MODELLING ARTIFICIAL INTELLIGENCE IN GAMES USING MINDSET BEHAVIOR TREES

A Thesis
Submitted to the Faculty of Graduate Studies and Research in Partial Fulfillment of the Requirements for the Degree of

Master of Science
in
Computer Science
University of Regina

by
Ryan Keith Marcotte
Regina, Saskatchewan
May 2017

© 2017: R.K. Marcotte
Ryan Keith Marcotte, candidate for the degree of Master of Science in Computer Science, has presented a thesis titled, *Modelling Artificial Intelligence in Games Using MindSet Behavior Trees*, in an oral examination held on April 21, 2017. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner: Dr. Craig Gelowitz, Software Systems Engineering

Supervisor: Dr. Howard Hamilton, Department of Computer Science

Committee Member: Dr. Malek Mouhoub, Department of Computer Science

Committee Member: Dr. Xue Dong Yang, Department of Computer Science

Chair of Defense: Dr. Janis Dale, Department of Geology
Abstract

Behavior trees are a popular way of structuring artificial intelligence in games and other virtual reality applications. A behavior tree is a model of plan execution and is graphically represented as a tree. Nodes in a behavior tree either encapsulate actions to be performed or act as control flow components that direct traversal over the tree. The popularity of behavior trees stems from their maintainability, scalability, reusability, and extensibility. However, constructing behavior trees only using a programming language is difficult because the behavior tree cannot be easily visualized.

We introduce MindSet, a new architecture for constructing behavior trees. Accompanying the MindSet architecture is the MindSet Editor software and its corresponding MindSet application programming interface (API). MindSet Editor is designed for creating and modifying behavior trees using a graphical interface. The MindSet API is for marking code that can be imported into MindSet Editor. Using the API, users can define AI methods and their own custom behavior tree extensions.

We demonstrate MindSet’s usage for modelling the behavior of game entities controlled by AI in three simple game applications. With MindSet, programmers can develop AI code quickly and efficiently for any system requiring behavior control. We also show how utility-based prioritization behaviors can be incorporated into the base behavior tree architecture to build more dynamic behaviors.
Acknowledgements

I would like to thank my supervisor, Dr. Howard Hamilton, for guiding me through my M.Sc. program. I benefitted from his advice concerning academic procedures and research. I greatly appreciated his copious, detailed comments on the drafts of my thesis and associated software.

I am also grateful to Dr. Malek Mouhoub and Dr. Xue Dong Yang for serving on my examining committee.

I would also like to acknowledge the financial support of both the Natural Sciences and Engineering Research Council (NSERC) of Canada and the Faculty of Graduate Studies and Research (FGSR) at the University of Regina. Grants from NSERC were administered by Dr. Hamilton.
Post-Defense Acknowledgements

I would like to gratefully acknowledge my external examiner, Dr. Craig Gelowitz from the Faculty of Engineering, for his valuable feedback on my thesis. Thanks to Dr. Janis Dale from the Department of Geology for chairing my thesis defense.
Dedication

I dedicate this thesis to my mom and dad, Darlene and Keith, who have always supported me through every challenge, including the M.Sc. program. This work is also dedicated to my friend Kristy who continues to be an endless source of encouragement and inspiration.
Contents

Abstract ............................................................................................................................................. ii
Acknowledgements .......................................................................................................................... iii
Post-Defense Acknowledgements ................................................................................................. iv
Dedication .......................................................................................................................................... v
Contents ........................................................................................................................................... vi
List of Tables ................................................................................................................................... ix
List of Figures ................................................................................................................................... x
1 Introduction ................................................................................................................................... 1
  1.1 Motivation .................................................................................................................................. 3
  1.2 Problem Statement ................................................................................................................... 4
  1.3 Contributions ........................................................................................................................... 4
  1.4 Structure of the Remainder of the Thesis ............................................................................... 5
2 Behavior Trees ............................................................................................................................... 6
  2.1 Fundamental Behavior Tree Concepts ....................................................................................... 7
  2.2 Building a Behavior Tree ........................................................................................................... 14
  2.3 Comparisons to Hierarchical Finite State Machines ................................................................. 15
  2.4 Comparison to Heuristic Reasoning Methods ........................................................................... 21
  2.5 Comparison to Artificial Intelligence Formalisms ..................................................................... 25
  2.6 Overview of Existing Behavior Tree Editors ........................................................................... 28
    2.6.1 Behavior Designer ................................................................................................................. 28
    2.6.2 Behavior3 ............................................................................................................................. 29
    2.6.3 Behave ................................................................................................................................. 31
    2.6.4 Brainiac Designer ................................................................................................................ 32
    2.6.5 Unreal Engine Behavior Trees .............................................................................................. 33
    2.6.6 Summary and Comparison .................................................................................................. 35
3 Behavior Trees with MindSet ........................................................................................................ 37
  3.1 Extensions to the Base Behavior Tree Architecture ................................................................. 37
    3.1.1 Utility Selectors .................................................................................................................... 38
    3.1.2 Random Selectors ................................................................................................................ 41
    3.1.3 Parallel Sequences .............................................................................................................. 41
    3.1.4 Three-Valued Condition Components ............................................................................... 43
    3.1.5 Query Components ............................................................................................................ 44
3.2 Parameterizing Behavior Trees with Execution Contexts ........................................ 46

4 Results and Evaluation ............................................................................................... 49
  4.1 BomberCube Demo ............................................................................................... 49
  4.2 Pocket Critters Demo ........................................................................................... 56
  4.3 Defense Grid Demo ............................................................................................... 62

5 Conclusions and Future Work ................................................................................. 69
  5.1 Summary and Conclusions .................................................................................. 69
  5.2 Future Work .......................................................................................................... 70

References ..................................................................................................................... 74

Appendices .................................................................................................................... 76
  Appendix A. Pseudo-Code, Icons for Decorator Components .................................. 76
  Appendix B. Notation Used in UML Diagrams ............................................................ 79
  Appendix C. MindSet Implementation Details ............................................................. 81
    Appendix C1. MindSet ............................................................................................... 82
      Appendix C1.1. Encapsulating Properties and Metadata ........................................ 83
      Appendix C1.2. Execution Contexts ...................................................................... 87
      Appendix C1.3. Components ................................................................................. 90
      Appendix C1.4. Building a Behavior Tree in Code ................................................. 94
      Appendix C1.5. Managing Multiple Behavior Trees .............................................. 96
    Appendix C2. The MindSet Editor ........................................................................... 98
      Appendix C2.1. Editor Features and Internals ......................................................... 101
      Appendix C2.2. Managing Behavior Tree Data ...................................................... 109
    Appendix C3. The MindSet API .............................................................................. 114
      Appendix C3.1. Attributes and Metadata ............................................................... 115
      Appendix C3.2. Components ............................................................................... 119
      Appendix C3.3. Factories ..................................................................................... 123
      Appendix C3.4. Transforms ............................................................................... 128
      Appendix C3.5. Extension Capabilities .................................................................. 132
      Appendix C3.6. Setting Up AI Code to Work With MindSet Editor Using
                     MindSet API ............................................................................................... 134
    Appendix C4. Reading a MSXP File into Memory .................................................... 137
    Appendix D. Source Code Metrics ......................................................................... 140
List of Tables

Table 1. Examples of actions with utility scorers. ................................................................. 22
Table 2. The behavior in Table 1 modelled as a decision tree ........................................... 24
Table 3. Summary of features in available BT editors ......................................................... 35
Table 4. The selector strategies supplied by MindSet ......................................................... 39
Table 5. Example of how different utility coefficients can be assigned to individual condition checks for utility evaluation of “Gun is not loaded” for different actions. .............................. 40
Table 6. The BehaviorReturnCode aggregator strategies supplied by MindSet .................. 43
Table 7. Number of times individual nodes in MAIN TREE were visited during execution for each entity during a sample run. .............................................................................. 53
Table 8. Number of times individual nodes in MOVE tree were visited during execution for each entity during a sample run .............................................................................. 53
Table 9. Number of times individual nodes in COMPUTE PATH TO RANDOM TILE tree were visited during execution for each entity during a sample run. .................................................. 54
Table 10. Number of times individual nodes in COMPUTE PATH TO NEAREST POWER-UP tree were visited during execution for each entity during a sample run. .......................... 54
Table 11. Statistics gathered from Defense Grid demo runs with the defense grid firing three bullets per charged shot. ............................................................................................... 67
Table 12. The feature comparison table from Table 3, now including MindSet Editor .......... 70
List of Figures

Figure 1.  A behavior tree labelled with the identifier “simple attacker”.................................2
Figure 2.  Pseudo-code and example for the selector component.  Reference components are also depicted........................................................................................................................................9
Figure 3.  Pseudo-code and example for the sequence component.........................................10
Figure 4.  Pseudo-code and example for the inverter decorator decorating a condition check......13
Figure 5.  A simple FSM.............................................................................................................16
Figure 6.  A simple HFSM............................................................................................................17
Figure 7.  The HFSM from Figure 6 has been modified to include an additional “Wait” state. ...20
Figure 8.  The HFSM from Figure 7 is modelled as a set of BTs..................................................21
Figure 9.  Example of an and-or graph. .......................................................................................26
Figure 10. Example of a plan generated by an HTN. ...................................................................27
Figure 11. Screenshot of Behavior Designer while adding a new component (original in color).29
Figure 12. The Behavior3 BT editor (original in color)..............................................................30
Figure 13. The Behavior3 BT editor’s configuration settings (original in color). .......................31
Figure 14. Screenshot of Behave (original in color)..................................................................32
Figure 15. Screenshot of Brainiac Designer (original in color)...................................................33
Figure 16. Screenshot of a BT in Unreal Engine BT Designer (original in color) .........................34
Figure 17. Pseudo-code and example for the utility selector node. ............................................38
Figure 18. Pseudo-code and example for the random selector node..........................................41
Figure 19. Pseudo-code and example for the parallel sequence node........................................42
Figure 20. Pseudo-code and example for the query node..........................................................45
Figure 21. A component modifies data within the execution context during BT traversal. ....47
Figure 22. Screenshot of Bomberman Live gameplay (original in color) (Wong, 2007). .........49
Figure 23. Screenshot of the BomberCube demo. ......................................................................51
Figure 24. The BTs used for modelling bomber entity behavior..................................................52
Figure 25. A memory allocation histogram that demonstrates that MindSet does not perform continual memory allocations (original in color)................................................................55
Figure 26. Screenshot of Pokémon gameplay (original in color) (pokemonhalloffame’s Bucket, n.d). ..................................................................................................................................................57
Figure 27. Screenshot of the Pocket Critters demo.....................................................................57
Figure 28. The BTs used to control BLUE and its critters in Pocket Critters.............................60
Figure 29. Sample log for a single combat step in Pocket Critters..............................................61
Figure 30. Screenshot of the Defense Grid demo. ................................................................. 62
Figure 31. The BT used for the enemy ship cannon entities. ............................................. 64
Figure 32. The BT and utility score table used for the defense grid entity. ....................... 65
Figure 33. Screenshot of the Defense Grid demo after completion. .................................. 66
Figure 34. Example of explicit conversion used for the Entity and World properties of the execution context. ........................................................................................................ 71
1 Introduction

