observed that the simulation was not paused while the pathing was calculated, and that the units therefore started to move at different frames in the simulation. However, after investigating these issues further, we discovered that path planning was not the cause of the issues with stochastic evaluation outcomes. Section 7.2 covers the identified issues and how they were addressed.

The consequence of the issues with stochastic simulation outcomes was that we had to account for stochastic evaluation of chromosomes, where a chromosome can be evaluated better or worse based on luck. A way of managing stochastic problems with GAs is to simulate each chromosome a large number of times, combining the results [45]. However, as it took approximately 18 seconds to simulate each behaviour tree, running a large number of simulations per chromosome was not an option. The inconsistencies in fitness evaluation can affect the selection process of NSGA-II negatively. We suspect that this issue contributed to the stagnation of fitness development on Scenario 1, shown in Figure 6.4a. The were no improvements to the best fitness on both scenarios after the 20$^{\text{th}}$ epoch. The small population size used for this experiment is also a potentially contributing factor to the stagnation.

Theoretically, the system should be able to find behaviour trees that result in 0 fitness on both scenarios. We suspect that the stochastic simulation outcomes during this experiment significantly limited the performance of the algorithm by causing it to keep trees that were lucky during evaluation over trees with statistically better performance that were unlucky during evaluation, and by increasing bloat. We observed that a significant portion of the population consisted of bloated trees that had been more lucky during evaluation than the functionally identical non-bloated versions. These trees typically had redundant condition nodes, with no effect on the represented behaviour. Normally, these bloated trees would be dominated by the smaller functionally identical trees as they would receive the same fitness on the scenarios and a worse fitness on size. However, due to the stochastic evaluations, the bloated trees now had a chance to receive a better fitness on the scenarios.

Despite the issues with behaviour tree evaluation, when comparing the two selected resulting behaviour trees with the manually made behaviour tree used for recording the example, we can see that there are significant similarities. For the tree in Figure 6.6a, it has managed to represent an approximation of what we consider the most important part of the example behaviour – moving only when more than 30 meters away from the target. In both example scenarios, checking for distance is more important than checking whether the target is approaching. This is because the target usually moves away from the follower, and is standing still for a significant portion of the scenarios.

The bigger resulting tree, shown in Figure 6.6b, includes most of the behaviour of the manually made example tree. Before moving to the target, both distance and movement angle is checked with approximately the same distance and angle used in the example tree. However, due to the sequence of distance checks and wait node, the follower might move closer while it is between 29.77 and 26.19 meters away from the target. Then again, as the target is moving, the wait node between the distance checks will often cause the target to be further than 29.77 meters away for the next tick.

Combined with the long time it takes to simulate a behaviour tree, finding optimal models could take a very long time, even with simple experiments. However, the resulting behaviour trees reproduce the most essential parts of the behaviour used to record the example data, which shows that the system is able to replicate simple behaviour by generating and evolving behaviour trees.

7.1.2 Evaluation of Experiment 2

Experiment 2 was conducted after the issues identified during Experiment 1 had been addressed. The goal of the experiment was to see if we could improve on the results from Experiment 1 and to do a new evaluation of the system.

In Experiment 2, the system was able to generate behaviour trees that perform close to identical to the observed behaviour. It is important to note that in Experiment 2, the best fitness score on Scenario 1 and 2 both belong to the same behaviour tree. This tree is shown in Figure 6.9, and had a fitness score of 0.012 on Scenario 1 and 0.03 on Scenario 2. This tree is structurally identical to the tree used to record the example behaviour. The only difference is small variations in the variables of the variable condition nodes: 21.11° versus 20° for *Is approaching* and 30.22m versus 30m for *Is within*. The system was able to generate a behaviour tree with a fitness score approximating 0 on both scenarios, indicating that it is able to create generalised behaviour models based on multiple scenarios. In Experiment 1, the best behaviour tree for Scenario 1 is not the same as the best tree for Scenario 2. This is a significant improvement from Experiment 1 to Experiment 2.

During Experiment 2, we observed that the generated behaviour trees contained few or no ineffective subtrees, and we observed that the algorithm actively removed the present ineffective subtrees, reducing bloat. While we have not done a thorough analysis, our evaluation is that the applied bloat-control works well after the improvements to simulation determinism had been implemented. A downside with the applied fixes to increase simulation determination is increased simulation time. The process of simulating a single chromosome on the two scenarios took approximately 32 seconds during Experiment 2 – almost 80% more than that of Experiment 1. However, with the observed reduction in bloat, we consider the more deterministic evaluation to be worth the increased simulation time.

While the average fitness on Scenario 1 and Scenario 2 is increasing from approximately epoch 90, the average size of the behaviour trees in the population is decreasing. A possible explanation is that this is due to NSGA-II's multi-objective selection, which is also set to minimise the number of nodes in the trees. Overall,

the system was continuously improving the average and best fitness on one or more fitness objective.

For the first 30 epochs, the Scenario 2 fitness development, shown in Figures 6.7 and 6.8, is close to the Scenario 2 fitness development in Experiment 1. A comparison of the fitness development between Experiment 1 and 2 is shown in Tables 6.2 and 6.3. At 30 epochs, the average fitness for Scenario 2 is ~26% better in Experiment 2 than in Experiment 1, but for the the best fitness, Experiment 2 is ~30% worse than Experiment 1. Although the relative difference in best fitness is ~30%, the actual difference in fitness value is only ~0.3, which is not that significant. For Scenario 1, the best fitness achieved in Experiment 2 after 30 epochs is significantly better than the best fitness achieved in Experiment 1 after 30 epochs. An improvement of ~83% with an actual fitness value difference of ~70. With the fitness function used in this experiment, we consider this to be a significant improvement.

From epoch 30 to epoch 60, the development on average and best fitness has mostly stagnated on both scenarios in Experiment 1. In Experiment 2, however, the best fitness on Scenario 1 and average and best fitness on Scenario 2 are continuously improving. After 60 epochs, the best fitness on Scenario 1 is ~99% better in Experiment 2 than in Experiment 1, with a fitness of ~84 in Experiment 1 and a fitness of 1 in Experiment 2.

At 90 epochs, there has been little to no improvement on both scenarios in Experiment 1. This is also true for Scenario 2 in Experiment 2, but for Scenario 1, the best fitness in Experiment 2 has improved from 0.66 at the 60th epoch to 0.03 at the 90th epoch. A fitness of 0.03 means the behaviour is approximately identical to the example behaviour. In addition, when letting Experiment 2 run for 148 epochs, the best fitness on Scenario 1 further improved to 0.012.

There are several significant factors that could have caused Experiment 2 to have better results than Experiment 1: a larger population size, a longer running time, more evaluations performed, and a more deterministic evaluation of behaviour trees. After 90 epochs with a population of 30, the system has done 2700 evaluations, which would be equivalent to that of 270 epochs with a population of 10. The system might be able to produce just as good results with a population of 10, given the same number of evaluations. Therefore, we can not exclude the possibility that Experiment 1 could have produced as good or better results as Experiment 2 given the same number of evaluations or time. However, when comparing the results after 90 epochs in Experiment 1 and 30 epochs in Experiment 2 (where the system has done the same amount of evaluations), Experiment 2 is still considerably better at Scenario 1, and they are approximately even on Scenario 2. In any case, the goal of the conducted experiments is not to analyse the effects of these factors, but to investigate if the system and applied GP method is feasible for imitating observed behaviour in complex environments. The conclusion from these experiments is that the system is able to imitate simple observed behaviour, and that the applied bloat-control seems to work well in the improved system.

7.2 Evaluation of the System

Experiment 1 and Experiment 2 have shown that the system is able to produce behaviour models that imitate simple behaviour. It is able to connect to complex simulation systems over HLA, and to extract data and control entities in order to evaluate the generated behaviour models. The system was designed to be usable with different types of experiments, with different simulated entities, different types of data, different types of activities. The system is also designed to be used with multiple GAs, but has only been tested with an implementation of NSGA-II. The system can in theory also be used for both observational and experiential learning, but has only been tested for observational learning.

Experiment 1 identified issues with how the system handled data extraction and time management in the HLA federation. When conducting experiment 1, the system used a lookahead value (see Subsection 2.4.1) of 1. This was done to enable the simulation engine to calculate new world states in parallel with our systems data processing and calculation of entity instructions. We suspect the issues were due to the parallel time advancement, where the updates from the simulation engine would be received by our system at different steps of its current internal logical time, for different runs of the simulation. By setting the lookahead of our federate to 0, timestamping all outgoing entity instructions, and introducing an internal tick rate for how long our system should advance in time, the determinism of behaviour tree evaluations was greatly improved. The same tree will now get the exact same score or a score with a minimal difference when evaluated in the same scenario multiple times. While this seems to have improved bloat-control, the evaluation is still not completely deterministic, which may cause GAs to converge to bad solutions [45]. A common way of handling stochastic evaluation outcomes is to run evaluations a large number of times and then using those results to calculate a final evaluation score [45]. However, as complex simulations are computationally heavy, applying this method would probably result in inefficient use of training time.

When the system sends the first interaction after connecting to a new HLA federation, it will sometimes crash. This is most likely caused by improper handling of timestamped interactions in HlaLib, the library used for HLA in our system. This is only an issue with the first interaction.

The system has only been tested with VR-Forces from MÄK. For controlling simulated entities in the simulation engine, the system sends instructions with the use of LLBML. For VR-Forces to be able to process LLBML messages, it is required to install the LLBML plugin in VR-Forces. This means that our system is not fully compatible with other simulation systems using HLA. The system is designed to use RPR FOM 2.0, a specification of data and data formats to exchange in the HLA federation, and it is required that simulation systems are compatible with this FOM.

The system has only been used with scenarios containing maximum five simulated entities. It is therefore unknown how the systems would handle scenarios with a large number of simulated entities. However, the operations done by our system are minimal compared to the calculations done in the simulation engine, and

we can not foresee any immediate problems with using it in large scale exercises.

Overall, the system is able to perform the tasks that were necessary to conduct the experiments required for answering RQ1. It also meets the requirements that were specified as part of RQ2: that it should use HLA for communication with an external simulation system, that it should be able to extract data from the external simulation system and then use the simulation system for evaluating generated behaviour models, and that it should support running different experiments with different with different scenarios, different simulated entities with different possible actions, and different types of data. It is also compatible with running more complex experiments than the ones conducted as part of this thesis.

7.3 Research Questions Revisited

In this section, the research questions are revisited, and we present how they are answered through the conducted research.

RQ1: *How do behaviour trees generated with GP perform in imitating observed behaviour in complex, realistic environments?*

The goal of this thesis is to investigate whether the CGF behaviour modelling process can be automated by using GP to evolve behaviour trees from observations of example behaviour. CGFs are used in military simulations, with highly complex and realistic simulated environments. Therefore, in order to answer the hypothesis, we need to research the feasibility of using GP for generating behaviour trees through observational learning in complex, realistic environments. To our knowledge, this has not been answered in current literature.

The results from Experiment 1 and Experiment 2 showed that behaviour trees generated with GP was able to successfully imitate the observed behaviour. We used NSGA-II, a MOGA, for our experiments. The example behaviour was recorded in a complex, realistic simulated environment, which was used to evaluate the generated solutions during training as well. In Experiment 1, the generated behaviour models were not identical to the observed behaviour, but contained the most essential aspects of the behaviour. After improvements were made to the system used for generating behaviour models, the results from Experiment 2 show that we are able to generate behaviour models that are approximately identical to the model used to control observed behaviour. The example behaviour used for these experiments was, however, very simple. How GP and behaviour trees perform in imitating more complex behaviour is still unclear. RQ1 has therefore been only partially answered, with the conclusion that behaviour trees generated with GP are able to successfully imitate *simple* behaviour in complex, realistic environments.

One of the issues with using GAs for training complex behaviour is that it can become difficult to define how the behaviour should be evaluated. When training complex behaviour in scenarios with a lot of affecting factors, it might be difficult to identify relevant data and to define the fitness function(s). A way of addressing this

issue is by learning pieces of the behaviour separately, and then combining the behaviour parts either manually or with machine learning. The modular hierarchical structure of behaviour trees is suitable for this approach.

RQ2: *How should a system for generating behaviour trees with GP be designed to be used with an external simulation system?*

In order to answer RQ1, we needed to develop a system that would enable us to run experiments with using GP to generate behaviour trees based on observed behaviour in complex simulations. The system had to satisfy three requirements, shown in Table 1.1. FFI requested that the system should use HLA for communication with the simulation system, forming the first requirement. They also required that it should be able to extract data from the simulation system and then use the simulation system for evaluating generated behaviour models, forming the second requirement. The system was intended to be used for multiple experiments, resulting in the third requirement.

During the literature review, we found no existing system or solution that would meet these requirements. Some of the reviewed publications discussed issues such as complex simulations being time-consuming, but provided no conclusions that could be used as basis for the design. The system created as part of this thesis explores a possible design solution. While our research does not provide a definitive answer to how such a system should be designed, it illustrates a working design that can be used for different exercises, simulated entities, actions, and data.

The design allows the system to be connected to different simulation systems over HLA. When using HLA, it is important to design the system for use with HLA time management in order to achieve deterministic solution evaluation, as discussed in Section 7.2. The system should be designed to be a time regulating federate, for being able to restrict how far the simulation engine is able to advance in time while the system is calculating new entity instructions, and to be a time constrained federate, for being able to receive timestamped messages from the simulation engine. When choosing what lookahead to use, it is important to remember that the lookahead will be the minimal delay between the timestamp of incoming data to the timestamp of the outgoing instruction calculated from that data, effectively setting a minimal response time for the agent. Our experience is that using a lookahead of 0 makes it easier to handle time management properly, at the cost of increased simulation time.

The system uses a HLA module (LLBML) to control entities in the simulation engine. The controlled entities are created and owned by the simulation engine. An alternative approach is to create the simulated entities internally, and then calculate their new states and send updates on where and what the entities are doing to the simulation engine during simulation. This will, however, require complex calculations in the local system. With our system, we have shown that the approach of controlling the simulated entities remotely via a module like LLBML works. By using strict federation time management, it should be possible to guarantee that the instructions are executed at the right time. However, this approach may

require extra software specifically designed to work with a simulation system, such as the LLBML plugin we have used with MÄK VR-Forces. In addition, many simulation systems do not support plugins at all, and this approach therefore limits compatibility with a lot of simulation systems.