Behavior trees (BTs) have emerged as a popular tool for modelling the limited form of artificial intelligence (AI) that is present computer games since their use in Halo 2 was first publicized by Bungie Software in 2004. They are also gaining popularity as an effective method for controlling robots (Marzinotto, Colledanchise, Smith, & Ogren, 2014). In the remainder of this thesis, the abbreviation “AI” is used for “game AI”.

A BT is a model for reactive plan execution and is graphically represented as a set of nodes arranged in a tree, such as that shown in Figure 1. Execution of the behavior modelled by the BT begins at the root node and proceeds using pre-order depth-first search. Execution of the behavior is performed during each of the game’s AI processing steps.

We begin by discussing the motivation for the thesis research, followed by an outline of the problem being solved. Original contributions to the field are then presented, followed by a synopsis of the remainder of the thesis.
Figure 1. A behavior tree labelled with the identifier “simple attacker”.
1.1 Motivation

Developing an appropriate software architecture for effectively modelling AI for a computer game engine is often difficult and expensive. The main challenge in designing such an architecture is that it should not depend on the particular AI code it is going to execute. The game engine will likely be used for a series of games and the detailed requirements for the behaviors of AI entities cannot be predicted at the time the AI-modelling architecture is designed. Therefore, a game-independent AI-modelling architecture is desirable.

The second challenge when using an AI-modelling architecture is that modelling behaviors displayed by game entities can be difficult: a wide variety of new behaviors need to be created and many existing behaviors may need to be changed during the design process due to shifting requirements. To remain productive, developers need to shorten the feedback loop so that they can view the effects of their changes and quickly respond to any problems. They must also structure their code such that it is easy to modify and maintain. The price of not doing so is decreased productivity and a code base that is resistant to change.

In practice, our research is tailored toward helping developers spend less time writing code to support the execution of the AI code and more time writing the actual AI code that will be executed.
1.2 Problem Statement

The goal of this research is to design, implement, and evaluate a software architecture for modelling BTs that is both flexible and extensible. To be flexible, it should allow many different BTs to be expressed using the architecture’s base components. To be extensible, it should allow developers to create new components suited to their needs. We aim to demonstrate the usefulness of fundamental BT concepts while improving upon features common to existing architectures.

1.3 Contributions

The major contribution of this thesis is the development of utility selector components in conjunction with utility coefficients. These are used to model heuristic reasoning within our BT architecture: the utility selector components model prioritization of sub-behaviors and the utility coefficients are weights that affect that prioritization.

Another contribution of this research is the creation of the MindSet AI-modelling software architecture for creating and modifying BTs and the development of MindSet Editor. MindSet Editor displays a BT as a tree with accompanying contextual information to clearly illustrate the behaviors being modelled. BTs are managed as files within a project, and individual BTs are modified using a point-and-click interface. It is much easier for users to build a BT using a graphical editor than for them to write the BT structure in code.
1.4 Structure of the Remainder of the Thesis

In Chapter 2, we present an overview of fundamental BT concepts along with accompanying examples. We compare BTs to other software architectures commonly used for developing code for AI entities: finite state machines, hierarchical state machines, and heuristic reasoning. Existing BT architectures are shown, followed by a comparison of features in their editing software. We use these examples to outline a basic set of features that are important for a graphical interface users would use when building BTs.

In Chapter 3, we introduce the MindSet BT architecture. We begin by discussing extensions made to the base BT architecture, including a novel utility-based AI extension. Next, we show our method for parameterizing BTs because it is central to MindSet.

In Chapter 4, we show three simple games we have developed to showcase different aspects of the MindSet architecture. BomberCube demonstrates BT parameterization, Pocket Critters describes the use of an inter-BT communication system for issuing orders, and Defense Grid shows the use of our utility-based BT extension. Note that although we focus on the application of BTs in game software, they can be applied to any computer program requiring a model for AI.

We close by summarizing the results obtained via this research project and outline possibilities for future research in Chapter 5.
2 Behavior Trees

In this chapter, we give a detailed explanation of BTs and how they are used to model behaviors for entities. An entity is anything that will be controlled by a BT.

Section 2.1 outlines components, the fundamental concept of BTs. The main building block of a BT is a component, which is represented as a node in the graph. Components are granular and can be used to specify complex high-level behaviors via composition. The topology of a BT is exactly the same as a tree in graph theory: components have at most a single parent, and no component can be the root node’s parent. The different component types are explained alongside accompanying examples. In Section 2.2, we walk through the process of building a simple BT using those components.

In Section 2.3, we compare the BT architecture and finite state machines (FSMs). FSMs are another popular method for modelling AI. In fact, BTs were developed within the computer gaming community as a more modular alternative to FSMs (Colledanchise & Ogren, 2014). Hierarchical finite state machines (HFSMs) are an extension of FSMs in which individual states can contain more granular FSMs. We give an example of a HFSM, and then compare the BT and HFSM architectures with regards to the following questions: How ‘easy’ is it to develop AI for entities using the architecture? How much maintenance effort is required? Does the architecture scale well for handling large, complex behaviors? Can behaviors previously developed be incorporated into new behaviors? Can the architecture be easily extended to incorporate user-defined concepts?
We discuss the limitations of the BT architecture when modelling behaviors that use heuristic reasoning to model intuition or prioritization in Section 2.4. We give an example of an entity whose behavior is governed by heuristic reasoning and outline the difficult involved with converting that logic for use within the BT architecture.

In Section 2.5, we survey the field of available BT architectures. Brief overviews of the architectures are given. We conclude the chapter by identifying features that are desirable in a new approach.

2.1 Fundamental Behavior Tree Concepts

Every component in a BT possesses the same basic structure: it is given some CPU time to do something and then it returns a status code to its parent component upon completion. The status codes are SUCCESS, FAILURE, RUNNING, and ERROR. SUCCESS indicates that the task performed by the component completed successfully. If something prevented the task from being completed, FAILURE is returned. If the task performed by the component did not complete and possibly requires more AI steps to finish, RUNNING is returned. ERROR indicates that an exceptional error occurred while attempting to complete the component’s task, likely due to some programming error such as trying to access an undefined variable. This status code is for debugging purposes only. To simplify discussion, we omit further mention of the ERROR status code.

While it is possible to define a myriad of arbitrarily complex component types, complex BTs can be built by using only the following fundamental component types: the reference, the condition, the action, the control flow node, and the decorator. After
describing these five types of components, we explain why these five types are sufficient to specify all types of behaviors.

The root node of a BT is a component. Therefore, the BT itself returns one of the status codes. It follows that a new BT can be constructed from a set of other BTs by adding a new root node that links the other BTs. This linking is accomplished using *reference components*. A reference component is a component that refers to another BT.

Reference components enable a crucial capability of BTs: the ability to build complex behaviors using a set of simpler sub-behaviors. A reference component returns the status code produced by executing the BT to which it is linked. A reference component is represented by a double-boxed node labelled with an identifier, as shown in Figure 2.

A *condition component* stores a Boolean statement that evaluates to either true or false. These include tests for proximity (“Am I near the player?”,” “Is the player within my line of sight?”), tests on game entity state (“Am I low on health?”, “Am I low on ammo?”), and so on. If the Boolean statement evaluates to true, then SUCCESS is returned as the status code; otherwise, FAILURE is returned. It is not possible for RUNNING to be returned. A condition is represented by a node labelled with a question.

An *action component* alters the state of an entity by performing an *action*. Examples of actions include moving the entity, changing the game entity’s internal state, playing a sound, or executing some specialized logic such as pathfinding. If the action was successfully completed, then SUCCESS is returned. If the action was completed but not successfully, FAILURE is returned. If the action requires additional processing
beyond the current AI step, the RUNNING code is returned instead. An action component is represented by a node labelled with a statement.

The type of a component determines where it may appear within a BT. Both conditions and actions can only be leaf nodes within a BT. Non-leaf nodes are control flow components that group a set of child components and their behavior is determined by status codes returned by those children. Only two control flow component types are needed to express most grouping behaviors: the selector and the sequence.

A selector is a fundamental control flow component type. When executed, a selector processes its child components from left to right and immediately returns a SUCCESS code if one of its children returns a SUCCESS code. It returns a RUNNING code if one of its children returns a RUNNING code. If a child returns FAILURE, the selector continues and processes the next component. If each of the selector’s children returns a FAILURE code, then the selector returns a FAILURE code. A selector node is depicted as a circle labelled with a question mark, as shown in Figure 2. The selector is analogous to logical-OR.

```
foreach child in children do
    childStatus ← execute(child)
    if childStatus == RUNNING
        return RUNNING
    else if childStatus == SUCCESS
        return SUCCESS
end-foreach
return FAILURE
```

Figure 2. Pseudo-code and example for the selector component. Reference components are also depicted.
Selectors are used to choose one branch of the BT from a set of possible branches. For example, consider a selector used to specify an enemy entity’s behavior when its goal is to engage the player in an attack. The enemy entity can attack at close-range with a sword, attack at long-range with a bow and arrow, or simply taunt the player. Figure 2 depicts a BT that models this behavior. When this BT is executed, the selector will first attempt the “Attack player with sword” action; if that action fails, it will then attempt the “Fire arrow at player” action. If the action is successful, processing stops and control is returned to the selector node, which returns a SUCCESS code. The “Taunt the player” action is not attempted because one of its sibling components returned SUCCESS.

A sequence is another fundamental control flow component type. When executed, a sequence processes its child components from left to right and immediately returns a FAILURE code if one of its children returns a FAILURE code. Similarly, it returns a RUNNING code if one of its children returns a RUNNING code. If a child returns SUCCESS, the sequence continues and processes the next component. If each of the sequence’s children returns a SUCCESS code, then the sequence returns a SUCCESS code. A sequence node is depicted as a box labelled with an arrow, as shown in Figure 3. The sequence is analogous to logical-AND.

```plaintext
foreach child in children do
    childStatus ← execute(child)
    if childStatus == RUNNING
        return RUNNING
    else if childStatus == FAILURE
        return FAILURE
end-foreach
return SUCCESS
```

Figure 3. Pseudo-code and example for the sequence component.
Sequences are used to specify a series of sub-tasks to be completed. For example, the “Attack player with sword” reference component in Figure 2 is shown as the BT depicted in Figure 3. When this BT is executed, the “Player in front?” condition is evaluated first. If that condition evaluates to true, then processing continues; otherwise, the condition node returns FAILURE to the parent sequence node and the sequence returns FAILURE. The “Swing sword” action is executed next. Due to how sequence processing works, the sequence node will return the same status code that the “Swing sword” action returns. This is because a sequence aborts execution upon receiving a FAILURE or RUNNING code from one of its children and the “Swing sword” action is the last component in the sequence. The sequence returns SUCCESS if the “Swing sword” action returns SUCCESS.

A decorator is applied to exactly one component and modifies its processing logic. Any component can have decorators applied to it. A decorator is depicted as an icon in the top-left of a node. Each type of icon represents a different type of decorator, as shown in Figure 4. The four types of decorators of interest in this thesis are return-code decorators, repeater decorators, count-based limit decorators, and timer-based limit decorators.

Return-code decorators override a component’s behavior return code. Examples of return-code decorators include inverters, succeeders, and failers. An inverter is analogous to the NOT operator in a programming language, negating the result of the component it is applied to: SUCCESS becomes FAILURE, and FAILURE becomes SUCCESS. The inverter does not modify the component’s return code if the component
returns RUNNING. Succeeders and failers are very simple: the succeeder always returns SUCCESS and the failer always returns FAILURE.

Repeater decorators are used to execute loops of component behavior via iteration. A basic repeater decorator will repeatedly process its wrapped component a specified number of times, regardless of whether the child returns SUCCESS or FAILURE. After the number of repetitions has been completed, it returns SUCCESS. Two more complex looping structures are repeat-until-SUCCESS and repeat-until-FAILURE. These decorators will repeatedly process its wrapped component until receiving the corresponding return code; the decorator will then return SUCCESS. Any repeat-until decorator can also specify the maximum number of times its wrapped component is repeated. If the maximum number of repetitions is reached without its child returning SUCCESS, it returns FAILURE.

A count-based limit decorator imposes a maximum number of executions its wrapped component can have within the whole execution of the BT for a specific entity; that is, after a certain number of calls, its wrapped component will never be executed again. If the wrapped component has not been executed the maximum number of times, then the decorator returns the same code returned from processing the wrapped component. If the wrapped component has been executed the maximum number of times, then the wrapped component is not processed and the decorator returns FAILURE.

A timer-based limit decorator forces a certain amount of game time to pass between executions of its wrapped component. During each AI step, a timer ticks up. The units of measurement used for game time is game-dependent. For example, a turn-based game could increment the timer by one to signify a new turn cycle, while a real-
time strategy game could increment the timer by how much real time in milliseconds has passed since the previous AI step. Once a certain amount of time has elapsed and processing reaches the decorator, the decorator will execute its wrapped component and return that component’s result. Until that amount of time has elapsed, it will return RUNNING if the decorator is visited during traversal.

```plaintext
childStatus ← execute(child)
if childStatus == SUCCESS
    return FAILURE
else if childStatus == FAILURE
    return SUCCESS
return childStatus
```

Figure 4. Pseudo-code and example for the inverter decorator decorating a condition check.

A single component can have multiple decorators applied to it.

Psuedo-code and icons for each of the decorator types listed above can be found in Appendix A.

Recall our assertion that actions, conditions, decorators, control flow nodes, and references are sufficient for specifying all types of behaviors. We use the Böhm-Jacopini theorem as the basis of an informal proof of this claim. The theorem states that we can compute any computable function by combining subprograms using only three specific methods: (1) executing one subprogram and then another subprogram in sequence; (2) executing one of two subprograms selected according to the value of a Boolean expression; and (3) repeating the execution of a subprogram until a Boolean expression is
true. In our case, a subprogram is a BT. We can execute one BT and then another by using a sequence component, which is analogous to (1). Similarly, we can execute one BT from a set of two BTs by using a selector component, which models (2). Finally, we can use a repeat-until-SUCCESS decorator to execute a BT until a Boolean expression is true, which fulfills (3). Since the three methods listed by the theorem can be represented using BT components, BTs can be used to specify any computable behavior.

2.2 Building a Behavior Tree

As an example, we wish to model the following behaviors for a swordsman entity that will battle against the player: FIND AID, EVADE, ATTACK, and WANDER. FIND AID has the swordsman seek a first aid kit if it has lost health points (HP). The first aid kit will restore its HP. EVADE has the swordsman attempt to dodge an attack from an enemy entity. The swordsman dodges by jumping back from the attacker until at a safe distance. Once at a safe distance, it will watch the attacker. If any enemies are within range, the swordsman will display the ATTACK behavior. When not performing any of the above tasks, the swordsman will wander around aimlessly.

We can build a behavior tree for the swordsman entity using the component types described above. There are four tasks to perform and each of them can be encapsulated in its own BT. The low-level steps for performing the task are given from the first-person perspective, followed by a description of the BT structure used.

EVADE behavior states “If I am being attacked, I will move away from the enemy attacking me until at a safe distance. Once at a safe distance, I will watch that enemy for further attacks.” This task is more complex. Its root is a sequence. It
evaluates a check against attacks from the enemy; if true, control passes to a selector. The selector’s first child is a sequence that consists of the following subtasks: check if the enemy is close enough to successfully hit the swordsman; then, if true, then the swordsman will move away from that enemy until at a safe distance. The selector’s second child is a single action: watch for further attacks.

FIND AID behavior states “If I have low HP, then I will find the nearest first aid kit and pick it up.” This is a sequence: a condition that, if true, is followed by an action.

ATTACK behavior states “If an enemy is nearby and not attacking, then I will attack them with my sword.” This is a sequence: a condition that, if true, is followed by an action.

WANDER is simple: “I walk around.” This is a sequence of two actions: choosing a random location within a short range and then walking toward it.

The above four BTs are grouped via a selector node ordered from left-to-right as ordered above. The result is the BT depicted in Figure 1 in the thesis introduction.

2.3 Comparisons to Hierarchical Finite State Machines

Finite state machines (FSMs) and hierarchical finite state machines (HFSMs) are two other techniques used for modelling game AI. A FSM models behaviors using a set of states and a set of transitions between them. Using the terminology of graph theory, a FSM is represented as a network where each vertex represents a state and each edge represents a transition between states. Each edge is labelled with the condition that an entities must fulfill in order to perform the state transition. An example FSM is given in Figure 5.
In this example, the states are WATCH and MOVE AWAY. The arrow on top represents a transition that says “if the entity is in the WATCH state and the ‘player is close’ condition evaluates to TRUE, then the entity will change to the MOVE AWAY state”.

A HFSM extends the FSM architecture by allowing an individual state to be a “super-state” that represents another HFSM. For example, in Figure 6, the EVADE state is a super-state that represents the FSM shown on the right. When an entity transitions to the EVADE state in the HFSM on the left, processing continues starting at the MOVE AWAY state in the FSM on the right. When evaluating transition conditions, the parent HFSM transitions for the EVADE state are evaluated first: namely, ‘HP is low’ and ‘Player is idle’. If neither of those transition conditions evaluate to TRUE, then we evaluate the transition conditions within the EVADE FSM: either ‘Player is far away’ or ‘Player is close’ depending on the entity’s current sub-state.
Figure 6. A simple HFSM.
We assert that BTs outperform HFSMs in five key areas: maintainability, scalability, reusability, extensibility, and prioritization. Our rationale follows.

Changing the behavior modelled by a HFSM requires adding new states and transitions or removing them. The interconnectedness of states means that a ‘simple’ change could actually require a major restructuring of the HFSM: transitions to a new state from existing states must be defined, and invalid transitions to a deleted state must be removed. This process is susceptible to error, especially with larger HFSMs. BTs are not constrained by any inter-node dependencies because a BT is defined by structure rather than by state. Adding or removing nodes in a BT does not require any modifications to other nodes; that is, branches within a BT can be modified independently of other branches.

All issues described for maintaining HFSMs apply when discussing scalability. Graphically, as more states and transitions are added, the HFSM may potentially turn into a mess of boxes and arrows (Isla, Building a Better Battle: The Halo 3 AI Objectives System, 2008) as depicted in Figure 7 below. In contrast, assuming connections to other BTs due to references are not shown and the references do not result in circular dependencies, a BT can always be drawn as a planar graph; that is, it can be drawn in such a way that no edges cross each other. Additionally, when a BT has many nodes, it can be decomposed into smaller separate subtrees. This enhances the readability of the graph.