Complex simulations are able to simulate a large variety of different entities. These entities can e.g. have different types of possible actions. When creating a system for use with complex simulations with unknown scenarios and entities, we consider it important to provide a framework for specifying what data that should be extracted, how it should be processed and to create and send different types of instructions to the simulated entities, etc. Learning different behaviour may often require different tuning of algorithm-specific variables and genetic operators, and we recommend that the design allows these settings to be accessible to the user. To summarise, modularity should be prioritised.

The system designed as part of this thesis shows how a system for generating behaviour trees with GP can be designed to be used with an external simulation system, and satisfies the requirements specified as part of RQ2. The design satisfies the first requirement, as all communication with the simulation system is done over HLA. We have shown that this design works for training behaviour models, however, there are still some minor issues with time management that need to be addressed. It satisfies the second requirement as it uses the simulation system for both data extraction and solution evaluation. Finally, the modular, data agnostic framework design makes the system compatible with different experiments with different scenarios, different simulated entities with different possible actions, and different types of data, satisfying the third requirement. The implementation described in this paper has been proven to be successful, and works as a possible design for answering RQ2.

7.4 Evaluation of the Contributions

In this section, the contributions are evaluated. The contributions provided in this thesis are significant enough to have been published at CogSIMA 2018, an international peer-reviewed conference, where we received positive response on the work that has been done. C1 and C3 have been further improved after the first publication.

C1: *Proof that GP and behaviour trees can be used to mimic recorded, simple behaviour in complex simulations.*

The results from our experiments show that GP and behaviour trees can be used to successfully mimic simple behaviour in a complex simulated environment. While we can not guarantee that our solution will work for more complex behaviour, the results are promising. In current literature, there is limited research on using GP to evolve behaviour trees with observational learning, and we found no work where it

was applied in complex simulated environments. The results from our experiments work as a proof of concept, and motivates further research on the subject.

The experiments discussed in this thesis are two single runs with a probabilistic algorithm. Therefore, while the results are promising, they do not guarantee that our method will be able to consistently imitate behaviour models. This could have been investigated by running the experiments a large number of times, but was not possible within the timespan of this project. However, the results of the experiment represent the general trend in training performance we have observed during testing of the system.

While we have shown that the technique can work on simple behaviour, there are limitations to using it for more complex behaviour. The long simulation times required for evaluating chromosomes drastically limits the speed of the training, even with the simple experiments described in this thesis. In addition, when using a MOGA for learning the behaviour, it requires fitness functions that effectively measure the different aspects of the desired behaviour. Defining these fitness functions may be a significant challenge when working with complex behaviour, and it should be investigated how this process can be made easier.

C2: *A proposed set of methods for mutating behaviour trees.*

In this thesis, we propose a set of methods for performing mutations on behaviour trees. A mutation is a genetic operation which alters an existing candidate solution. To our knowledge, four of the seven methods for mutating behaviour trees described in this thesis have not been discussed in current research. Although several authors have used mutations for evolving behaviour trees, they typically use one or two mutations [5, 4]. In our research, we propose multiple independent behaviour tree mutations, as well as a technique for combining the use of different mutations when evolving behaviour trees. With inspiration from SA [16, p. 128], this technique uses scaling probability weights to select which mutation method should be used, favouring mutations that make smaller changes as time passes.

With the ability to scale the selection weights of the different mutations independently, we are able to encourage use of drastic mutations early in the training process without limiting the localised search as the algorithm converges. Drastic mutations help the algorithm from getting stuck in local minima, while smaller mutations are important during the fine tuning of the generated behaviour trees. We propose to use a set of different mutations with varying effects on the behaviour trees, and to assign scaling selection weights based on how drastic the changes they make to the behaviour trees are.

The proposed mutations and technique for selecting between them has been successfully applied in evolving behaviour trees that imitate simple behaviour. In Experiment 2, we were able to generate behaviour trees that are close to identical to the example behaviour by using the proposed mutation method with a MOGA. While we have shown that the combined use of the mutations works, we have not analysed how the individual mutations affect the training performance. We have also not done any detailed analysis on how the weight and factors used for selecting between the mutations should be tuned.

C3: *A modular system for using GP to evolve behaviour trees through observational learning with a complex, realistic simulation system over HLA.*

The system can be used for running different experiments, with different types of data, different types of simulated entities, different algorithms and potentially with different simulation systems. The system has been primarily designed for observational learning, but should also be able to support experiential learning due to its modular structure. The system is intended to be used for further research on automating behaviour modelling for CGF as part of a larger project at FFI. The system is published as open source at `https://github.com/eivinmor/msc-gbh-em` under the MIT license[1].

The results from Experiment 2 show that the system is able to provide a framework that enables the applied GA to generate behaviour trees that successfully imitate the observed behaviour. In current literature, there are proposed frameworks for using observational learning in complex environments. However, the ones we have seen are for use with other learning techniques, including CBR, decision trees and ANN, or have not been designed to work with HLA. As far as we know, our system is the first to support evolving behaviour trees with GP through observational learning with complex simulations over HLA.

[1]`https://opensource.org/licenses/MIT`

Chapter 8

Conclusion and Future Work

In this chapter, we present the conclusion of the thesis and provide suggestions for future work.

8.1 Conclusion

The hypothesis underlying this thesis was stated as *the process of creating behaviour models for CGFs can be automated by replacing manual behaviour analysis and programming with GP that generates behaviour trees from observing examples.* The hypothesis was divided into RQ1 and RQ2. To evaluate the hypothesis, we had to answer RQ1: *How do behaviour trees generated with GP perform in imitating observed behaviour in complex, realistic environments?* In order to answer RQ1, we first had to develop a system that would enable us to run experiments with using GPs to generate behaviour trees based on observed behaviour in complex simulation. This led to the creation of RQ2: *How should a system for generating behaviour trees with GP be designed to be used with an external simulation system?* The implemented system is one of the contributions of this thesis, C3. C3 supports RQ2 by working as a proposed design that has been proven successful in the conducted experiments. C3 is also intended to be used for further research, as part of a larger project on DDBM at FFI.

As part of the developed system, we created a set of seven behaviour tree mutations. The proposed set of mutations and the probabilistic SA-inspired method for choosing which mutation to use forms the contribution C2. C2 has been used in the experiments conducted to answer RQ1.

We conducted two experiments, where we were able to mimic simple behaviour in complex simulations by evolving behaviour trees with GP. The proof that GP and behaviour trees can be used to mimic recorded, simple behaviour in complex simulations forms C1, which supports RQ1. Based on C1, RQ1 has been partially answered, with the conclusion that behaviour trees generated with GP are able to successfully imitate *simple* behaviour in complex, realistic environments.

In conclusion, we have shown that the hypothesis is true when working with

simple behaviour. However, more work is required to evaluate how the proposed method performs with more complex behaviour. The issues related to defining good fitness functions for observational learning of complex behaviour with MOGA, and the long simulation times required to evaluate behaviour models in complex simulations are important limitations. Even for experiments with simple behaviour, the time required to simulate a single chromosome significantly limits the speed at which you are able to train behaviour models. A solution proposed in current literature is to distribute the load over a large number of computers.

In any case, the results of this thesis work as a proof of concept, and motivate further research on automating CGF behaviour modelling by using GP to evolve behaviour trees in complex environments. The contributions of this thesis add knowledge to the literature, and provides a system and mutation methods that can assist in further research on the topic.

8.2 Future Work

Following are suggestions for future work.

Reduce simulation time A significant limitation with using GP to evolve behaviour trees in complex simulations is the long time it takes to simulate, and therefore evaluate, the behaviour trees. For using the proposed method for learning complex behaviour, reducing the simulation time is important. We suggest looking at ways of distributing the simulation over multiple computers, either by simulating the behaviour trees in parallel or by distributing the computational load of a single simulation.

Defining fitness functions Defining MOGAs fitness functions that are able to capture the important aspects of complex behaviour is difficult. It should be investigated if there are any helpful techniques or alternative methods, e.g. learning the behaviour in chunks and then combining it by hand or with machine learning, that can make the process of defining good fitness functions easier.

Evaluate method with more complex behaviour In this thesis, we were only able to partially answer the hypothesis, by showing that the proposed method works with simple behaviour. Creating realistic CGFs will often require modelling of complex behaviour, and the techniques used for automating the modelling process should therefore be able to work with complex behaviour as well. In order to fully answer the hypothesis underlying this thesis, it should be evaluated whether GP and behaviour trees are feasible for learning complex behaviour in complex simulations.

Human behaviour Realistic CGF behaviour is important for creating realistic simulated environments for training and evaluating battle plans. Being able to create

unpredictable behaviour models with personal human behavioural traits is therefore important. Evolving GP with behaviour models has been shown successful at recording personal traits in driving behaviour [13], but not in realistic simulations. We suggest further investigation on how the method investigated in this thesis performs in capturing human behavioural traits. Demonstrating the desired behaviour can be done with e.g. VR-Engaged.

Perceived truth simulation data For generating more realistic behaviour models, the trained agent should only have access to the perceived simulation data, in the same way a human would have in a real-life situation or in a simulated exercise. For future experiments with the proposed method, we suggest using only the simulation data that is perceived by the controlled simulated entity, in order to create more human-like behaviour.

Heuristics for randomising node variables In the proposed mutation for randomising node variables, the variables are currently randomised between a minimum and maximum boundary set in the respective node. This can make it difficult to finely tune the variables, which may be necessary for creating good behaviour models. Introducing a heuristically guided search, e.g. SA, could improve the convergence of the algorithm used with this mutation.

Further testing of proposed mutations The proposed set of mutations has not been thoroughly tested. While we have shown that we are able to reach good results when using the proposed mutations, it still remains to investigate how each mutation, and combinations of the mutations, affect the training performance. It could also be interesting to investigate the effects of using different weights and scaling factors.

Compare the computer generated behaviour to manually created behaviour Realistic CGF behaviour is important for creating realistic simulated environments for training and evaluating battle plans. It should therefore be investigated how well the models generated by evolving behaviour trees with GP through observational learning in complex simulations are able to imitate the observed behaviour compared to behaviour models that have been manually created by domain experts.

Bibliography

[1] S. Bruvoll, J. E. Hannay, G. K. Svendsen, M. L. Asprusten, K. M. Fauske, V. Kvernelv, R. A. Løvlid, and J. I. Hyndøy, "Simulation-supported wargaming for analysis of plans," in *NATO Modelling and Simulation Group Symphosium. M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (MSG-133)*, 2015.

[2] M. R. Endsley, "Toward a theory of situation awareness in dynamic systems," *Human factors*, vol. 37, no. 1, pp. 32–64, 1995.

[3] M. R. Endsley and D. J. Garland, "Pilot situation awareness training in general aviation," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 44, no. 11. SAGE Publications Sage CA: Los Angeles, CA, 2000, pp. 357–360.

[4] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *arXiv preprint arXiv:1504.05811*, 2015.

[5] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game defcon," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 100–110.

[6] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving behaviour trees for the mario ai competition using grammatical evolution," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2011, pp. 123–132.

[7] G. Robertson and I. Watson, "Building behavior trees from observations in real-time strategy games," in *Innovations in Intelligent SysTems and Applications (INISTA), 2015 International Symposium on*. IEEE, 2015, pp. 1–7.

[8] A. Toubman, G. Poppinga, J. J. Roessingh, M. Hou, L. Luotsinen, R. A. Løvlid, C. Meyer, R. Rijken, and M. Turčaník, "Modeling cgf behavior with machine learning techniques: Requirements and future directions," in *Proceedings of the 2015 Interservice/Industry Training, Simulation, and Education Conference*, 2015, pp. 2637–2647.

[9] J. J. Roessingh, A. Toubman, J. van Oijen, G. Poppinga, M. Hou, L. Luotsinen *et al.*, "Machine learning techniques for autonomous agents in military simu-

lations—multum in parvo," in *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 3445–3450.

[10] L. J. Luotsinen, F. Kamrani, P. Hammar, M. Jändel, and R. A. Løvlid, "Evolved creative intelligence for computer generated forces," in *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 003 063–003 070.

[11] R. A. Løvlid, L. J. Luotsinen, and F. Kamrani, "Data-driven behavior modeling for computer generated forces," Norwegian Defence Research Establishment (FFI), Tech. Rep., 2017.

[12] A. J. Gonzalez, R. F. DeMara, and M. Georgiopoulos, "Vehicle model generation and optimization for embedded simulation," *Proceedings of the 1998 Spring Simulation Interoperability Workshop (SIW'98)*, 1998.

[13] H. K. G. Fernlund, "Evolving models from observed human performance," Ph.D. dissertation, University of Central Florida, 2004.

[14] H. K. Fernlund, A. J. Gonzalez, M. Georgiopoulos, and R. F. DeMara, "Learning tactical human behavior through observation of human performance," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 36, no. 1, pp. 128–140, 2006.

[15] B. J. Oates, *Researching information systems and computing*. Sage, 2006.

[16] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Pearson Education, 2014.

[17] F. Kamrani, L. J. Luotsinen, and R. A. Løvlid, "Learning objective agent behavior using a data-driven modeling approach," in *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 002 175–002 181.

[18] T. Murata and H. Ishibuchi, "Moga: multi-objective genetic algorithms," in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 1, Nov 1995, pp. 289–.

[19] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992, vol. 1.

[20] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu.com, 2008.

[21] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, "The department of defense high level architecture," in *Proceedings of the 29th conference on Winter simulation*. IEEE Computer Society, 1997, pp. 142–149.

[22] R. M. Fujimoto, "Time management in the high level architecture," *Simulation*, vol. 71, no. 6, pp. 388–400, 1998.

[23] L. J. Luotsinen and R. A. Løvlid, "Data-driven behavior modeling for computer generated forces," in *NATO Modelling and Simulation Group Symp. M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (MSG-133)*, 2015, pp. 1–13.

[24] G. Stein and A. J. Gonzalez, "Building high-performing human-like tactical agents through observation and experience," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 41, no. 3, pp. 792–804, 2011.

[25] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[26] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of machine learning*. Springer, 2011, pp. 760–766.