While it is possible to reuse super-states across HFSMs, there still exists strong coupling between states because the logic used to handle transitions is intrinsic to the
HFSM. This can make it difficult to reuse the same behavior across multiple projects. For example, a transition condition depending on a set of game-specific variables prevents the transition condition code from being used in a different project without having to modify the transition condition code. Reusability is inherent to the BT architecture because of the independence between branches.

The control flow processing in HFSMs is done by the states themselves, which does not lend itself to an extensible architecture. Conversely, the BT architecture is extensible, as demonstrated in Section 3.1.

The HFSM model as currently described does not depict prioritization of transitions, whereas the prioritization of sub-behaviors within a BT is explicitly defined by the control flow components used and their relative positions to each other. This makes it easy to determine the high-priority sub-behaviors when looking at a BT: they are always located on the left-hand side of a given branch in the BT because components are processed from left to right.

Given the above, it is clear that a BT provides a more flexible architecture than an HFSM for representing AI. The diagrams corresponding to BTs also yield more detailed information to the reader.

Figure 7 and Figure 8 show models of a behavior using the HFSM and BT architectures, respectively.

Figure 7 is an extended version of the HFSM shown in Figure 6 that has been modified to include a new WAIT state. Adding the new state to the HFSM required adding new transitions to and from existing states. In contrast, performing the same
Modification on the BT only required the addition of a new sub-tree at the top level with no effect on existing sub-trees.

Also note that the graph in Figure 7 representing the HFSM is no longer planar. In contrast, Figure 8 is a planar graph.

Additionally, the prioritization of sub-behaviors is clearly defined via the BT’s structure in Figure 8: the selector located at the root of the complex attacker BT processes children from left to right. The transition labels in Figure 7 require descriptive labelling than is provided in order to determine priority amongst available transitions for a given state. For example, if the entity is in the EVADE state and both ‘Player is out of sight’ and ‘HP is low’ transition conditions are true, the appropriate transition is not explicitly defined. The ability to intuitively identify priorities among sub-behaviors is another important property of BTs compared to FSMs (Cutumisu & Szafron, 2009).

![Diagram](image.png)

Figure 7. The HFSM from Figure 6 has been modified to include an additional “Wait” state.
2.4 Comparison to Heuristic Reasoning Methods

A heuristic is a “rule of thumb for reasoning” that “reduces or limits the search for solutions in domains that are difficult and poorly understood” (Heuristics and Heuristic Evaluation, n.d.). Unlike formal structures like algorithms, heuristics do not necessarily guarantee optimal solutions.
In the context of game AI, a *heuristic function* is a function that calculates the utility of performing some action given the current world state. Heuristic functions often use utility to evaluate preferences and priorities. We define *utility* as a measure of preference over some set of actions that can be performed by the entity given the current world state. Utility is represented as a numeric value that is the sum of individual values returned from *utility scorers* (Rasmussen, 2016). A utility scorer analyzes some portion of the game state and returns a score. Examples of actions with utility scorers are given in Table 1.

<table>
<thead>
<tr>
<th>Action</th>
<th>Scorer</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE TO ENEMY</td>
<td>Distance to enemy</td>
<td>0 – 100</td>
</tr>
<tr>
<td></td>
<td>Gun is not loaded</td>
<td>-100</td>
</tr>
<tr>
<td></td>
<td>Proximity to enemy &lt; 50</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>Cannot make it to cover</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Gun is not loaded</td>
<td>-125</td>
</tr>
<tr>
<td>FIRE AT ENEMY</td>
<td>Is not in cover</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Proximity to cover &lt; 50</td>
<td>50</td>
</tr>
<tr>
<td>MOVE TO COVER</td>
<td>Gun is not loaded</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>Is in cover</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Gun is loaded</td>
<td>-125</td>
</tr>
</tbody>
</table>

Although a BT may provide a better technique for organizing behaviors than an FSM, it does not necessarily provide a model for better decision-making. Decisions about which sub-behaviors to process are made based upon fixed binary conditions or control flow components in a BT. Thus, a BT may be unable to model heuristic decision-making effectively due to the BT structure’s rigidity.

To demonstrate this limitation of the base BT architectures, we will model the behavior described by Table 1 using BT components discussed thus far in this thesis. There are four actions given: MOVE TO ENEMY, FIRE AT ENEMY, MOVE TO
COVER, and LOAD. Each of these actions relies upon some set of these four input parameters: the entity’s distance to the enemy in meters, the entity’s distance to cover in meters, whether or not the entity is currently in cover, and whether or not the entity’s gun is loaded. We must also account for magnitudes in the distance parameters. To provide for different numeric ranges for the ‘distance to enemy’ parameter, we use a set of binary decisions instead: “am I within 1m?”, “am I within 10m?, and “am I more than 10m away?” If we also use these same magnitudes for the ‘distance to cover’ parameter, then we will produce a decision table with 36 different action branches, as pictured in Table 2.

Modelling this decision table as a BT yields a BT with 36 leaf nodes. The resulting BT is difficult to maintain because similar information is replicated several times. The BT is also difficult to change if another conditional check needs to be added because information must be added in many subtrees. This example demonstrates that the base BT architecture is not suitable for modelling behaviors that use heuristic reasoning.
Table 2. The behavior in Table 1 modelled as a decision tree.

<table>
<thead>
<tr>
<th>Gun loaded?</th>
<th>In cover to enemy?</th>
<th>Distance to enemy? ((d_e))</th>
<th>Distance to cover? ((d_c))</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &gt; 5\ m)</td>
<td>Move to enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(d_e &gt; 10\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Move to enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Move to enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(d_c &gt; 5\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(d_c &lt; 1\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &gt; 5\ m)</td>
<td>Move to enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Move to enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(d_c &gt; 5\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Fire at enemy</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &gt; 5\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(d_e &gt; 10\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(d_c &gt; 5\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>(d_e &lt; 1\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &gt; 5\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &gt; 10\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &gt; 10\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(d_c &gt; 5\ m)</td>
<td>Load</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(10\ m \geq d_c \geq 1\ m\</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(5\ m \geq d_c \geq 1\ m)</td>
<td>Move to cover</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>(d_e &lt; 1\ m)</td>
<td>(d_c &lt; 1\ m)</td>
<td>Move to cover</td>
</tr>
</tbody>
</table>
2.5 Comparison to Artificial Intelligence Formalisms

In this section, we compare BTs to two similar artificial intelligence formalisms. They are and-or graphs and hierarchical task networks.

An and-or graph is a graphical representation of how a complex problem can be reduced to conjunctions and disjunctions of sub-problems (Luger & Stubblefield, 1998). Conjunctions correspond to logical-AND operations and disjunctions correspond to logical-OR operations.

A node with children having joined arcs in an and-or graph is a success node if all its children are success nodes; otherwise, the parent node is a failure node. This is analogous to the sequence component in a BT that returns SUCCESS if all its child components return SUCCESS. A node with children having separate arcs in an and-or graph is a success node if at least one of its children is a success node; otherwise, the parent node is a failure node. This is analogous to the selector component in a BT that returns SUCCESS if one of its child components returns SUCCESS.

A leaf node in an and-or graph is considered a success node if a method exists for solving the problem associated with that node. Conversely, a leaf node is considered a failure node if a method does not exist for the solving the problem associated with that node. These are analogous to action and condition components in BTs, which return SUCCESS if the action successfully completed or if the condition check evaluated to TRUE.

Unlike a BT, which has a deterministic traversal order, an and-or graph simply defines a search space for a given problem; it does not specify how to traverse that search space. Search strategies such as depth-first, breadth-first, or best-first can be used to
traverse the search space. An example of an and-or graph is given in Figure 9. In this example, a high-level problem A is solved using a set of sub-problems B through F. B is solvable if either E or F is solvable. A is solvable if B and C is solvable, or if D is solvable.

![And-or graph example](image)

Figure 9. Example of an and-or graph.

BTs are also similar to plans generated by *hierarchical task networks* (HTNs). An HTN is an approach to automated planning that takes a high-level problem statement as input and returns a *plan* that solves that problem with respect to the current world state (Humphreys, 2014). A plan is a series of tasks to perform that are arranged in a hierarchy. Each task can be a *primitive action* or a *compound action*. A primitive action is a single action that can be performed; that is, the action cannot be decomposed into smaller sub-actions. A compound action consists of multiple primitive actions or other compound actions.

Figure 10 shows an example of a plan generated by an HTN. This plan allows a gunman to perform a high-level task described as “attack enemy squad”. Per the pictured
plan, the gunman will perform first the “reload gun” and “alert other squad members” tasks in parallel. Once other squad members have been alerted, then the “seek cover” task is performed. After all three of those tasks have been completed, the gunman will then proceed to perform the “fire at enemy squad” task. The plan has been executed in full after the completion of that final task.

Figure 10. Example of a plan generated by an HTN.

While BTs are imperative in nature, HTNs are declarative; that is, BTs are just a set of commands to execute, while HTNs use a planning algorithm that can read from a set of declarative rules that specify dependencies between tasks. Given a goal, the planning algorithm reads the declarative rules and then it either yields a plan or fails. The plans generated by the planning algorithm act much like a BT.
2.6 Overview of Existing Behavior Tree Editors

A few BT editors already exist with varying feature sets. A few of the more popular ones are discussed here, namely Behavior Designer, Behavior3, Behave, Brainiac Designer, and Unreal Engine Behavior Trees. Screenshots of the editors are included, along with details surrounding their user interfaces.

2.6.1 Behavior Designer

Behavior Designer is a visual BT editor specific to the Unity game development platform. It has “a powerful API [that] gives plenty of freedom for programmers” and boasts “exceptional performance with zero runtime allocations after initialization” (Mosiman & Watson, 2014). Due to full integration with the Unity game development environment, it is possible to debug BTs within the editor. The debugger supports breakpoints, watched variables, and the ability to peek at task execution status. It is possible for BTs to be aborted based on external events. It integrates with a variety of other third-party plugins within the Unity ecosystem.

Behavior Designer uses icons and labels to distinguish between individual components. Additionally, comments can be added to components that provide contextual information. Parent components are always located above their child nodes, but not necessarily centered over them. Components are added to BTs via the use of a context menu that appears via right-click.

The UI is fairly involved with a set of tabs in the top-left that determine the contents of the left pane, allowing users to show behaviors currently constructed via Behavior, a list of components to add to the BT currently open via Tasks, a set of
variables that can be manipulated via Variables, and a debugging pane via Inspection. The UI for Behavior Designer is pictured in Figure 11.

![Behavior Designer UI](image)

**Figure 11.** Screenshot of Behavior Designer while adding a new component (original in color).

### 2.6.2 Behavior3

Behavior3 is an open-source visual BT editor (Pereira, 2015). BTs are imported and exported using JSON (JavaScript Object Notation) format. Users can extend the architecture by creating their own components. Components displayed within a BT can be auto-organized. This editor does not depend on other tools, editors, or engines.

Behavior3 has a more simplistic UI compared to Behavior Designer as pictured in Figure 12. Condition checks are denoted by circle nodes labelled with a question and actions are denoted by rounded rectangles labelled with a statement. Selectors, sequences, and other control flow components are identified via icons rather than text. Although this representation is compact, it sacrifices some readability since users must familiarize
themselves with the component icons. A list of BTs contained within the current project is displayed in the top-left corner. A list of components that can be added to the BT currently being viewed is located below the list of BTs.

Figure 12. The Behavior3 BT editor (original in color).

Behavior3 Editor is also customizable, allowing users to configure placement-grid dimensions, component layout, and colors. The customization screen is shown in Figure 13. The component layout can be set for either vertical or horizontal layout. A vertical layout displays the BTs in the same manner as we have done thus far: parent components are located above their children, and children are executed from left to right. A horizontal layout has parent components placed to the left of their children, and children are executed from top to bottom. These customizations enable users to optimize their BT workspace. It is also possible to resize the backing grid via Snap X and Snap Y to affect
placement of individual components, and can modify the color scheme of the interface to suit the user’s preferences.

Figure 13. The Behavior3 BT editor’s configuration settings (original in color).

2.6.3 Behave

Behave is another visual BT editor specific to the Unity game development platform (Johansen, 2016). The suppliers of the editor claim there is “no heavy API requiring [the user] to jump through hoops to integrate with [the user’s] code”. Like Behavior Designer, it also sports a debugger due to integration with Unity. Behave follows many of the same UI patterns and work flows as the Behavior3 BT editor, as pictured in Figure 14.
2.6.4 Brainiac Designer

Brainiac Designer is an open-source visual BT editor (Brainiac Designer, 2009). BTs are stored as XML files, but they can also be exported as PNG image or EMF vector image formats for illustration purposes. The user must implement all components themselves and must write an exporter to generate a file containing the BT structure in the format required for the user’s project. A screenshot of Brainiac Designer is shown in Figure 15.
Brainiac Designer uses Windows Form controls for its UI. The application window is split into panes that can be docked within the window itself or pulled out and floated around the user’s computer screen. This allows users to customize their workspace to suit their needs and also enables the viewing of multiple BTs at once. BTs are arranged using a directory structure like code projects in Visual Studio.

### 2.6.5 Unreal Engine Behavior Trees

Unreal Engine Behavior Trees are a proprietary BT architecture with an accompanying editor specific to the Unreal game engine (Epic Games, n.d.). It uses
*event-driven BTs*: instead of running the BT during each AI step, it is instead only executed when an event occurs that the game entity is passively listening for. Therefore, BTs created by this editor and used within Unreal Engine at runtime support dynamic aborting of BTs based on external events. BT execution can then be resumed from any component in the BT. A debugging suite is also provided.

Unreal Engine BT Designer follows the same UI principles as Behavior Designer. A screenshot of Unreal Engine BT Designer is shown in Figure 16. The major difference is that individual components are large enough to display more comprehensive information. This enhances readability of the BT’s functionality. Decorators are
displayed above the component they decorate and are grouped together by a gray border. The decorated component can then be moved around as if it were a single unit.

2.6.6 Summary and Comparison

A summary of features included in the editors discussed previously is given in Table 3.

Table 3. Summary of features in available BT editors.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Behavior Designer</th>
<th>Behave</th>
<th>Behavior3</th>
<th>Brainiac Designer</th>
<th>Unreal Engine BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamental BT components (covered in Section 2.1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Manage collections of BTs</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Drag-and-drop interface</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can auto-arrange components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Add comments to provide context for individual components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Extensible via user components</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Event-driven BTs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Requires specific development environment</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT-specific debugger included</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional aborts and restarts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Utility selector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As shown in the table, the editors all manage BTs as a collection; that is, multiple BTs can be open and being edited within the editor at the same time. A drag-and-drop system is available in all the editors for constructing BTs within a UI. Under such a system, users are free to move individual components around the screen using the mouse. Of the editors, only Behavior3 and Unreal Engine BTs possess the ability to auto-arrange components into an aesthetically-pleasing layout. The auto-arrange feature can be especially helpful when dealing with large, complex BTs. When performing auto-
arranging, a strict hierarchy is enforced by the placement of components: parent components are always placed above their children. Behavior Designer also provides users with the ability to add comments to individual components to provide context to specific sub-behaviors. Only Brainiac Designer and Unreal Engine BTs allow the user to extend the basic architecture by creating their own component types. In fact, utility selectors are implemented in Unreal Engine BTs via a third-party plugin.

Behavior Designer, Behave, and Unreal Engine BTs are tied to specific development environments. While there is power in doing so, we wish to remain agnostic to development environments. Integration with other pieces of software is beyond the scope of this thesis.
3 Behavior Trees with MindSet

The implementation of the BT architecture is holistically known as MindSet and is divided into three parts: the MindSet component class library, MindSet Editor (MSE), and the MindSet API (MSAPI). Although the implementation details for these parts are covered in the next chapter, we will first discuss important innovative aspects of the MindSet BT software architecture design.

MindSet extends the base BT architecture to provide increased functionality while avoiding any dependencies on a specific development environment. We discuss the extensions first in Section 3.1. We introduce new components used for modelling heuristic reasoning. We also introduce new components to provide some control flow structures that are common to AI behaviors but not present in the base BT architecture.

In Section 3.2, we describe how game entity data is modified during BT traversal, and discuss the benefits of a parameterized approach versus instantiating copies of BTs that are used by the same types of game entities.

3.1 Extensions to the Base Behavior Tree Architecture

We provide five extensions to the base BT architecture. Briefly, they are utility selectors, random selectors, parallel sequences, three-valued condition components, and query components.
3.1.1 Utility Selectors

The first extension to the base BT architecture is the utility selector component. It adds heuristic reasoning capabilities. Recall our definition of utility: it is a measure of preference over some set of actions that can be performed by the entity given the current world state. A utility score is a numeric representation of that measurement: a high utility score signifies high preference for that action and a low utility score signifies low preference.

```plaintext
highestScore ← -INFINITY
results ← empty list

// compute utility scores
foreach child in children do
    score ← utility(child)
    if score > highestScore
        results ← empty list
        results.Add(child)
    else-if score == highestScore
        results.Add(child)
end-if
end-foreach

// choose child with high score
// settle ties via strategy
childToExecute ← pickChild(results)

// process that child
status ← execute(childToExecute)
return status
```

Figure 17. Pseudo-code and example for the utility selector node.

The algorithm for the utility selector is given in Figure 17. The utility selector must have at least one child component, with each child representing some sub-behavior to perform. Every child component is required to specify a set of one or more utility score evaluation methods, or scorers. For example, the ‘Move to enemy’ action has two
scorers defined: ‘Distance to enemy’ and ‘Gun is not loaded’. When a utility selector is processed, it first computes the utility scores of all child components. The utility score of a child component is the sum of all values returned by scorers defined for that child. The child components with the highest score are stored in a list, and then one of them is selected for execution.

We introduce the idea of a selector strategy for a utility selector component in a BT. A selector strategy allows a user to customize how the utility selector behaves when multiple child components are tied for the highest utility score. This idea is represented by the pickChild function call in Figure 17. The selector strategies supplied by MindSet are shown in Table 4. All selector strategies take a list of child components and return a single component from that list. It is also possible for developers to define novel selector strategies in their own AI code when the above strategies are insufficient. The novel selector strategies can then be used within MindSet.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectFirst</td>
<td>The left-most child component with the highest score is chosen.</td>
</tr>
<tr>
<td>SelectLast</td>
<td>The right-most child component with the highest score is chosen.</td>
</tr>
<tr>
<td>SelectRandom</td>
<td>A random child component with the highest score is chosen.</td>
</tr>
</tbody>
</table>

MindSet provides three utility-based components: utility action, utility decorator, and utility reference. They are extensions of the existing action, decorator, and reference components of the base BT architecture, which were discussed in Section 2.1. The utility action component executes some behavioral action defined in AI code. The utility decorator is a pass-through component that enables the assignment of a utility value to a branch in a BT. In this case, the decorator simply returns the behavior return code of its
decorated component. Finally, the utility reference encapsulates a reference to another BT.

Each of the utility-based components requires some collection of scorers to be specified. The sub-tree of a utility-based component will be processed if its scorers produce the highest score among all utility-based components being evaluated.

Utility selectors enable the modelling of AI entities that use heuristic reasoning. They can be used to simplify BTs that would otherwise rely on a complex series of binary decisions, such as in the example given previously in Section 2.4. Building a BT for such a series would yield a short and wide graph structure; also, building and maintaining such a BT is likely to be error-prone due to the BT’s complexity. The utility selector reduces this complexity using a single component and a set of utility scoring functions, one for each sub-behavior.