[27] J. J. Roessingh, A. Toubman, J. van Oijen, G. Poppinga, R. A. Løvlid, M. Hou, and L. Luotsinen, "Machine learning techniqus for autonomous agents in military simulations - multum in parvo," in *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 3445–3450.

[28] M. W. Floyd and B. Esfandiari, "A case-based reasoning framework for developing agents using learning by observation," in *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*. IEEE, 2011, pp. 531–538.

[29] Q. Zhang, Q. Yin, and K. Xu, "Towards an integrated learning framework for behavior modeling of adaptive cgfs," in *Computational Intelligence and Design (ISCID), 2016 9th International Symposium on*, vol. 2. IEEE, 2016, pp. 7–12.

[30] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super mario evolution," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 156–161.

[31] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in *European Conference on Genetic Programming*. Springer, 2003, pp. 204–217.

[32] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler, "Multiobjective genetic programming: Reducing bloat using spea2," in *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, 2001, pp. 536–543.

[33] E. D. De Jong and J. B. Pollack, "Multi-objective methods for tree size control," *Genetic Programming and Evolvable Machines*, vol. 4, no. 3, pp. 211–233, 2003.

[34] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[35] E. D. De Jong, R. A. Watson, and J. B. Pollack, "Reducing bloat and promoting diversity using multi-objective methods," in *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2001, pp. 11–18.

[36] E. F. Crane and N. F. McPhee, "The effects of size and depth limits on tree based genetic programming," in *Genetic Programming Theory and Practice III*. Springer, 2006, pp. 223–240.

[37] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evolutionary Computation*, vol. 14, no. 3, pp. 309–344, 2006.

[38] A. Alstad, O. Mevassvik, M. Nielsen, R. Løvlid, H. Henderson, R. Jansen, and N. de Reus, "Low-level battle management language," in *Proceedings of the 2013 Spring Simulation Interoperability Workshop*, no. 13S-SIW-032, 2013.

[39] J. Ruiz, D. Désert, A. Hubervic, P. Guillou, R. Jansen, N. de Reus, H. Henderson, K. Fauske, and L. Olsson, "BML and MSDL for multi-level simulations," in *Proceedings of the 2013 Fall Simulation Interoperability Workshop*, no. 13F-SIW-002, 2013.

[40] NATO NSA, *STANAG 4603 - Modelling and Simulation Architecture Standards for Technical Interoperability: High Level Architecture (HLA)*, 2nd ed., 2015.

[41] Simulation Interoperability Standards Organization (SISO), *Standard for Guidance, Rationale, and Interoperability Modalities (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download&EntryId=30822, 2015, SISO-STD-001-2015.

[42] ——, *Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download&EntryId=30823, 2015, SISO-STD-001.1-2015.

[43] A. Alstad, R. A. Løvlid, S. Bruvoll, M. N. Nielsen, and O. M. Mevassvik, "Autonomous simulation of a battalion operation - seamless integration of command and control and simulation for planning and training," Norwegian Defence Research Establishment (FFI), FFI-rapport 2013/01547, 2013.

[44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[45] R. Al-Aomar, "Incorporating robustness into genetic algorithm search of stochastic simulation outputs," *Simulation Modelling Practice and Theory*, vol. 14, no. 3, pp. 201–223, 2006.

Glossary

AI artificial intelligence.

ANN artificial neural network.

CBR case based reasoning.

CGF computer generated force.

CogSIMA 2018 2018 Conference on Cognitive and Computational Aspects of Situation Management.

CSV comma-separated values.

DDBM data-driven behaviour modeling.

FFI Norwegian Defence Research Establishment.

FOI Totalförsvarets forskningsinstitut.

FOM Federation Object Model.

GA genetic algorithm.

GP genetic programming.

GUI graphical user interface.

HLA High Level Architecture.

LLBML Low Level Behaviour Markup Language.

MOEA multi-objective evolutionary algorithm.

MOGA multi objective genetic algorithm.

NSGA-II Non Sorting Genetic Algorithm II.

RPR FOM Real-time Platform-level Reference Federation Object Model.

RTI Run-Time Infrastructure.

SA simulated annealing.

SVG Scalable Vector Graphics.

VBS3 Virtual Battlespace 3.

Appendices

Appendix A

CogSIMA2018 Article and Poster

This appendix includes the poster article that was published at the CogSIMA2018 conference. It also includes the poster that was presented at the conference.

Automating Behaviour Tree Generation for Simulating Troop Movements (Poster)

Gabriel Berthling-Hansen*, Eivind Morch*, Rikke Amilde Løvlid†, and Odd Erik Gundersen*‡
*Norwegian University of Science and Technology (NTNU), Trondheim, Norway
†Norwegian Defence Research Establishment (FFI), Kjeller, Norway
‡Corresponding author: odderik@ntnu.no

***Abstract*—Computer generated forces are simulated units that are used in simulation based training and decision support in the military. These simulations are used to help trainees build a mental model of how different scenarios could play out, and thus give them a better situation awareness when conducting operations in real life. The behaviour of these simulated units should be as realistic as possible, so that the lessons learned while simulating are applicable in real situations. However, it is time consuming and difficult to build behaviour models manually. Instead, we explore the possibility of applying machine learning to generate behaviour models from a set of examples. In this paper we present the results of our preliminary experiments on using machine learning for behaviour modelling. We implement a follow behaviour by using behaviour trees that are evolved using genetic algorithms. The fitness of the evolved behaviour trees have been evaluated by comparing them with a manually generated behaviour tree that implements the behaviour properly. The genetic algorithm converges to a tree that is very similar to the manually generated behaviour tree, suggesting that the method works. Further work is necessary to test whether this approach will work on more complex behaviours.**

***Index Terms*—Behaviour tree, simulation, genetic algorithms**

I. Introduction

Computer generated forces (CGFs) are autonomous or semi-autonomous entities that represent military units, such as tanks, soldiers and combat aircrafts, in simulation software for military operations. CGFs are similar to non-player characters in computer games and are used in military simulation-based training and decision support applications. CGFs enable simulating large military operations as one operator is able to control several military units. The behaviour of the CGFs, e.g. how they move, where they look, when they shoot etc., should represent the behaviour of corresponding human soldiers or manned systems as accurately as possible. Ideally, a soldier training with a virtual simulator should not notice whether his teammates or opponents are human controlled entities or CGFs. Realistic CGF behaviour also makes it possible to simulate various plans or courses of actions to improve the situation awareness and get a good understanding of how a situation could play out [1]. Simulations can help build and train the mental model of the trainee by practising situation comprehension and projection, situation awareness level two and three in Endsley's model of situation awareness [2]. In aviation, around 20% of the errors are related to problems with the mental model according to Endsley and Garland [3]. Given that errors made by soldiers in a stress situation are similar to those made in aviation, improving their mental model is of high importance. This requires the CGFs to behave in a natural way, as their behaviour affects the situation comprehension and projection of the trainee.

There are several ways to represent the behaviour of CGFs. The most common way is to use state machines that describe different states that the CGFs can be in and the actions they can perform in every state. Lately, however, behaviour trees have grown very popular [4]. In any case, the behaviour models are typically made manually. This means that military experts have to tell programmers how they want CGFs to behave. This is a difficult and time consuming process [5], and we want to explore how to use machine learning to generate behaviour models from examples of desired behaviour.

One potential problem when building a behaviour model with machine learning is that the model often becomes opaque, meaning it becomes hard to interpret what the model has learned. In our work we decided to focus on trying to learn behaviour trees, i.e. the same type of model that can be used to model the behaviour manually. By using behaviour trees, the learned model for a CGF is explicit, which enables and simplifies explaining the behaviour. Core et al. [6] present a similar system with a separate module for explaining the CGF's behaviours. By contrast, they represent the behaviours as rules and not behaviour trees.

Using machine learning to generate behaviour models for CGFs has been discussed in the NATO Research Task Group IST-121 RTG-060 "Machine Learning Techniques for Autonomous Computer Generated Entities" [7]. The paper refers to different case studies performed by the participating nations. Worth mentioning is the work done by Totalförsvarets forskningsinstitut (FOI), who used machine learning to create autonomous agents that learn a tactical movement called bounding overwatch for dismounted infantry [8]. They divided the behaviour into different decision-models, like whether to move or not, where to aim and whether to stand or

kneel. These models where learned separately and combined manually. This research was done with Virtual Battle Space 3 (VBS3), a game based military simulation system [9].

In this paper we present our preliminary work on using machine learning to generate a behaviour model for CGFs. The work has been done with a real, military simulation system called VR-forces from MÄK [10]. We have used a genetic algorithm (GA) to generate a model for a soldier who follows another soldier. The next section includes necessary background information on genetic algorithms and behaviour trees. Section III describes the simulation system architecture and how data generation and the actual training are performed. An experiment and results are presented in section IV, followed by a discussion of the results and related work in section V. Finally, conclusion and suggestions for future work are included in VI.

II. Background

A. Genetic Algorithms

GAs are stochastic search algorithms inspired by evolution. A GA generates and evolves a population of chromosomes, where each chromosome is a candidate solution for solving a problem. Chromosomes are assigned a value representing how well they solve the problem, called fitness. In each epoch, the GA produces a new generation of chromosomes through crossover and selection. Crossover is the creation of a new chromosome by combining the traits of two existing chromosomes to produce a hybrid solution. Selection is the process of deciding which chromosomes should be used for crossovers, and which should be potentially included as they are in the new generation, which is usually done by comparing fitness values. The chromosomes are also randomly mutated, making direct changes to existing solutions. See [11] for more information on GAs.

B. Behaviour Trees

Behaviour trees are trees of hierarchical nodes that control decision making and task execution, and have been popularly used for modelling the behaviour of computer-controlled units in video games [4]. They provide a scalable and modular solution for representing complex behaviour without the exponential scalability of Finite State Machines (FSM) and reusability-problem of Hierarchical Finite State Machines (HSFM) [12]. Behaviour trees are also human-readable, giving the opportunity for visual analysis of the represented behaviour.

Behaviour tree nodes can return one of three statuses: *running*, *success* or *failure*. Running means that the node is currently active, has not completed its tasks and needs more time to finish. Success is returned when a node is finished executing and its task was successfully completed, and failure is returned when the task finished unsuccessfully.

Behaviour trees are traversed from the root node and down. If all visited nodes are finished, the tree will be traversed from the root and down again on the next timestep. However, if one of the nodes return running, the tree will keep running that node every timestep until it returns either failure or success. Once the node is done, the tree will continue traversing from the position of the node.

1) Composite Nodes: A composite node is used to group nodes into a higher level task [12]. The type of the composite node dictates in which order it will execute children nodes, when to stop, and what status to return. The system described in this article uses two types of composite nodes, *sequence* and *selector*. A sequence node (displayed as $\rightarrow$) executes its children from left to right until one of the children returns failure or all return success. If a child returns failure, then the sequence will stop and return failure. If all its children return success, it will return success. A selector node (displayed as **?**) executes its children from left to right until one of the children returns success or all children return failure. If a child returns success, the selector will stop and return success. If all its children return failure, the selector will return failure.

2) Leaf Nodes: A leaf node has no children, and is either an action node or a condition node. An action node is used to perform a specific low-level action, e.g. move to a certain location. A condition node returns success or failure based on some condition, e.g. whether an object is within a specific distance or not.

3) Blackboard: A blackboard contains data that is accessible for all the nodes of the behaviour tree. Nodes may also alter data inside the blackboard. A blackboard is an important feature of a behaviour tree as it enables nodes to share and alter the same state representation, avoiding an exponential state complexity such as in FSMs.

III. Generating Behaviour Trees for CGF

A. System Architecture

For our experiments we used a real, military simulation system called VR-Forces from MÄK. The virtual terrain, physical simulation of entities etc. are simulated in this system. We made a separate system that can record data from VR-Forces, generate the behaviour models using machine learning and send commands to the entities in VR-Forces. Figure 1 shows an example of a virtual terrain in VR-Forces. Our system communicates with VR-Forces using high level architecture (HLA), a standard for distributed simulation that is commonly used in military simulation systems [13]. Other CGF systems that support HLA could be used in place of VR-Forces.

Systems that communicate over HLA must agree on data and data formats to exchange. This is formalised in a federation object model (FOM), and we have used the Real-time Platform-level Reference (RPR) FOM, which is a standard FOM that many military simulation systems support [14], [15]. However, this FOM does not include commands or the perceived truth of the CGF entities. For this we use an extra module for low level battle management language (LL-BML), which is made as an extension to the RPR FOM [16], [17]. Bruvoll et al. describe using a multiagent system to control a CGF system in a similar manner [1], [18].

Our system generates and evaluates behaviour models for different types of units in different scenarios. Different units, scenarios and objectives require different behaviour tree nodes

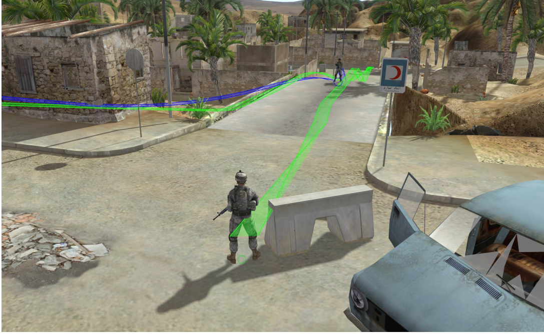


Fig. 1: VR-Forces simulation environment

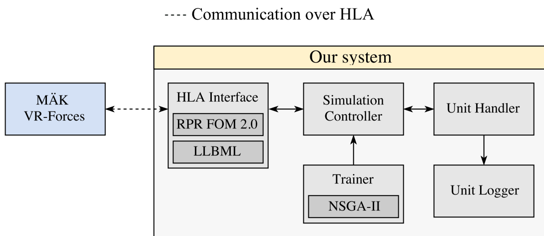


Fig. 2: Architecture Overview

with specialised tasks, different data collection and data processing, different performance evaluations (fitness), and different training algorithm tuning.

Figure 2 shows a high level overview of the system architecture. The *Simulation Controller* provides an abstraction level for the rest of the system to interact with VR-Forces, which it communicates with through the *HLA Interface*. These interactions include sending instructions to play, pause and load a scenario, as well as forwarding events of newly discovered units from VR-Forces to the *Unit Handler*. The Unit Handler handles the registration of simulation entities as local unit objects that can be used for logging data or giving commands. The Simulation Controller also instructs the Unit Handler to update unit data when the simulation time advances (tick). The *Unit Logger* writes data from the units registered by the Unit Handler to a database. The *Trainer* handles training of behaviour models. This includes deciding which scenario to simulate, initiating simulations, and creating, evaluating and modifying the behaviour trees. The main system has two modes—example recording and training.

1) Example Recording: The recording mode is used to record the behaviour of a unit that is controlled by an external source (human or script), in order to generate training data. This could be a person controlling a unit by joystick or mouse and keyboard through VR-Engage [19] while our system records data relevant to the performed task from the simulation. When recording, VR-Forces is initiated externally, and our system is only listening to and logging data updates,

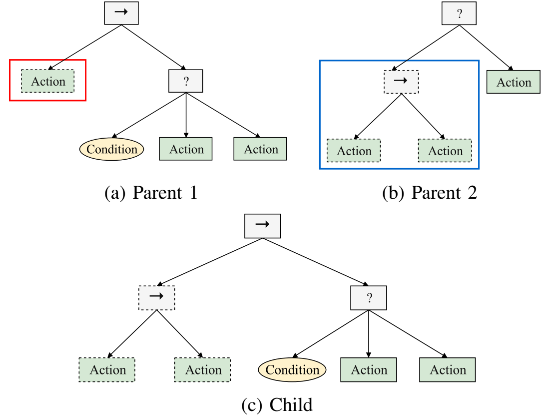


Fig. 3: Crossover

such as position, velocity and current engagement status. This data can later be used as examples of desired behaviour in the respective scenario.

2) Training: During training, the system uses a GA to generate, evaluate and improve behaviour trees. This is handled by the *Trainer*, shown in Figure 2, which takes a GA implementation as an argument upon initiation. NSGA-II [20], a multi-objective GA, is used for the experiments described in this paper. The *Trainer* also requires a list of scenarios and recorded example data, an object responsible for evaluating the fitness of behaviour trees, and an object responsible for collecting and holding the data to be used for evaluation.

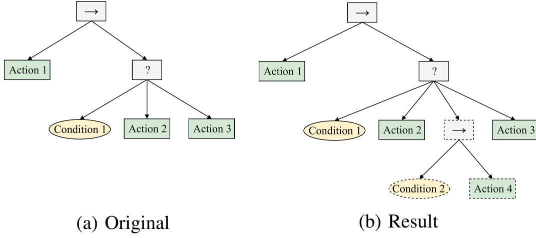
B. Evolving Behaviour Trees

The evolution of behaviour trees is done using NSGA-II, a multi-objective GA which sorts and selects chromosomes by non-domination [20]. A chromosome is said to dominate another chromosome if it has a better fitness value for one or more objectives and equal fitness value for the rest.

1) Crossover: The crossover operator chooses two behaviour trees as parents through tournament selection, and then chooses a random subtree in each of the parents, as illustrated in Figures 3a and 3b. A clone of parent 1 is then created and the subtree of parent 2 is inserted at the position of the subtree of parent 1. Figure 3c displays the final tree after crossover. This is the same crossover operator as is used in [4].

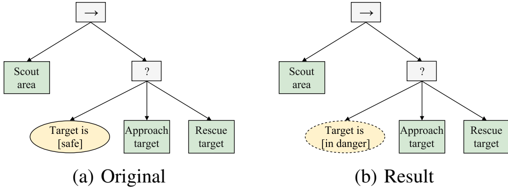
2) Mutations: The system uses six different mutations with varying level of impact on the behaviour trees. The probability of choosing one mutation over the others is decided by weights that are tuned per experiment. It is also possible to set weight factors for each mutation, altering the mutation weight depending on how many epochs the algorithm has been running. All mutations were designed by the authors of this paper for this specific system.

a) Add Random Subtree: This mutation generates a random subtree with a specified minimum and maximum number of nodes, and inserts it at a random position in the behaviour



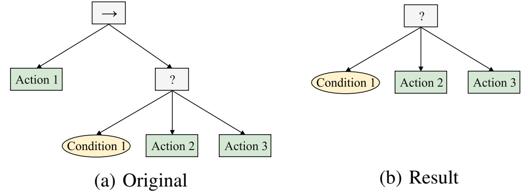
(a) Original (b) Result

Fig. 4: Add Random Subtree



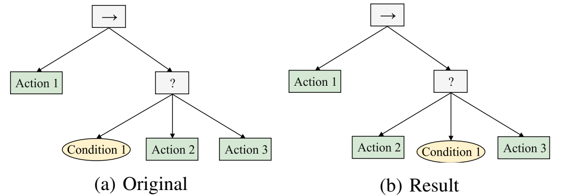
(a) Original (b) Result

Fig. 5: Randomise Variables of Random Node



(a) Original (b) Result

Fig. 6: Replace Tree With Subtree



(a) Original (b) Result

Fig. 7: Switch Positions of Random Sibling Nodes

tree. Figure 4 shows the insertion of a tree with three nodes, marked with a dotted line.

b) Randomise Variables of Random Node: Both action nodes and condition nodes can have variables that affect their functionality. This reduces the total number of nodes a developer has to make and also allows the system to fine-tune the behaviour of the behaviour tree. E.g., for a condition node that checks if the distance between two units is lower than a certain value, the value can be altered during training to check for different distances. This mutation randomises one or multiple variables in an action or condition node. In Figure 5 the value of the *"Target is ..."* condition node is changed from *safe* to *in danger*.

c) Remove Random Subtree: This mutation removes a random subtree from a behaviour tree.

d) Replace Random Node With Node of Same Type: This mutation replaces a random node with another random node of the same type. This means that a composite node can be replaced with another composite node (e.g. sequence to selector) or that a leaf node is replaced with another leaf node. Condition and action nodes are not treated differently, and may be replaced by any other leaf node.

e) Replace Tree With Subtree: This mutation replaces the entire tree with a random subtree of that tree. In Figure 6 the entire tree is replaced by the selector subtree.

f) Switch Positions of Random Sibling Nodes: This mutation switches the position of two random sibling nodes, including both leaf and composite nodes. In Figure 7 the *Condition 1* and *Action 2* nodes have switched places.

IV. Experiments and Results

A. Experiments

The experiment revolves around a wanderer and a follower. The objective for the experiment is to have the follower agent follow the wanderer. The wanderer is pre-programmed to follow a specific plan, which involves going to different locations and waiting a given amount of time at certain positions. The example file used to for training was recorded from a manually created behaviour tree, shown in Figure 11. This was an easy way to generate training data, and it also made it possible to compare the learned model with the "true" model.

A behaviour tree for a follower unit can have five different types of leaf nodes: *Move to target*, which makes the follower move toward its target for one tick. *Turn to target*, which makes the follower turn towards its target. *Wait*, which makes the follower stand still for a single tick. *Is within*, a variable condition node which checks whether the followers target is within a specified euclidean distance. *Is approaching*, a variable condition node which checks whether the angle between the movement vector of the target and the vector between the follower and the target is smaller than a specified threshold—effectively checking whether the followers target is moving towards the follower.

For this experiment, the training was done on two separate scenarios, using different terrains and wanderer paths. 2D overviews of the terrain and wanderer paths for both scenarios are shown in Figure 8. The first scenario simulates approximately 18 minutes of real-time over 1100 ticks, and the second approximately 12 minutes of real-time over 700 ticks. The training ran for 30 epochs with a population of 10 behaviour trees. The objective is to imitate the recorded behaviour. The fitness function therefore compares the behaviour produced by the behaviour tree with the recorded behaviour. The behaviour trees were evaluated by comparing the euclidean distance in each tick between follower and target position in the training data with the follower-target distance during behaviour tree simulation. All trees were tested on both scenarios. The equation for calculating the fitness value of a behaviour tree

(a) Scenario 1 (b) Scenario 2

Fig. 8: Scenario terrain and path overviews

for a single scenario is shown in Equation 1, where n is the number of ticks that were simulated.

$$Fitness = \frac{1}{n} \times \sum_{t=0}^{n} \Big(dist(example_t) - dist(btree_t)\Big)^2 \quad (1)$$

For each of the recorded ticks we find the follower-target distance from the training data and the simulated behaviour tree. The difference of these two distances is then squared so that a large difference over a few ticks is worse than a small difference over a large number of ticks. The squared differences in euclidean distance are summed and normalised over the number of ticks (n) to make the different scenario fitness values more comparable. The fitness values should be minimised. We chose this fitness function, as it is a simple formula that captures the similarity of the example and evaluated behaviour.

During training in this experiment, NSGA-II was set to simultaneously minimise three fitness values: the fitness value from running scenario 1, the fitness value from running scenario 2, and the number of nodes in the behaviour tree. By adding the tree size as a minimisation objective, we prevented the algorithm from creating bloated behaviour trees with unnecessary subtrees that have no significant effect on the behaviour.

All mutations were initially weighted the same, however with different weight factors, as described in Section III-B2. The mutations that change the behaviour trees drastically—*add random subtree* and *Replace tree with subtree*—were given a factor of less than 1, while *randomise variable of random variable node* was given a factor higher than 1. This way, the algorithm prioritises local search over larger changes at later epochs. The creation, crossover and mutation functions for behaviour trees were also restricted from producing behaviour trees with less than three and more than 12 nodes.

B. Results

Figure 9 shows the development of the fitness for the first scenario over 30 epochs. Figure 10a shows the fitness development over time for scenario 2, and Figure 10b shows a zoomed in view of the best-fitness development. The results show that the generated behaviour trees improve over time. For both scenarios, the trend is that the best and average score

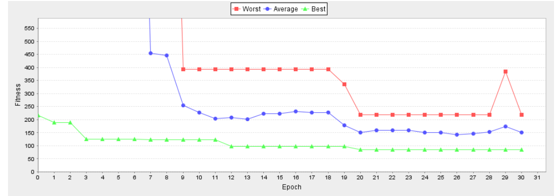
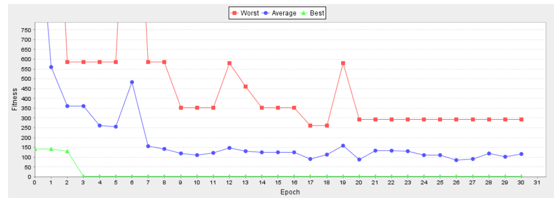
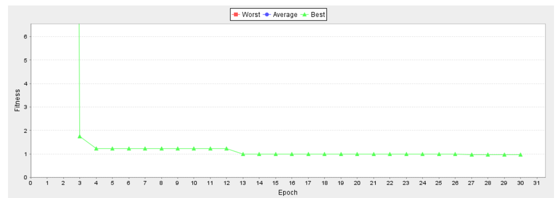


Fig. 9: Scenario 1 fitness development



(a) Complete



(b) Zoomed

Fig. 10: Scenario 2 fitness development

is continually decreasing as the algorithm is running. Short-term increases in average fitness values are due to the multi-objective selection of the NSGA-II algorithm.

We have chosen two of the non-dominated behaviour trees from the population at epoch 30, shown in Figure 12 with fitness values included in the sub figure captions. The behaviour tree in Figure 12a has the smallest possible size following the size restrictions, and has the lowest fitness on scenario 2 of all the trees of the same size. The larger one, shown in Figure 12b, performs better at scenario 1 and scenario 2, but has more than twice the number of nodes.

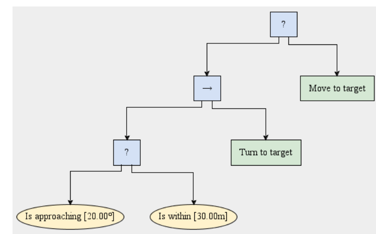
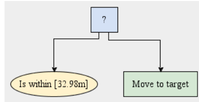
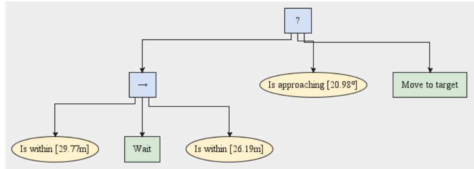


Fig. 11: Manually made behaviour tree



(a) Fitness: [size=3, scenario1=219, scenario2=114]



(b) Fitness: [size=7, scenario1=84, scenario2=5]

Fig. 12: Two of the resulting behaviour trees from epoch 30

V. Evaluation and Discussion

We used a manually created behaviour tree, shown in Figure 11, to record the example file used in the training phase. While running the experiment, we observed different outcomes of the simulations with identical inputs. E.g., running the multiple simulations with the same behaviour tree controlling the follower resulted in slightly different pathing, and therefore different fitness values. When simulating the same behaviour tree that was used for recording the example data, the fitness on scenario 1 varied between 0 and 40, and on scenario 2 the fitness varied between 0 and 9.

It appears that the simulation engine has internal inconsistencies on when received unit tasks are executed. We suspect this is caused by the internal path planning of the simulation engine taking different lengths of time to complete due to available processing power. It seems the simulation is not paused while the pathing is calculated, and so the units start to move at different ticks.

The consequence is that we have to account for stochastic evaluation of chromosomes, where a chromosome can be evaluated better or worse based on luck. A way of managing stochastic problems with GAs is to simulate each chromosome a large number of times, combining the results [21]. However, as it takes approximately 18 seconds to simulate each behaviour tree, running a large number of simulations per chromosome was not an option. The inconsistencies in fitness evaluation can also affect the selection process of NSGA-II negatively.

When comparing the two selected resulting behaviour trees with the manually made behaviour tree used for recording the example, we can see that there are significant similarities. For the tree in Figure 12a, it has managed to represent approximately what we consider the most important part of the example behaviour—moving only when more than 30 meters away from the target. In both example scenarios, checking for distance is more important than checking whether the target is approaching. This is because the target usually moves away from the follower, and that the target is standing still for a significant portion of the scenarios.

The bigger resulting tree, shown in Figure 12b, includes most of the behaviour of the manually made example tree. Before moving to the target, both distance and movement angle is checked with approximately the same distance and angle used in the example tree. However, due to the sequence of distance checks and wait node, the follower might move closer while it is between 29.77 and 26.19 meters away from the target. Then again, as the target is moving, the wait node between the distance checks will often cause the target to be further than 29.77 meters away for the next tick.