To facilitate reusability of utility scorers, we introduce utility coefficients. The coefficients act as weights that enable the same scorer to return different utility scores for individual behaviors. An example of using utility coefficients to adjust utility scores for different actions is given in Table 5.

Table 5. Example of how different utility coefficients can be assigned to individual condition checks for utility evaluation of “Gun is not loaded” for different actions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Scorer</th>
<th>Score (S)</th>
<th>Coefficient (C)</th>
<th>Total Score (S * C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to enemy</td>
<td>Distance to enemy</td>
<td>0 – 100</td>
<td>1</td>
<td>0 – 100</td>
</tr>
<tr>
<td></td>
<td>* Gun is not loaded</td>
<td></td>
<td>-100</td>
<td></td>
</tr>
<tr>
<td>Fire at enemy</td>
<td>Proximity to enemy &lt; 50</td>
<td>1</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>* Cannot make it to cover</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>* Gun is not loaded</td>
<td>1</td>
<td>-125</td>
<td>-125</td>
</tr>
<tr>
<td>Move to cover</td>
<td>Is not in cover</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Proximity to cover &lt; 50</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Load</td>
<td>* Gun is not loaded</td>
<td>1</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>* Is in cover</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>* Gun is loaded</td>
<td>1</td>
<td>-125</td>
<td>-125</td>
</tr>
</tbody>
</table>
3.1.2 Random Selectors

The second extension is the random selector component. The control flow structures for BTs presented so far use deterministic child branch execution; that is, child components are always processed in the same order, from left to right. The random selector enables developers to mimic non-deterministic behaviors. Instead of processing child components from left to right, the random selector shuffles the components into a random order before beginning processing. The algorithm for the random selector is shown in Figure 18.

```
shuffled ← children.Shuffle()

foreach child in shuffled do
    childStatus ← execute(child)
    if childStatus == RUNNING
        return RUNNING
    else if childStatus == SUCCESS
        return SUCCESS
end-foreach
return FAILURE
```

Figure 18. Pseudo-code and example for the random selector node.

Like the normal selector node, processing stops immediately once a child returns SUCCESS or RUNNING; otherwise, if all children fail, a FAILURE code is returned.

3.1.3 Parallel Sequences

The third extension is the parallel sequence component to allow more efficient processing on parallel processors. With the basic BT architecture, traversal through a BT is limited to visiting a single component at a time before proceeding to the next. This
approach is adequate for most operations. However, suppose that one wishes to perform some computationally expensive operations as part of the behavior being modeled. If these operations are sufficiently expensive, the performance of the entire BT will be degraded because processing must wait until the expensive operation is completed. The parallel sequence component is used for executing multiple branches of a BT in parallel. The pseudo-code for the parallel sequence component is given in Figure 19, along with an example. Unlike an ordinary sequence component, a parallel sequence component will wait for all branches to complete processing before returning control to its parent component. This waiting is necessary because the BehaviorReturnCode results returned from all child components must be accumulated to determine what the parallel sequence component should return to its parent.

We introduce the use of a **aggregator strategy** so that users can customize how the parallel sequence component behaves after processing has been completed on all child components. It is similar to the selector strategy previously described in Section 3.1.1. The aggregator strategies supplied by MindSet are shown in Table 6. It is also
possible for developers to define aggregator strategies in cases where none of the above strategies fulfill their requirements.

Table 6. The BehaviorReturnCode aggregator strategies supplied by MindSet.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All must succeed</td>
<td>If all child components return SUCCESS, then return SUCCESS; otherwise, FAILURE is returned. Analogous to logical-AND.</td>
</tr>
<tr>
<td>At least one must succeed</td>
<td>If at least one child component returns SUCCESS, then return SUCCESS; otherwise, FAILURE is returned. Analogous to logical-OR.</td>
</tr>
<tr>
<td>Majority rules, SUCCESS breaks ties</td>
<td>Return SUCCESS if the majority of child components return SUCCESS or return FAILURE if the majority of child components return FAILURE. In the event of a tie, return SUCCESS.</td>
</tr>
<tr>
<td>Majority rules, FAILURE breaks ties</td>
<td>Return SUCCESS if the majority of child components return SUCCESS or return FAILURE if the majority of child components return FAILURE. In the event of a tie, return FAILURE.</td>
</tr>
</tbody>
</table>

Users of the parallel sequence component must ensure that child branches all act in a read-only fashion on the execution context or ensure that they are modifying different types of data within the execution context. Data integrity is a common concern in multi-threaded programming, so we assume that users will be aware of this constraint and write code for their AI methods accordingly.

3.1.4 Three-Valued Condition Components

The fourth extension is the three-valued condition component. Its functionality is similar to the condition component because if the condition check evaluates to true, then the component returns SUCCESS; and if the condition check evaluates to false, then the component returns FAILURE. However, it also extends the condition component to support a third return value for Boolean expressions: null. If the three-valued condition
check associated with the component returns null, then the component will return
RUNNING to signify an indeterminate state.

The three-valued condition component is useful for modelling scenarios where a
Boolean check requires multiple AI steps to be evaluated. For example, an entity could
visually scan for a few seconds before finally declaring if the player is present via a
SUCCESS code. In this example, the three-valued condition component returns
RUNNING until the visual search is complete, at which point either SUCCESS or
FAILURE is returned.

3.1.5 Query Components

The fifth extension is the query component. The query component is a powerful
extension to the reference component: instead of linking to exactly one BT, the query
component represents a link to one or more BTs matching some criteria.

Using examples, we have shown that it is easy to add new components and new
behaviors to existing BTs due to the nature of the underlying data structure. In fact, each
BT represents an abstraction of some goal to achieve and can be combined with other
BTs; in other words, one goal could have many associated BTs, such that each BT
defines some particular method of achieving the goal.

Suppose that at an early stage in the process of developing a game the AI designer
has accumulated five BTs associated with the ATTACK PLAYER goal. The AI designer
now wishes to design another set of different BTs that also achieves that goal. If all BTs
for all entity types are statically defined and linked into existing BTs via reference nodes,
then all BTs that reference the five BTs used to achieve the ATTACK PLAYER goal
must be manually modified to allow for our new BT. It is obvious that any BT added late in the design phase is not accounted for by BTs developed earlier in the process. It would be preferable if references to the new BT could be automatically inserted into all existing BTs at the proper points.

Query-enabled BTs (Flórez-Puga, Gómez-Martín, Gómez-Martín, Díaz-Agudo, & González-Calero, 2009) have been proposed as a solution to this problem, and we provide the query component as a fifth extension to the base BT architecture. Apparently, none of the behavior tree frameworks surveyed in Section 2.6 provide an implementation of the query component, although they all provide reference components. The query component is another type of reference component that acts as a proxy corresponding to the execution of some task. We represent a query component in our BTs using a double-lined box labelled “??”, as displayed in Figure 20. In this example, the query component references all other BTs related to attacking.

```
// retrieve a collection of BTs from the
// repository with metadata matching some
// criteria
collection ← repository.Query(criteria)

// select a single BT from the result set
// using some filter criteria
behaviorTree ← collection.Select(filter)

// abort if no BT matches the criteria
// otherwise, execute that BT
if behaviorTree == null
    return FAILURE
else
    return execute(behaviorTree)
```

Figure 20. Pseudo-code and example for the query node.

The method used by the query component to select BTs is made possible by specifying some metadata for each BT. The query component is then able to scan the
metadata of all BTs and returns those whose metadata match some criteria. From that set, one BT is selected by a result-filtering method that acts on additional input parameters. The selected BT is then executed as though it were directly referenced.

Unfortunately, the design-time flexibility provided by query components comes at the cost of runtime performance. With a naïve implementation, the query method executed by the query component executes in $O(n + m)$ time, where $n$ is the total number of BTs in the collection and $m$ is the number of BTs returned by the query method. We reduce the runtime cost of this process by replacing $n$ by $n' < n$, where $n'$ is the total number of BTs with a specified BT-type. Each query-enabled BT must specify a BT-type as part of its metadata. BTs are then grouped according to their BT-type, so $n$ becomes $n'$.

### 3.2 Parameterizing Behavior Trees with Execution Contexts

The MindSet BT architecture utilizes a dataset called an execution context to encapsulate an individual game entity’s state. The execution context is always accessible as execution traverses the BT; in other words, code associated with any individual node can access and modify data stored in the execution context. The execution context simplifies using a BT to control the behavior of multiple game entities. Duplicating the BT structure in memory for each game entity is not ideal as we cannot consume all available RAM to store BTs (Isla, Handling Complexity in the Halo 2 AI, 2005). Instead of duplicating the BT across all applicable game entities to perform AI processing, all actions act upon an individual game entity’s state.
The execution context manages a collection of key-value pairs consisting of a unique identifier and an associated value. These key-value pairs correspond to properties of the current game state.

An example operation is illustrated in Figure 21. The game entity’s state consists of two key-value pairs: Position and Orientation, representing the entity’s location within the game world and the direction that they are facing, respectively. Prior to executing the ‘Move’ action, the game entity is located at position 0.0 and is facing to the left. The ‘Move’ action modifies the values of these two pairs: after the ‘Move’ action is completed, the game entity is now located at position 1.0 and is now facing to the right.

![Diagram](image)

**Figure 21.** A component modifies data within the execution context during BT traversal.

The execution context defines the following two key-value pairs by default: 
*Entity* and *World*. Entity is a reference to the game entity that is being controlled by the BT. This allows the BT to query the entity’s state and modify its properties directly. World is a reference to the ‘game world’ object. This object exposes queries about the game state to the BT; that is, it allows the BT code to call functions that return information about the game state.
The execution context for a game entity is external to any BTs and so the entity’s state is preserved between BT traversals. For example, an entity can keep track of which areas of the map it has already visited, including when those areas were visited. This information can then be accessed by a SEARCH FOR PLAYER BT to ensure that the entity does not check the same area again until a certain amount of time has passed.

Execution contexts allow components to be treated as functions with arguments (Shoulson, Garcia, Jones, Mead, & Badler, 2011). This capability enables the use of generalized behaviors that act according to the values stored in the execution context. It also reduces the amount of memory required to perform AI processing because a set of key-value attribute pairs is stored for each game entity instead of creating deep copies of BTs for each entity. Runtime performance is not noticeably affected because the BT is traversed in the same manner as before. Additionally, the execution context is passed around by reference rather than by value. Passing by value would require copies of the execution context to be placed on the program stack, which is unacceptable for potentially-large data structures.
4 Results and Evaluation

To demonstrate the power and flexibility of our architecture, three testbed video games were developed. BTs were designed using MSE and then imported into the game software during the initialization step.

4.1 BomberCube Demo

The first testbed game is BomberCube, a simple game with mechanics based on those found in the Bomberman series of games by Hudson Soft. A screenshot of Bomberman gameplay is shown in Figure 22.

![Figure 22. Screenshot of Bomberman Live gameplay (original in color) (Wong, 2007).](image)

The game pits four entities against each other in a rectangular arena. Each entity starts in a separate corner: northwest, northeast, southwest, and southeast. The objective
is to plant bombs in areas occupied by other entities in an attempt to damage them with bomb blasts. Obstacles are randomly positioned around the map. Obstacles cannot be penetrated by blasts, but some obstacles can be destroyed when a blast comes touches the obstacle. A time limit of five minutes is imposed.

When a bomb detonates, fire shoots off horizontally and vertically. Initially, the fire only reaches a grid square away from the blast origin. If a bomb explodes and an entity is caught within the blast, the entity is incapacitated for a few seconds and loses one life. This occurs regardless of who actually planted the bomb; in other words, it is possible for entities to damage themselves with their own bombs.

Power-ups are randomly placed throughout the arena. They grant varying effects to the entity who picks them up, including increased movement speed, increased bomb blast distance, increased bomb carry capacity, and a temporary super-bomb with an explosion that covers a rectangular area and ignores obstacles.

The game ends after three of the four entities lose all of their remaining lives or when time runs out. If one entity has lives remaining at the end of the game, that entity is declared the winner. If time runs out, the entity with the most lives remaining wins. If one or more entities are tied for most lives remaining, the game ends in a draw.

A screenshot of the BomberCube demo is given in Figure 23. In this particular game state, P1 has been damaged by a bomb blast detonated before the time of the screenshot. P3 is caught by bomb blast while going to collect a power-up. P2 and P4 are dropping bombs in an attempt to damage each other in the bottom-right. Entities are free to move along blue squares. Dark red squares are destructible obstacles; if a bomb detonates next to them, they will be destroyed. Dark grey squares are indestructible
obstacles. The game runs smoothly at 60+ frames per second without any noticeable dips in performance.

![Figure 23. Screenshot of the BomberCube demo.](image)

The game uses BTs to control all four entities. In fact, every entity runs the same BTs but uses its own execution context. Each entity follows this general pattern of behavior: the entity always moves toward the nearest power-up if one exists; otherwise, the entity moves randomly around the field; and the entity plants a bomb every 1 – 3 seconds if it has bombs available to plant and if another entity is nearby. In addition to these behaviors, the entity shows panic if it has two or fewer lives remaining or if only two players remain. A panicking entity periodically flashes a “!” icon above it. The collection of BTs used for controlling game entities are given in Figure 24. Note that two of the BTs share the ‘compute path’ BT-type defined by the query component: ‘Path to nearest power-up’ and ‘Path to random tile’.
At the beginning of every update loop, the entity first checks its status and then shows panic if the “Should panic?” condition has been satisfied. The panicking behavior only runs every two seconds due to the timer decorator on the ‘Panic!!’ action component. Processing continues regardless of the result due to the skip-on-failure decorator on the sequence component located at the top of the left-most branch. There are two execution choices when stepping through the tree due to the selector component.
located at the top of the right-most branch. If the entity has not already computed a path, one is created using the BT obtained by the query component. Otherwise, the entity follows an existing path. In the latter case, the entity first moves along that path and then attempts to plant a bomb. The sequence used to plant bombs is wrapped in a random timer decorator that only executes its contained sequence node when a random amount of time between one and three seconds has elapsed. The “is near player?” and “plant bomb” nodes are only visited if the “can plant bomb?” condition evaluates to TRUE.

The number of times that individual nodes in the BT were visited during one round of gameplay is given in Table 7, Table 8, Table 9, and Table 10. Entities P1 and P2 were the last remaining players before the game ended, hence the higher usage numbers relative to entities P3 and P4.

Table 7. Number of times individual nodes in MAIN TREE were visited during execution for each entity during a sample run.

<table>
<thead>
<tr>
<th>NODE NAME</th>
<th>P1 (# of visits)</th>
<th>P2 (# of visits)</th>
<th>P3 (# of visits)</th>
<th>P4 (# of visits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>2797</td>
<td>2797</td>
<td>2341</td>
<td>1864</td>
</tr>
<tr>
<td>‘Skip on failure’ decorator</td>
<td>2797</td>
<td>2797</td>
<td>2341</td>
<td>1864</td>
</tr>
<tr>
<td>‘Should panic?’ condition</td>
<td>2797</td>
<td>2797</td>
<td>2341</td>
<td>1864</td>
</tr>
<tr>
<td>Timer decorator</td>
<td>0</td>
<td>113</td>
<td>220</td>
<td>152</td>
</tr>
<tr>
<td>‘Panic!!’ action</td>
<td>0</td>
<td>11</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>Selector</td>
<td>2797</td>
<td>2797</td>
<td>2341</td>
<td>1864</td>
</tr>
<tr>
<td>‘Compute path’ query</td>
<td>2797</td>
<td>2797</td>
<td>2341</td>
<td>1864</td>
</tr>
<tr>
<td>‘Follow path’ reference</td>
<td>2763</td>
<td>2762</td>
<td>2314</td>
<td>1838</td>
</tr>
<tr>
<td>Random timer decorator</td>
<td>2763</td>
<td>2762</td>
<td>2314</td>
<td>1838</td>
</tr>
<tr>
<td>‘Can plant bomb?’ condition</td>
<td>21</td>
<td>20</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>‘Is near player?’ condition</td>
<td>19</td>
<td>20</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>Plant bomb</td>
<td>11</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 8. Number of times individual nodes in MOVE tree were visited during execution for each entity during a sample run.

<table>
<thead>
<tr>
<th>NODE NAME</th>
<th>P1 (# of visits)</th>
<th>P2 (# of visits)</th>
<th>P3 (# of visits)</th>
<th>P4 (# of visits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>2763</td>
<td>2762</td>
<td>2314</td>
<td>1838</td>
</tr>
</tbody>
</table>
Table 9. Number of times individual nodes in COMPUTE PATH TO RANDOM TILE tree were visited during execution for each entity during a sample run.

<table>
<thead>
<tr>
<th>‘COMPUTE PATH TO RANDOM TILE’ BRANCH</th>
<th>P1 (# of visits)</th>
<th>P2 (# of visits)</th>
<th>P3 (# of visits)</th>
<th>P4 (# of visits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has no path?</td>
<td>225</td>
<td>225</td>
<td>225</td>
<td>225</td>
</tr>
<tr>
<td>Compute path to random tile</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 10. Number of times individual nodes in COMPUTE PATH TO NEAREST POWER-UP tree were visited during execution for each entity during a sample run.

<table>
<thead>
<tr>
<th>‘COMPUTE PATH TO NEAREST POWER-UP’ BRANCH</th>
<th>P1 (# of visits)</th>
<th>P2 (# of visits)</th>
<th>P3 (# of visits)</th>
<th>P4 (# of visits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has no path?</td>
<td>2572</td>
<td>2572</td>
<td>2116</td>
<td>1639</td>
</tr>
<tr>
<td>Compute path to nearest power-up</td>
<td>32</td>
<td>33</td>
<td>26</td>
<td>23</td>
</tr>
</tbody>
</table>

Note that the use of a ‘follow path’ reference node was not necessary because only one BT uses that sub-tree. It is purely for demonstration purposes. However, if there were more entity types and movement patterns, it would be possible to share that sub-tree amongst all BTs that require it, eliminating behavior duplication.

Figure 25 shows a histogram of memory allocations, specifically the length of time that the objects exist in memory before being cleaned up by the C# garbage collector. The histogram was produced by Microsoft’s CLR Profiler for .NET Framework 4.
Figure 25. A memory allocation histogram that demonstrates that MindSet does not perform continual memory allocations (original in color).
For the test run given by the histogram, only 11 kB of memory was needed for objects that existed for less than 20 seconds, and most of this memory was allocated by libraries external to MindSet such as Farseer Physics and the Settlers pathfinding library. In other words, all allocations that were made were done by the AI routines stored within the BTs as opposed to the BT itself. This demonstrates that MindSet does not continually allocate memory for temporary objects during BT traversal. This is especially important on platforms with non-generational garbage collection. If the library performed many allocations, there would be many object allocations in the “less than 20 seconds” column due to objects being allocated and then being garbage-collected soon after.

4.2 Pocket Critters Demo

The second testbed game is Pocket Critters, a demo game with mechanics based on those found in the Pokémon series of games published by Nintendo. A screenshot of Pokémon gameplay is given in Figure 26.

Pocket Critters is a text-based version of the Pokémon battle system. A screenshot of Pocket Critters is shown in Figure 27. RED is controlled by a player via keyboard. BLUE is RED’s rival and is controlled by a set of BTs. The two players duel using their critters.
Figure 26. Screenshot of Pokémon gameplay (original in color) (pokemonhalloffame's Bucket, n.d.).

Figure 27. Screenshot of the Pocket Critters demo.
Since it is for demonstration purposes, Pocket Critters only contains a subset of the battle system rules from the original game released for the Nintendo Gameboy in 1996. This game is appropriate as a testbed because it contains squad-based behavior, which is an important part of fighting and strategy games.