Theoretically, the system should be able to find behaviour trees that result in 0 fitness on both scenarios. However, we suspect that the stochastic simulation outcomes significantly limits the performance of the algorithm by causing it to keep trees that were lucky during evaluation over trees with statistically better performance that were unlucky during evaluation. Combined with the long time it takes to simulate a behaviour tree, finding optimal models will take a very long, even with simple experiments.

The bottleneck of the system is running simulations to evaluate the behaviour trees. Simulating a single behaviour tree on the two scenarios used for the previously described experiment takes approximately 18 seconds. This limits the population size we can use for NSGA-II, as evaluating a large number of chromosomes per epoch will be inefficient use of time, especially when multiple scenarios are used for evaluation.

Colledanchise et al. [4] and Lim et al. [12] also use GAs to generate behaviour trees, with the same crossover operator as we have used in our system. For mutation, however, Colledanchise et al. used a single mutation which replaces a single node in a behaviour tree with another node of the same type, while we use six different mutations that alter the behaviour tree in different ways. Another important difference is that Colledanchise et al. use reinforcement learning to generate behaviour trees designed to play Mario, while we generate behaviour trees that imitate example behaviour in complex simulation environments.

Lim et al. [12] use two mutations with similarities to two of our mutations—adding a random behaviour tree as a subtree, and changing the an inner variable of an existing node. They also had issues with long simulation times, which they handled by distributing their simulation over 20 computers, drastically speeding up each experiment. Lim et al. trains the behaviour using reinforcement learning while we use supervised learning.

The followers behaviour model is currently fed ground truth information about the target from the simulation system. This means that the follower always knows the current position of the target, even if the follower unit is unable to observe the targets position and velocity. For a more realistic experiment, the follower behaviour model should only have access to perceived truth, which is supported in VR-Forces through e.g. line of sight and radio communication. This would probably result in more human-like behaviour.

VI. Conclusion and Future Work

In this paper we used genetic algorithms to generate behaviour trees that control the behaviour of CGFs in a real military simulation system. The objective was to imitate a recorded behaviour. As mentioned in the previous section, other researchers have used GAs to generate behaviour trees, but we have not seen other work that has used GAs for learning from observation. Also, the other research has been done with simpler simulation systems.

We consider the experiment to be a success. The resulting behaviour trees reproduce the most essential parts of the behaviour used to record the example data, which shows that the system is able to replicate simple behaviour by generating and evolving behaviour trees.

We were surprised by the fact that the simulation system is not deterministic, which resulted in significant variation in the fitness of a behaviour tree when executed on the same scenario multiple times. Assuming that we are able to solve the issues with stochastic simulation outcomes, the system's ability to replicate more complex behaviour seems promising. Hence, solving this issue should has the highest priority going forward.

Increasing the complexity of the objective and scenarios is also very relevant. E.g. using a number of agents to perform a military action called bounding overwatch where multiple agents must work together to advance forward. See [8] for more information on bounding overwatch. The agents could use radio messages to communicate when the next agent should advance. Extending the experiment described in this paper with variable movement speeds and using perceived truth target information are other potential ways of increasing experiment complexity.

In addition to implementing a larger variety of action leaf nodes, we wish to introduce other types of composite nodes in future experiments, e.g., random and parallel composite nodes. We also wish to include the use of decorators, that alter the resulting status of a single node.

The example used in this experiment was recorded with a manually created behaviour tree used to control the follower unit. This was done to have a representation of optimal behaviour to compare the results with, making visual analysis easier. However, for future experiments, the example behaviour should be recorded with the unit controlled by a human, e.g. using VR-Engage to control the example unit with mouse and keyboard. It should be interesting to compare how the person controlling the unit would represent his/her own behaviour to how the computer ends up representing it.

Finally, we aim at investigating the possibilities of distributing the simulation to reduce the required run-time of simulating and evaluating behaviour trees. Lim [12] had a similar problem, where running the entire experiment would take approximately 41 days. They were able to distribute the computation to 20 computers, reducing the number of days to approximately 2 days per experiment. This is also an option for our simulation system and would allow for a larger population or a higher number of epochs in the experiment.

References

[1] S. Bruvoll, J. E. Hannay, G. K. Svendsen, M. L. Asprusten, K. M. Fauske, V. Kvernelv, R. A. Løvlid, and J. I. Hyndøy, "Simulation-supported wargaming for analysis of plans," in *NATO Modelling and Simulation Group Symphosium. M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (MSG-133)*, 2015.

[2] M. R. Endsley, "Toward a theory of situation awareness in dynamic systems," *Human factors*, vol. 37, no. 1, pp. 32–64, 1995.

[3] M. R. Endsley and D. J. Garland, "Pilot situation awareness training in general aviation," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 44, no. 11. SAGE Publications Sage CA: Los Angeles, CA, 2000, pp. 357–360.

[4] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *arXiv preprint arXiv:1504.05811*, 2015.

[5] A. Toubman, G. Poppinga, J. J. Roessingh, M. Hou, L. Luotsinen, R. A. Løvlid, C. Meyer, R. Rijken, and M. Turčaník, "Modeling cgf behavior with machine learning techniques: Requirements and future directions," in *Proceedings of the 2015 Interservice/Industry Training, Simulation, and Education Conference*, 2015, pp. 2637–2647.

[6] M. G. Core, H. C. Lane, M. Van Lent, D. Gomboc, S. Solomon, and M. Rosenberg, "Building explainable artificial intelligence systems," in *AAAI*, 2006, pp. 1766–1773.

[7] J. J. Roessingh, A. Toubman, J. van Oijen, G. Poppinga, R. A. Løvlid, M. Hou, and L. Luotsinen, "Machine learning techniqus for autonomous agents in military simulations - multum in parvo," in *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 3445–3450.

[8] F. Kamrani, L. J. Luotsinen, and R. A. Løvlid, "Learning objective agent behavior using a data-driven modeling approach," in *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 002 175–002 181.

[9] Bohemia Interactive. (2016) VBS. [Online]. Available: https://bisimulations.com/virtual-battlespace-3

[10] MÄK. (2018) VR-Forces. [Online]. Available: https://www.mak.com/products/simulate/vr-forces

[11] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Pearson Education, 2014, pp. 129–132.

[12] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game defcon," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 100–110.

[13] NATO NSA, *STANAG 4603 - Modelling and Simulation Architecture Standards for Technical Interoperability: High Level Architecture (HLA)*, 2nd ed., 2015.

[14] Simulation Interoperability Standards Organization (SISO), *Standard for Guidance, Rationale, and Interoperability Modalities (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download\&EntryId=30822, 2015, SISO-STD-001-2015.

[15] ——, *Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download\&EntryId=30823, 2015, SISO-STD-001.1-2015.

[16] A. Alstad, O. Mevassvik, M. Nielsen, R. Løvlid, H. Henderson, R. Jansen, and N. de Reus, "Low-level battle management language," in *Proceedings of the 2013 Spring Simulation Interoperability Workshop*, no. 13S-SIW-032, 2013.

[17] J. Ruiz, D. Dsert, A. Hubervic, P. Guillou, R. Jansen, N. de Reus, H. Henderson, K. Fauske, and L. Olsson, "BML and MSDL for multi-level simulations," in *Proceedings of the 2013 Fall Simulation Interoperability Workshop*, no. 13F-SIW-002, 2013.

[18] A. Alstad, R. A. Løvlid, S. Bruvoll, M. N. Nielsen, and O. M. Mevassvik, "Autonomous simulation of a battalion operation - seamless integration of command and control and simulation for planning and training," Forsvarets forskningsinstitutt, FFI-rapport 2013/01547, 2013.

[19] VT MAK. (2017) VR-Engage. [Online]. Available: https://www.mak.com/products/simulate/vr-engage

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[21] R. Al-Aomar, "Incorporating robustness into genetic algorithm search of stochastic simulation outputs," *Simulation Modelling Practice and Theory*, vol. 14, no. 3, pp. 201–223, 2006.

Automating Behaviour Tree Generation for Simulating Troop Movements

Gabriel Berthling-Hansen, Eivind Morch, Rikke Amilde Seehuus, Odd Erik Gundersen

We used machine learning to learn behaviour from examples.

This project explores automatic generation of behaviour based on recorded simulated exercises. Currently, behaviour models for computer generated forces (CGF) are made manually, which is a difficult and time-consuming process. By using machine learning to learn from examples, we hope to provide a faster and less expensive method.

The behaviour is represented using behaviour trees.

Behaviour trees have popularly been used for modelling the behaviour of computer-controlled units in video games. They provide a scalable and modular solution for representing complex behaviour, and enable visual analysis of the behaviour they represent.

A multi-objective genetic algorithm was used to evolve the behaviour trees.

The system uses the NSGA-II algorithm to generate and evolve a population of behavior trees. Each tree in the population is simulated in complex simulation systems and then evaluated based on recorded data. The next population is selected by using the evaluation scores. By using multi-objective algorithms, the system maintains and improves a diverse set of different behaviour. This allows the system to progressively improve the generated behavior to imitate the behavior shown in the recorded exercises.

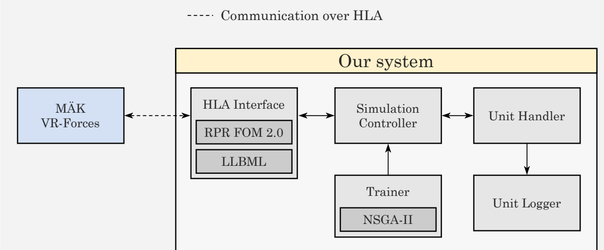
The experiment was performed with a real military simulation system.

The system uses a real military simulation system, which consist of complex real-life scenarios and terrains, as well as specialized military entity models. For our experiment we used VR-Forces from MÄK. Our system connects to the simulation system using high-level architecture (HLA), and records and processes data from the simulated entities. While the simulation is running, the generated behaviour trees are used to choose what commands to send to the simulated entities. The recorded data is also used to calculate evaluation scores for each behaviour tree.

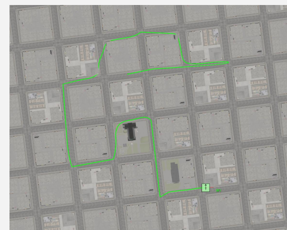
We managed to evolve behaviour trees that imitate a follow-behaviour.

The example behaviour was recorded from a unit controlled by the manually created behaviour tree shown below. The training was done on two scenarios, with different terrains and target movement paths. One of the resulting behaviour trees is shown in the lower right corner, representing behaviour that closely resembles the example behaviour.

System architecture

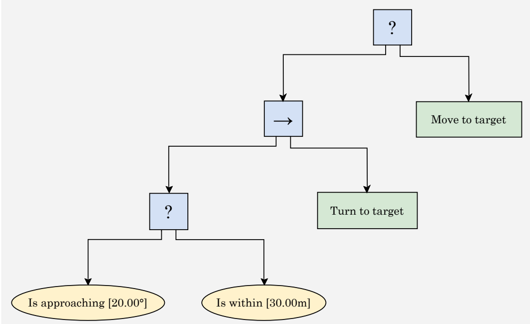


Scenario 1 path

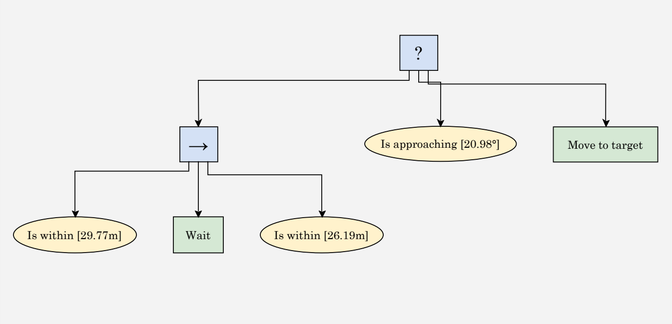


Scenario 2 path

Manually created behaviour tree



Generated behaviour tree



FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment

NTNU
Norwegian University of Science and Technology

Appendix B

Single-Page Figure Versions

In this appendix, single-page versions of Figures 5.5 and 5.7 to 5.10 are included. All figures use scalable vector graphics.

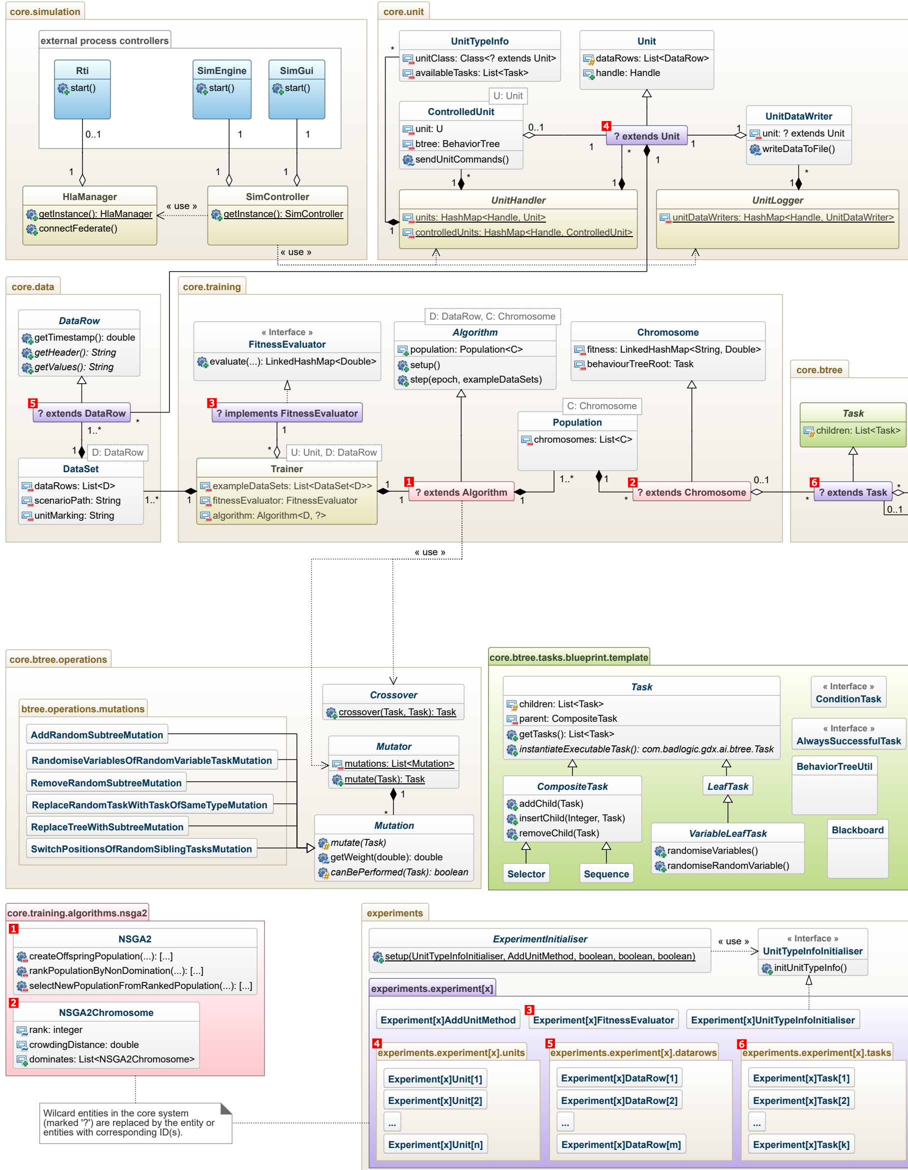


Figure B.1: Single-page version of Figure 5.5 – Class diagram of the core system and the experiment package. The colouring of the system entities follows the system diagram colour-coding, specified in Table 5.1. The core system requires class implementations from both the experiment package and an algorithm implementation to work. These missing classes are shown as purple (experiment) and pink (algorithm) classes with names that start with "?" and have red number boxes in the top left corner. The numbers indicate where each class or set of classes from the experiment and algorithm packages are used in the core system.

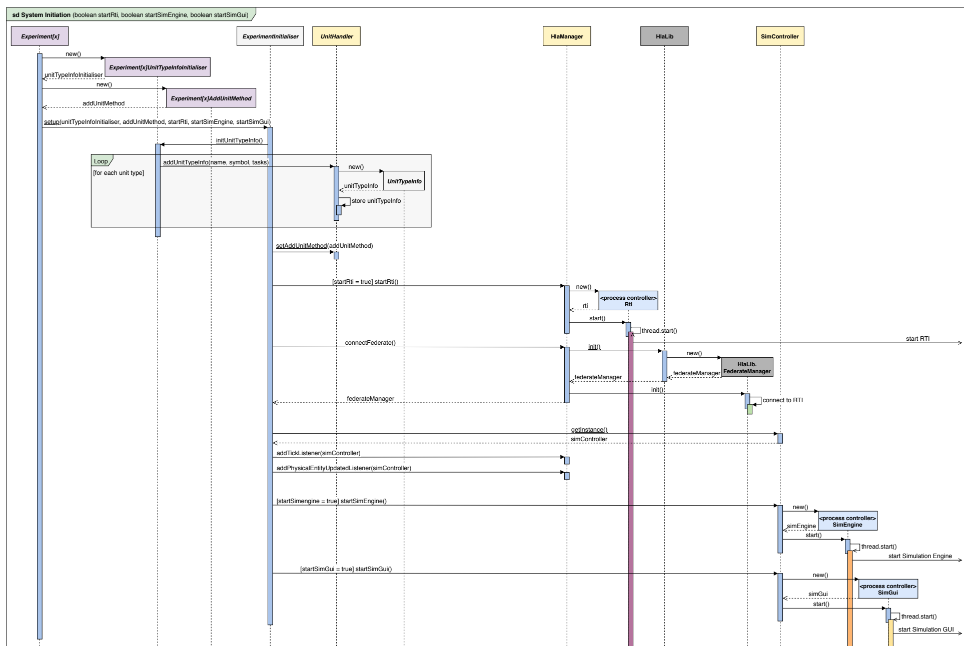


Figure B.2: Single-page version of Figure 5.7 – Sequence diagram of the system initiation process

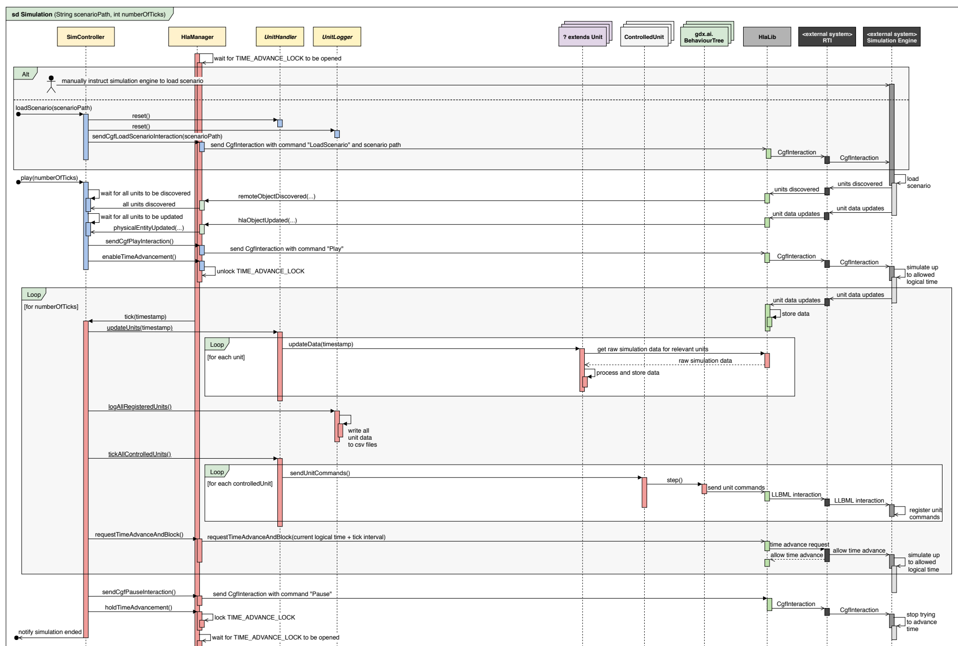


Figure B.3: Single-page version of Figure 5.8 – Sequence diagram of the simulation process

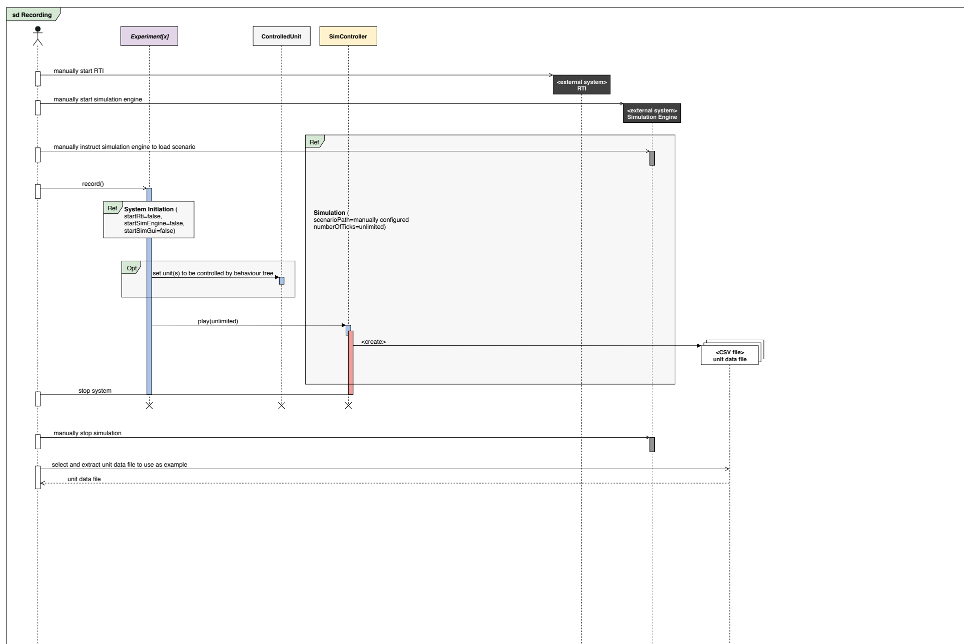


Figure B.4: Single-page version of Figure 5.9 – Sequence diagram of the recording process. References Figures B.2 and B.3 as subprocesses.

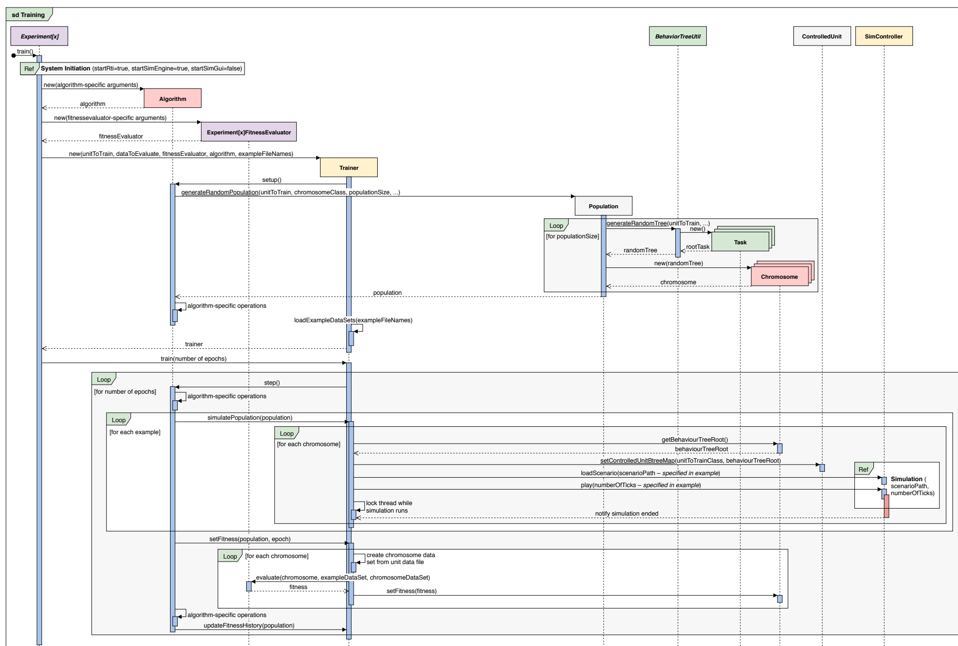


Figure B.5: Single-page version of Figure 5.10 – Sequence diagram of the training process. References Figures B.2 and B.3 as subprocesses.

Appendix C

VR-Forces Settings

This appendix includes the versions, plugins and settings of the MÄK VR-Forces software used when conducting our experiments.

MÄK system versions

- RTI: 4.4.2d VC10
- VR-Forces: 4.5 VC10
- VR-Link: 5.3.1 VC10

MÄK VR-Forces plugins

- LLBML
- CgfControl (LLBML addon)

VrfSim.mtl settings

- timeManagementMode: 1
- sendFedTime: 1
- disableParallelTick: 1

VrfSimSettings.xml

- myTranslationThreshold: 0
- myRotationThreshold: 0

Appendix D

Literature Review Notes

This appendix includes notes from the literature review. The notes include the title, author(s), year of publication, type of publication and relevant pages. They also include a topic, where we would write down the topics of the article, and a relevance and credibility score. The relevance and credibility score is a five star system where five stars are extremely relevant and 1 star is not so relevant. The relevance score is a measure of how relevant the publication is to the work done in the thesis. The credibility score represents our subjective opinion of how credible the publication is.

Lim2010 – Evolving BTs for [...] DEFCON

Title: Evolving Behaviour Trees for the Commercial Game DEFCON
Author: Chon-U Lim, Robin Baumgarten and Simon Colton
Year: 2010
Type: Article
Pages: 1–9

Topic: Using Grammatical Evolution (GE) to evolve BTs
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *Behaviour trees provide the possibility of improving on existing Artificial Intelligence techniques in games by being simple to implement, scalable, able to handle the complexity of games, and modular to improve reusability. This ultimately improves the development process for designing automated game players. We cover here the use of behaviour trees to design and develop an AI-controlled player for the commercial real-time strategy game DEFCON. In particular, we evolved behaviour trees to develop a competitive player which was able to outperform the game's original AI-bot more than 50% of the time. We aim to highlight the potential for evolving behaviour trees as a practical approach to developing AI-bots in games.*

Notes:

1. Investigates evolutionary techniques for developing BTs
2. (p. 1) Arguments for BTs advantages over FSM for game design.
3. (p. 5) Crossover through swapping random subtrees between the parents
4. (p. 5) Mutation
   a) Additive mutation (adding subtree)
   b) Altering mutation (changing node details etc)
5. The article also talks about the fitness functions it use to score its bot, these seem to be high level fitness functions.
6. (p. 9) The article concludes by speculating that GE might need to supplemented with other AI methods (marked section).
7. **Should be noted that with training against a single, deterministic AI-opponent, in combination with simplified behaviour (defence then attack) in the training AI, most likely resulted in overfitting.**

Simpson2014 – BTs for AI: How they work

Title: Behavior trees for AI: How they work
Author: Chris Simpson
Year: 2014
Type: Online (blog)
Pages: 1–12

Topic: A overview of how BTs work
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ☆ ☆ ☆

Abstract: *None*

Notes:

1. Very informative. Provides good examples and talks about all node types, execution orders and so on.
2. Suggested BT lib: `https://github.com/gaia-ucm/jbt`
3. (p. 8-9) Interesting point regarding recursive use of BTs. Can create deep behaviour from small trees run recursively.

Robertson2015 – Building BTs from Obs. in RTS

Title: Building Behavior Trees from Observations in Real-Time Strategy Games
Author: Glen Robertson and Ian Watson
Year: 2015
Type: Article
Pages: 1–6

Topic: BT reduction
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ☆ ☆

Abstract: *This paper presents a novel use of motif-finding techniques from computational biology to find recurring action sequences across many observations of expert humans carrying out a complex task. Information about recurring action sequences is used to produce a behavior tree without any additional domain information besides a simple similarity metric – no action models or reward functions are provided. This technique is applied to produce a behavior tree for strategic-level actions in the real-time strategy game StarCraft. The behavior tree was able to represent and summarise a large amount of information from the expert behavior examples much more compactly. The method could still be improved by discovering reactive actions present in the expert behavior and encoding these in the behavior tree.*

Notes:

1. Never tested the resulting tree, so this may not work in practice.
2. (p. 4–5) Discusses BT reduction based on GLAM2 algorithm
   a) Uses simulated annealing on sequence patterns
   b) Originally used for DNA – used on strings
   c) Note: Can be combined with GE? – see Perez2011 (1.9)

Colledanchise2015 – Learning of BTs for Autonomous Agents

Title: Learning of Behavior Trees for Autonomous Agents
Author: Michele Colledanchise, Ramviyas Parasuraman, and Petter Ögren
Year: 2015
Type: Article
Pages: 1–7

Topic: BTs and genetic programming
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ☆

Abstract: *Definition of an accurate system model for Automated Planner (AP) is often impractical, especially for real-world problems. Conversely, off-the-shelf planners fail to scale up and are domain dependent. These drawbacks are inherited from conventional transition systems such as Finite State Machines (FSMs) that describes the action-plan execution generated by the AP. On the other hand, Behavior Trees (BTs) represent a valid alternative to FSMs presenting many advantages in terms of modularity, reactiveness, scalability and domain-independence. In this paper, we propose a model-free AP framework using Genetic Programming (GP) to derive an optimal BT for an autonomous agent to achieve a given goal in unknown (but fully observable) environments. We illustrate the proposed framework using experiments conducted with an open source benchmark Mario AI for automated generation of BTs that can play the game character Mario to complete a certain level at various levels of difficulty to include enemies and obstacles.*

Notes:

1. Only focuses on evolution – no LfO.
2. **Provides a detailed description and explanation of all the steps in the algorithm – crossover, mutation, selection, diversity rank method, learning algorithm, anti-bloat control.**
3. Spends multiple pages explaining how BTs and genetic programming works. Good, graphical explanation of BT.
4. Provides useful graphical and mathematical explanations for mutation, crossover and ranking.
5. (p. 2) Mentions the use of GE in Perez2011, but uses EA instead due to possibility of a “natural” representation.
6. (p. 4) Uses only unary replacement mutation. Lim2010 (1.2) provides an additional mutation type (additive).
7. (p. 6) Discusses and describes methods of handling bloated BTs.

Zhang2016 – Integrated Learning Framework for BM

Title: Towards An Integrated Learning Framework for Behavior Modeling of Adaptive CGFs
Author: Qi Zhang, Quanjun Yin, Kai Xu
Year: 2016
Type: Article
Pages: 7–12

Topic: Generating BTs through LfO, combined with CBR in runtime
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ★

Abstract: *Computer generated forces (CGFs) are autonomous or semi-autonomous actors within military, simulation based, training and analyzing applications. Rapid, realistic and adaptive behavior modeling for CGFs is imperative and challenging. Traditional modeling approaches like rule-based script usually need time-consuming, repetitive endeavor and result in rigid, predictable behavior performance. Recent developments introducing Machine Learning (ML) techniques, such as dynamic script or neural network models, always present as black box systems, which are difficult to understand and revise for subject matter experts (SMEs). To overcome these limitations, we propose an integrated learning framework to facilitate adaptive CGF behavior modeling. The framework represents domain knowledge explicitly as Behavior Trees (BTs), and integrates learning BTs automatically from demonstration and Reinforcement Learning (RL) node into BTs. Besides, a CBR-style planner is adopted to retrieve executable behavior for diverse situations encountered at runtime. Through aforementioned components, the framework can make full use of the advantages of various learning approaches and knowledge sources to generate realistic and adaptive behaviors for CGFs easily.*

Notes:

1. **Provides a recipe for creating BTs from examples by directly copying actions as sequences, and combining them by applying a set of rules to structure them. Contrast to using GA to converge to equal resulting behaviour.**
2. (p. 7) Discusses drawbacks of different ML generated behaviour (black box, slow convergence, etc.)
3. (p. 7, 10) Use CBR to select behaviour tree during runtime.
4. References and discusses multiple relevant articles (most included here).
5. (p. 10) Provides requirements for automatically built BTs. Probably good to enforce for both faster convergence and better performance.
6. (p. 10) Definition of a demonstration:

- A set of demonstrations of experts carrying out a single task, $\{E_1, E_2, ..., E_n\}$.
- A demonstration is a sequence of cases ordered by time, $E_i = (C_{i1}, C_{i2}, ..., C_{im})$.
- A case is an observation and action pair, $C_{ij} = (O_{ij}, A_{ij})$.
- An observation and an action are selected state information available to agent, (eg. A key-value mapping).

7. (p. 10) Provides a detailed recipe for generating the BT from observation.

Ontanon2011 – Unified Framework for Obs. Learning

Title: Towards a Unified Framework for Learning from Observation
Author: Santiago Ontanon, Jose L. Montana and Avelino J. Gonzalez
Year: 2011
Type: Article
Pages: 1–6

Topic: Formalisation of Learning from Observation
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ☆

Abstract: *This paper discusses the recent trends in machine learning towards learning from observation (LfO). These reflect a growing interest in having computers learn as humans do — by observing and thereafter imitating the performance of a task or an action. We discuss the basic foundation of this field and the early research in this area. We then proceed to characterize the types of tasks that can be learned from observation and how to evaluate an agent created in this manner. The main contribution of this paper is a joint framework that unifies all previous formalizations of LfO.*

Notes:

1. (p. 1) LfO definition: "The agent shall adopt the behavior of the observed entity solely from interpretation of data collected by means of observation."
2. (p. 2) A behaviour trace is defined as the change of the set of variables the actor can control over time $[(t_1, y_1), ..., (t_n, y_n)]$
3. (p. 2) "The evolution of the environment is captured into an input trace". $[(t_1, x_1), ..., (t_n, x_n)]$
4. (p. 2) A learning trace is a combination of a beavhiour trace and an input trace. $[(t_1, x_1, y_1), ..., (t_n, x_n, y_n)]$
5. (p. 3) Evaluating the agent
   - Evaluate the performance of the agent A performing task T. Regardless of how the actions match the actors actions.
   - Evaluate how the actions of the agent and actor compares
   - Evaluate how the generated model compares with the model that created the traces, this is not as relevant for us as the traces will be generated by humans. "This evaluation method is specially interesting to assess whether a given procedure can be recovered by learning from a set of learning traces."
6. (p. 3) Key factors that determine the complexity of an LfO task
   - Generalisation.
   - Planning.

- Known environment. If the learner does not have a model of the environment then it might have to learn this model at some point.

7. (p. 3) Levels of difficulty
    - Strict imitation
    - Reactive skills. Includes the above and generalisation
    - Tactical behaviour in known environments. Includes the above and planning
    - Tactical behaviour in unknown environments. Includes generalisation, may include planning and has an unknown environment.

8. (p. 4 – 6) The article delves into detail about its statistical models for the solving the problems. It also has some examples for each of the difficulty levels. This may be mostly relevant as an argument to why we should not use GP or any other AI method, as we can achieve a better estimate using these statistical models.

Stein2011 – Human-Like Through Observation and Experience

Title: Building High-Performing Human-Like Tactical Agents Through Observation and Experience
Author: Gary Stein, Avelino J. Gonzalez
Year: 2011
Type: Article
Pages: 792–804

Topic: Combining observational and experiential learning
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *This paper describes a two-phase approach for automating the agent-building process when the agent is to perform tactical tasks. The research is inspired by how humans learn—first by observation of a teacher's performance and then by practicing the performance themselves. The objectives of this approach are to produce a high-performing agent that 1) approaches or exceeds the proficiency of a human and 2) does so in a human-like manner. We accomplish these objectives by combining observational learning with experiential learning. These processes are executed sequentially, with the former creating a competent but somewhat limited human-like model from scratch, and the latter improving its performance without significantly eroding its human-like qualities. The process is described in detail, and test results confirming our hypothesis are described.*

Notes:

1. Presents arguments for how the method is equal to human learning
2. (p. 2) "In observational learning, the objective is to be similar to the human movements, whereas in experiential learning, the objective is to be as high performing as possible."
3. (p. 8) The agent based on Violet outperformed the agent trained from Orange, although Orange outperformed Violet. The researchers propose that this happens because the agents based on Violet is able to learn how to escape from bad situations whereas from Orange, a good trainer, they are unable to learn this.
4. (p. 9) The researchers also try to find out how human like the agents are and point out that there is a difference between the actions taken and the output seen on the screen. For example in the car domain, the agent would move the steering-wheel back and forth (not affecting the movement of the car) but it is unnatural for humans.
5. (p. 10–11) The observational and experiential agents improved on the agents

that were only observational. And improved on the experiential agent or had the same score, while also being more human like.

6. Some human like traits were lost when using both observational and experiential methods compared to only observational

Perez2011 – Evolving BTs [...] Using Grammatical Evolution

Title: Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution
Author: Diego Perez et al.
Year: 2011
Type: Article
Pages: 1–11

Topic: Evolving BTs with grammatical evolution (GE)
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *This paper investigates the applicability of Genetic Programming type systems to dynamic game environments. Grammatical Evolution was used to evolve Behaviour Trees, in order to create controllers for the Mario AI Benchmark. The results obtained reinforce the applicability of evolutionary programming systems to the development of artificial intelligence in games, and in dynamic systems in general, illustrating their viability as an alternative to more standard AI techniques.*

Notes:

1. Intro to Grammatical Evolution (GE):
   `https://cran.r-project.org/web/packages/gramEvol/vignettes/ge-intro.pdf`
2. (p. 8) The researchers determined that adding a default sequence that would always be executed if no other sequences were executed was crucial. If this was not present, most agents would end up not moving the character.
3. (p. 12) "These results obtained strengthen the idea that GP systems are serious alternatives to more traditional AI algorithms, either on their own or combined into hybrid systems."

Hou2015 – Modeling CGF Behaviour [...] Req. and Future Dir.

Title: Modeling CGF Behavior with Machine Learning Techniques: Requirements and Future Directions
Author: Ming Hou et al.
Year: 2015
Type: Article
Pages: 2 – 12

Topic: Requirements and future directions for using machine learning in CGF behaviour
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *Commercial/Military-Off-The-Shelf (COTS/MOTS) Computer Generated Forces (CGF) packages are widely used in modelling and simulation for training purposes. Conventional CGF packages often include artificial intelligence (AI) interfaces, with which the end user define CGF behaviors. We believe machine learning (ML) techniques can be beneficial to the behavior modelling process, yet such techniques seem to be underused and perhaps underappreciated. This paper aims at bridging the gap between users in academia and the military/industry at a high level when it comes to ML and AI. Also, specific user requirements and how they can be addressed by ML techniques are highlighted with the focus on the added ML value to CGF packages. The paper is based on the work of the NATO Research Task Group IST-121 RTG-060 'Machine Learning Techniques for Autonomous Computer Generated Entities'.*

Notes:

1. (p. 7) "Moreover, development of CGFs often remains a painstaking development of a set of rules (for example 'if-then rules') that need to be derived for each specific problem or situation to be resolved, based on the manual elicitation of operational expertise."
2. (p. 7) ML might lead to more desirable behaviour, which in turn might lead to more autonomous CGFs which means less personnel is required for preparing and planning an operation.
3. (p. 8) "While ML techniques offer many benefits, they rely on consistent and accurate data to learn from, and one is not guaranteed sensible or even safe behavior."
4. (p. 8) The researchers defined some computational and functional requirements in regards to using ML.
5. (p. 8) Computational requirements

a) (p. 8) Speed. Behavior generation should be fast, as it is (possibly) done live
b) (p. 8) Effectiveness. Generated behavior should be effective, even while the system is still learning.
c) (p. 8) Robustness. Generated behavior has to be able to cope with randomness and unexpected events.
d) (p. 8) Efficiency. Generated behavior should quickly be optimized based on few interaction moments with the human participant.

6. (p. 8) Functional requirements
   a) (p. 8) Clarity. Generated behavior should be easily interpretable by human operators.
   b) (p. 8) Variety. A variety of behavior should be generated, as repeated behavior can be uninteresting or even suspicious.
   c) (p. 8) Consistency. The number of interaction moments needed to generate or adapt behavior should have low variance and should be independent from the behavior of the human participant.
   d) (p. 8) Scalability. Generated behavior should be scalable to the skills of the human participant.
7. (p. 10) The researchers suggest an architecture that follows two principles
   a) (p. 10) "Decoupling learning CGF models from the simulation application or from the scenario management application,"
   b) Enabling the distribution of such models at different client-CGFs.

Dey2013 – Enhancing BTs using Q-Learning

Title: QL-BT: Enhancing Behaviour Tree Design and Implementation with Q-Learning
Author: Rahul Dey, Chris Child
Year: 2013
Type: Article
Pages: 1–8

Topic: Enhancing BTs with Q-learning
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ☆

Abstract: *Artificial intelligence has become an increasingly important aspect of computer game technology, as designers attempt to deliver engaging experiences for players by creating characters with behavioural realism to match advances in graphics and physics. Recently, behaviour trees have come to the forefront of games AI technology, providing a more intuitive approach than previous techniques such as hierarchical state machines, which often required complex data structures producing poorly structured code when scaled up. The design and creation of behaviour trees, however, requires experience and effort. This research introduces Q-learning behaviour trees (QL-BT), a method for the application of reinforcement learning to behaviour tree design. The technique facilitates AI designers' use of behaviour trees by assisting them in identifying the most appropriate moment to execute each branch of AI logic, as well as providing an implementation that can be used to debug, analyse and optimize early behaviour tree prototypes. Initial experiments demonstrate that behaviour trees produced by the QL-BT algorithm effectively integrate RL, automate tree design, and are human-readable.*

Notes:

1. **Emphasises the lack of research on automated manipulation or improvement of initial BTs.**
2. References Lim2010 (1.2) and Perez2011 (1.9).
3. Q-learning requires discrete states – might be problematic in the simulation environment.
4. Deepest sequence nodes are found in the BT, these are made into the actions of the Q-learning.
5. Condition nodes are replaced with Q-condition nodes which is a "lookup table containing all high-utility states"
6. (p. 5) Setup of the game world, agents and BT
7. Rewards were given form a predefined table with state-action pairs. e.g.

health low any action gave -10 reward.

8. A drawback of QL-BT is it's reliance on correct Q-values.
9. Q learning grows exponential to the state-action space of the agents
10. A drawback of the algorithm is that the simulation time for a complex time can be intractable.

Colledanchise2017 – Synthesis of Correct-by-Construction BTs

Title: Synthesis of Correct-by-Construction Behavior Trees
Author: Michele Colledanchise, Richard M. Murray, and Petter Ogren
Year: 2017
Type: Article
Pages: 1–8

Topic: Synthesising correct-by-construction BTs using Linear Temporal Logic (LTL)
Relevance: ★ ★ ☆ ☆ ☆
Credibility: ★ ★ ★ ★ ☆

Abstract: *In this paper we study the problem of synthesizing correct-by-construction Behavior Trees (BTs) controlling agents in adversarial environments. The proposed approach combines the modularity and reactivity of BTs with the formal guarantees of Linear Temporal Logic (LTL) methods. Given a set of admissible environment specifications, an agent model in form of a Finite Transition System and the desired task in form of an LTL formula, we synthesize a BT in polynomial time, that is guaranteed to correctly execute the desired task. To illustrate the approach, we present three examples of increasing complexity.*

Notes:

1. (p. 2) Explains BT and LTL (Linear Temporal Logic)
2. As you will have to define an LTL formula for the problem, this is probably not a possibility for our project.
3. They are able to generate a BT in polynomial time that is guaranteed to solve the assignment.

SMC2016 – Learning Objective Agent Behavior using DDBM

Title: Learning Objective Agent Behavior using a Data-driven Modeling Approach
Author: Farzad Kamrani, Linus J. Luotsinen, Rikke Amilde Løvlid
Year: 2016
Type: Article
Pages: 1–7

Topic: DDBM
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ★

Abstract: *This paper presents a data-driven approach towards the modeling of agent behaviors in a full-fledged, commercial off-the-shelf simulation milieu for tactical military training. The modeling approach employs machine learning to identify behavioral rules and patterns in data. Potential advantages of this approach are that it may improve modeling efficiency and, perhaps more importantly, increase the realism of the training simulator.*

In this work, we present an architecture outlining the main components of the data-driven behavior modeling approach. Using a prototype that implements the approach, we conduct and present results from an experiment targeting the learning of cooperative military movement tactics. It is shown that the prototype is capable of identifying the rules of the tactics. Moreover, it is shown that the agents are able to generalize such that the learned behavior can be applied in a new setting different from the one observed in the training data.

Notes:

1. **Example of using DDBM for bounding overwatch**
2. Gives a good intro to the field of DDBM.
3. Defines observational, experiential and hybrid learning.
4. Discusses positives and negatives with experiential learnin (computational creativity).
5. Describes and explains the structure and functionality of a BT.
6. Discusses the necessity and provides examples of data feature extraction.
7. Promising results.

SMC2016 – Evolved Creative Intelligence for CGF

Title: Evolved Creative Intelligence for Computer Generated Forces
Author: Farzad Kamrani, Linus J. Luotsinen, Rikke Amilde Løvlid
Year: 2016
Type: Article
Pages: 1–8

Topic: Evolving CGFS's behaviour using GP (computational creativity)
Relevance: ★ ★ ★ ★ ★
Credibility: ★ ★ ★ ★ ★

Abstract: *This paper provides an example of using genetic programming for engendering computational creativity in computer generated forces, i.e. simulated entities used to represent own, opponent and neutral forces in military training or decision support applications. We envision that applying computational creativity in the development of computer generated forces may not only reduce development costs but also offer more interesting and challenging training environments.*

In this work we provide experimental results to strengthen our arguments using a predator/prey game. We show that predator behavior created by a computer, using genetic programming, surpasses predator behavior manually programmed by humans and argue that the sparse automatically generated code is unlikely to be generated by a human and therefore can be considered as a good example of computational creativity. Although the experiments are not conducted in a real-world training simulator they provide valuable insight that exemplifies the opportunities and the challenges of computational creativity applied to computer generated forces.

Notes:

1. Does not mention observational learning
2. (p. 1) Explains the purpose (with source) for simulations of military scenarios.
3. (p. 1) Reasons for why computational creativity is useful in simulations.
4. Introduces genetic programming (GP), and explains the program representation, variables and fitness functions in detail.
5. Uses "Hunting Game" toy problem for comparison
6. Compares human-made and computer-generated behaviour
   -> computer-generated performs better
7. (p. 7) Discusses challenges with applying this technique in real-time, complex environments such as VBS3:
   a) Real-time simulation – actions and fitness calcuation is done in real-

time

b) Fitness function – Difficult in complex environments such as VBS3 (This can be solved with Pareto-front?)

c) Complexity – Larger search space due to environment complexity (physics, sensors, available actions, etc.)

8. (p. 8) Argues for the benefits of successfully applying computational creativity in complex military simulations.

MSG2015 – DDBM for CGF

Title: Data-Driven Behavior Modeling for Computer Generated Forces
Author: Linus J. Luotsinen, Rikke Amilde Løvlid
Year: 2015
Type: Article
Pages: 1–14

Topic: DDBM (ANN and DT)
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *Computer generated forces (CGFs) are autonomous or semi-autonomous actors within military, simulation based, training and decision support applications. The CGF is often used to replace human role-players in military exercises to, ultimately, improve training efficiency. The modeling and development of CGFs is a complex, time-consuming and expensive endeavor where military domain expertise and doctrinal knowledge are interpreted and programmed into the CGF by hand. Furthermore, CGFs often represent human actors and behaviors (pilots, soldiers, manned systems, etc.) making it an even more challenging task.*

In recent years the Artificial Intelligence (AI) research community achieved some remarkable results where Intelligent Agents (IA) successfully defeated human champions in games such as chess and Jeopardy. AI researchers have demonstrated that Machine Learning (ML) algorithms can be used to learn IA behaviors from recorded observations such as log-files, GPS coordinate traces and, more recently, pixels from images and video.

The ability of the machine learning approach to learn the "behavioral rules" of the CGFs, which we from now on will refer to as Data-Driven Behavior Modeling (DDBM), has many potential advantages compared to the traditional CGF modeling approach where the "behavioral rules" are manually handcrafted using subject matter experts and doctrines. Using DDBM the modeling efficiency with respect to cost and time may improve, in particular, when modeling complex CGFs designed to mimic human actors and behaviors within complex environments. The DDBM approach may also improve behavior realism and objectiveness resulting in better and more realistic training and decision support tools.

In this work we introduce the concept of DDBM including its main components in the context of CGF behavior modeling. We also provide preliminary results of experiments where our DDBM-prototype is used to generate behaviors using both observational and experiential learning strategies.

Notes:

1. (p. 2) Investigates:

   a) Is DDBM more efficient than traditional manual behaviour programming? – N/A
   b) Can DDBM be used to create behaviors that are too complex to model using the traditional modeling approach? – N/A
   c) Can DDBM be used to create objective behavior models that imitates the behavior of its real-world counterpart? – Yes

2. Identified challenges:
   a) Data problems such as insufficient, incomplete and noisy data.
   b) Verification and validation problems related to black-box representations such as neural networks that are difficult to visualize and analyze by humans experts.
   c) Real-time simulation problems caused by advanced feature extraction functions (e.g. terrain analysis, route planning).
   d) The need for intuitive and easy-to-use DDBM authoring tools capable of visualizing, editing and processing datasets acquired synthetically or from military exercises.

3. (p. 4) Good overview figure of DDBM.

4. Emphasises lack of software targets data-acquisition, visualisation and preprocessing.

5. Uses ANN and decision trees.

6. Successful observational learning, even with small datasets (passing experiment).

7. Worse results than with *SMC2016 – Evolved Creative Intelligence for CGF* (Predator-sheep experiment)

MSG2015 – Sim.-supported Wargaming for Analysis of Plans

Title: Simulation-supported Wargaming for Analysis of Plans
Author: Solveig Bruvoll, Jo E. Hannay, Guro K. Svendsen, Martin L. Asprusten, Kjell Magne Fauske, Vegard B. Kvernelv, Rikke A. Løvlid, Jens Inge Hyndøy
Year: 2015
Type: Article
Pages: 1–17

Topic: Increase quality of plans and planning time (SWAP)
Relevance: ★ ☆ ☆ ☆ ☆
Credibility: ★ ★ ★ ★ ★

Abstract: *Wargaming is used in the military decision making process to visualize the execution of a preliminary plan or course of action in order to analyze and discover weaknesses and possibilities. The wargaming is traditionally done manually on a paper map, and the course of events is determined based on the experience and assumptions of the officers conducting the wargame. This paper describes ongoing research in Norway on the development of a demonstrator for Simulation-supported Wargaming for Analysis of Plans – SWAP. The focus is particularly on the synchronization of cooperating and supporting units, aiming to enable the planning group to more easily distribute supporting units to its subordinates when the support is most needed. This tool is intended to integrate simulation-supported wargaming in the planning process and thereby increase the quality of plans and decrease the planning time. SWAP uses a computer generated force federated with an agent-based simulation of C2 and combat management for simulation of a plan. It takes as input elements of a brigade plan from the Norwegian Command and Control Information System (C2IS) (NORCCIS). A web-based tool has been developed to support the officers in creating a synchronization matrix and to review the results of the simulation. The user can follow the simulated execution of the plan in the C2IS and receive information, such as fuel and ammunition consumption and casualties on both sides. C2 to Simulation (C2SIM) standards and a service-based approach is used to promote interoperability, while the simulation comprises a time-managed High Level Architecture (HLA) federation.*

Notes:

1. (p. 1) Introduction, how a plan is made
2. (p. 3-5) Explains the SWAP program
3. (p. 5-10) Development of SWAP and Architechture of SWAP
4. (p. 12) Route planning service, probably the only relevant thing for our system as talked about using it to calculate routes that we could use in our system.

Yao2011 – The Behavior Modeling of CGF based on ANN

Title: The Behavior Modeling of Computer Generated Warship Forces System Based on Neural Network
Author: Nan Yao, Jianyun Wang and Yaoxing Shang
Year: 2011
Type: Article
Pages: 1–6

Topic: Intelligent decision-making for surface warships
Relevance: ★ ★ ★ ☆ ☆
Credibility: ★ ★ ★ ☆ ☆

Abstract: *This paper focuses on an intelligent decisionmaking model of surface warship modeling in Computer Generated Forces (CGF) System. Different kinds of AI technologies were used to found the decision-making model, enable the surface warship CGF system to select battle region, layout path, identify targets, estimate threat and control firing. The model was validated through simulating with other CGF systems. Behavior model can make the Computer Generated Forces system more intelligent and actual.*

Notes:

1. Battle region selection employs maximum likelihood and fuzzy inference
2. The path planning is based on ordered state-space searching.
3. Used shallow ANN to determine the identification of the target. Was also based on some simple rules based on the velocity of objects and such
4. Used simple Perceptron network to determine the threat of the target.
5. Used simple rules to determine if the warship should file at the target

Floyd2011 – A CBR Framework for Developing Agents Using LfO

Title: A Case-Based Reasoning Framework for Developing Agents Using Learning by Observation
Author: Michael W. Floyd and Babak Esfandiarin
Year: 2011
Type: Article
Pages: 1–8

Topic: Learning from Observation with a case base
Relevance: ★ ★ ★ ☆ ☆
Credibility: ★ ★ ★ ★ ☆

Abstract: *Most realistic environments are complex, partially observable and impose real-time constraints on agents operating within them. This paper describes a framework that allows agents to learn by observation in such environments. When learning by observation, agents observe an expert performing a task and learn to perform the same task based on those observations. Our framework aims to allow agents to learn in a variety of domains (physical or virtual) regardless of the behaviour or goals of the observed expert. To achieve this we ensure that there is a clear separation between the central reasoning system and any domain-specific information. We present case studies in the domains of obstacle avoidance, robotic arm control, simulated soccer and Tetris.*

Notes:

1. No information about the meaning of what the inputs actually mean, i.e. that the touch sensor indicates that the robot is in contact with an obstacle.
2. No need to define the goals of the expert user or add non-observable features related to the goals that the expert user may have.

Fujimoto1998 – Time Management in HLA

Title: Time Management in the High Level Architecture
Author: Richard M. Fujimoto
Year: 1998
Type: Article
Pages: 1–21

Topic: Time management in HLA
Relevance: ★ ★ ★ ★ ☆
Credibility: ★ ★ ★ ★ ☆

Abstract:*Time management is required in simulations to ensure temporal aspects of the system under investigation are correctly reproduced by the simulation model. This paper describes the time management services that have been defined in the High Level Architecture. The need for time management services is discussed, as well as design rationales that lead to the current definition of the HLA time management services. These services are described, highlighting information that must flow between federates and the Runtime Infrastructure (RTI) software in order to efficiently implement time management algorithms*

Notes:

1. Physical time: Time of the simulation, e.g. Pearl Harbor midnight to 06:00 December 7 1941
2. Simulation time: Simulation engine's representation of time
3. Wallclock time: When the simulation was executed, e.g. 12:00 to 15:00 if the simulation takes 3 hours to complete and was started at 12:00
4. If realtime simulation then the simulation time pace and the wallclock time pace are matched.
5. (p. 6) Different methods to advance simulation time: event driven, time stepped, parallel discrete event simulation, wallclock time driven.
6. (p. 14 - 15) Time constrained and time regulating federates, lookahead
7. (p. 16) Timestamp guarantees

Weber2012 – ABL versus Behaviour Trees

Title: ABL versus Behaviour Trees
Author: Ben Weber
Year: 2012
Type: Online (blog)
Pages: 1–3

Topic: ABL and BTs
Relevance: ★ ☆ ☆ ☆ ☆
Credibility: ★ ☆ ☆ ☆ ☆

Abstract: *While preparing for the 2011 Paris Game AI conference, Alex Champandard asked me if there are any differences between ABL and behavior trees (BTs) at the planning level, which motivated me to dig a bit deeper into this topic. The goal of this post is to distinguish differences between behavior trees and ABL. While I am contrasting ABL with the Behavior Tree architecture described in Alex's chapter in AI Game Programming Wisdom 4, I am aware that there are several flavors of BT implementations so feel free to add feedback or corrections based on any BT variant.*

Notes:

1. Talks about the difference between ABL and BT. ABL is a behaviour language written in Java, whereas BT is a data structure. A result of this is that ABL cannot be modified during runtime (it needs to compile again).
2. "ABL runs asynchronously from the main game update, while BT are updated during an AI tick."
3. Another difference is when the action is executed, in ABL an action is executed when it is chosen. In BT an action is executed on the discretion of a scheduler.
4. There is also a difference in semantics between the two, the article goes on to explain the different nodes of ABL.