Upon startup, each player chooses a starting critter out of their collection of six. RED is presented with a fixed list of critters and must choose one before the game proceeds. BLUE’s list of critters is randomly generated and BLUE chooses one at random. The battle then proceeds through a series of combat steps.

At the beginning of each combat step, the turn number and the statuses of the fighting critters are displayed. Then a list of options is presented to RED: FIGHT, SWITCH, ITEMS, or RUN.

If FIGHT is selected, RED is presented with a list of moves to execute and must choose one. In order to execute a move, the critter must have at least one Power Point (PP) remaining for that move. Different moves have different levels of effectiveness against opposing critters depending on the attacking critter’s type, the defending critter’s type, the move’s attack power, and the move’s accuracy.

If SWITCH is selected, RED is presented with a list of the remaining critters in their collection and must choose one. The critter currently on the battlefield is pulled back and the selected critter takes its place.

If ITEMS is selected, RED is presented with a list of items in their inventory and must choose an item to use. Items are used to restore Health Points (HP) to the critter currently on the battlefield. Different items restore a different number of HPs.

If RUN is selected, then the game is terminated.
Once RED has chosen their action, BLUE chooses a FIGHT, SWITCH, or ITEMS action based on the BT currently controlling it. RED and BLUE then order their critters to execute the actions that they have chosen. If BLUE is ordering its critter to execute a move, BLUE’s critter may ignore orders and perform a random move instead due to BLUE’s inexperience as a critter trainer. The game continues until a player’s critters are defeated.

We employ two separate sets of BTs for controlling BLUE and its critters. They are displayed in Figure 28. The ‘smart trainer’ and ‘dumb trainer’ BTs are used to control BLUE’s overall actions. The ‘obedient critter’ and ‘disobedient critter’ BTs are used to control the individual critters on BLUE’s team. BLUE acts like a squad commander and the critters act like squad members. The BTs demonstrate that BTs have applications for modelling squad-based behavior. In the implementation, squad member execution contexts are modified to contain the orders stored in the squad leader’s execution context.
The smart trainer BT uses more thinking techniques compared to BTs discussed thus far. We show this by logging important decision-making processes, as shown in Figure 29. Each log entry contains a timestamp, the name of the AI method currently being executed, and a short description of what the underlying AI code is doing. Some AI methods, such as OrderCritterToUseBestMove, write multiple entries in sequence. This log demonstrates that while individual methods in AI code can be complex, the structure of the BT that will execute that code is simple.
Figure 29. Sample log for a single combat step in Pocket Critters.
4.3 Defense Grid Demo

Our final testbed game is Defense Grid, an original game designed to showcase MindSet’s utility-based components. A screenshot is shown in Figure 30. The game takes place on a 75x11 grid. The game state updates once every half-second.

In the game, a defense grid containing 11 slots is used to defend against incoming bullets. The defense grid is located on the left side of the screen. Individual slots fire a set of three shots after briefly charging up energy. On the screen, each defense grid slot is divided into three parts. The left-most part displays the health of the slot as ‘#’ characters, the center part is a movement space for the entity, and the right-most part displays the slot’s current energy charge as ‘>’ characters. Charged shots from the
defense grid are displayed as a set of ‘>’ in space, while enemy bullets are displayed as ‘<’.

The defense grid is controlled by an entity called the defense grid controller represented by a cursor labelled ‘X’. In Figure 30, the defense grid controller is located in the third-lowest slot on the left side of the screen. The entity is controlled by a BT containing utility-based components. The objective of the entity is to destroy incoming enemy bullets, which approach from the right. The entity moves itself up and down in the set of slots. The entity can move a maximum of one slot per update step. While occupying a defense grid slot, the entity can initiate charged shots. A charged shot will destroy up to three enemy bullets. The entity can instead repair the slot it currently occupies. This process gradually restores some health over time.

The enemy ship is displayed as a set of dashes on the right side of the screen and fires single bullets from each of its slots. The enemy ship also has 11 cannon slots. Each of the enemy ship’s cannon slots fires a single bullet once every five to ten seconds. These cannon slots are controlled by separate instances of the same BT, shown in Figure 31. The BT consists of a sequence of a single action component that fires a bullet from the enemy cannon. This action component is wrapped in a randomized timer decorator component that generates a cooldown period for action executions of between five and ten seconds. The bullets fired from enemy cannons move from right to left at a rate of one to three units per update step.
The defense grid controller entity’s BT and utility score table are given in Figure 32. The BT consists of a utility selector with five child components. All of the children are utility action components.

The BT for the defense grid controller is executed as part of the update step. It evaluates the utility of each of the five actions and performs the action with the highest utility. If more than one action is tied for the highest utility score, one of those actions is executed at random.

The utility of moving up or down to another defense grid slot is determined by computing a utility value for each track. Moving up only examines the state of tracks positioned above the entity’s current location, and moving down only looks at the tracks below. The utility for individual tracks is derived from the proximity of enemy bullets to the defense grid on that track, whether the defense grid slot on that track requires repairs, and whether the defense grid recently fired or is currently firing a charged shot along that track. The final utility score for moving up or moving down is determined by the maximum utility for the set of tracks evaluated. If the entity is in the top-most defense grid slot, then an extreme negative value (-1000) is returned by the scorer to prevent this
action from being chosen. The same negative value is also returned by the scorer for moving down when the entity is in the bottom-most defense grid slot.

The demo runs for five minutes before terminating, at which point statistics are displayed to the user as in Figure 33. The statistics displayed include the number of
moves made, the number of bullets fired by the defense grid, the number of bullets fired by the enemy cannons, the amount of defense grid damage the entity repaired, and the amount of damage taken by the defense grid. We use this data over multiple runs to determine the effectiveness of our utility-based AI for controlling the defense grid control entity.

Figure 33. Screenshot of the Defense Grid demo after completion.

The results of our demo runs are shown in Table 11. We list the number of moves made by the defense grid controller, the number of charged shots fired from the defense grid, the number of enemy cannon bullets fired, the amount of damage repaired by the defense grid controller, and the amount of damage sustained.
Table 11. Statistics gathered from Defense Grid demo runs with the defense grid firing three bullets per charged shot.

<table>
<thead>
<tr>
<th>Run</th>
<th>Number of moves</th>
<th>Number of defense grid bullets fired</th>
<th>Number of enemy cannon bullets fired (EB)</th>
<th>Amount of damage repaired</th>
<th>Amount of damage sustained (DS)</th>
<th>Enemy hit rate (DS / EB) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>387</td>
<td>408</td>
<td>427</td>
<td>10</td>
<td>13</td>
<td>3.00%</td>
</tr>
<tr>
<td>2</td>
<td>352</td>
<td>410</td>
<td>432</td>
<td>9</td>
<td>12</td>
<td>2.78%</td>
</tr>
<tr>
<td>3</td>
<td>344</td>
<td>416</td>
<td>425</td>
<td>16</td>
<td>17</td>
<td>4.00%</td>
</tr>
<tr>
<td>4</td>
<td>326</td>
<td>424</td>
<td>436</td>
<td>7</td>
<td>9</td>
<td>2.06%</td>
</tr>
<tr>
<td>5</td>
<td>355</td>
<td>416</td>
<td>431</td>
<td>12</td>
<td>17</td>
<td>3.94%</td>
</tr>
<tr>
<td>6</td>
<td>334</td>
<td>417</td>
<td>428</td>
<td>2</td>
<td>4</td>
<td>0.93%</td>
</tr>
<tr>
<td>7</td>
<td>355</td>
<td>419</td>
<td>434</td>
<td>6</td>
<td>10</td>
<td>2.30%</td>
</tr>
<tr>
<td>8</td>
<td>384</td>
<td>408</td>
<td>425</td>
<td>8</td>
<td>10</td>
<td>2.35%</td>
</tr>
<tr>
<td>9</td>
<td>385</td>
<td>412</td>
<td>428</td>
<td>8</td>
<td>12</td>
<td>2.80%</td>
</tr>
<tr>
<td>10</td>
<td>315</td>
<td>420</td>
<td>432</td>
<td>12</td>
<td>13</td>
<td>3.01%</td>
</tr>
<tr>
<td>11</td>
<td>326</td>
<td>426</td>
<td>428</td>
<td>2</td>
<td>4</td>
<td>0.93%</td>
</tr>
<tr>
<td>12</td>
<td>323</td>
<td>421</td>
<td>436</td>
<td>8</td>
<td>9</td>
<td>2.06%</td>
</tr>
<tr>
<td>13</td>
<td>346</td>
<td>417</td>
<td>429</td>
<td>5</td>
<td>7</td>
<td>1.63%</td>
</tr>
<tr>
<td>14</td>
<td>344</td>
<td>414</td>
<td>428</td>
<td>11</td>
<td>12</td>
<td>2.80%</td>
</tr>
<tr>
<td>15</td>
<td>382</td>
<td>402</td>
<td>425</td>
<td>10</td>
<td>15</td>
<td>3.53%</td>
</tr>
<tr>
<td>Avg (nearest 0.01)</td>
<td>350.53</td>
<td>415.33</td>
<td>429.60</td>
<td>8.40</td>
<td>11.93</td>
<td></td>
</tr>
<tr>
<td>StdDev (nearest 0.01)</td>
<td>23.50</td>
<td>6.27</td>
<td>3.61</td>
<td>3.63</td>
<td>3.86</td>
<td></td>
</tr>
</tbody>
</table>

In these runs, the defense grid received little damage (an average of about 12 hits) relative to the total number of enemy cannon bullets fired. This demonstrates good behavior for our AI as we would expect poor behavior by the defense grid control entity to result in more damage sustained to the defense grid. The maximum enemy hit rate across all runs is 4%; that is, the defense grid controller’s worst performance only had 4% of all enemy shots fired hit the defense grid. The data’s standard deviations also show consistency in the defense grid controller’s behavior and performance.

Most of the effort for developing the utility-based AI for this testbed game was concentrated on the domain-specific scoring method. More than typical BT development, utility-based BTs require a lot of iterative, experimental development.
Overall behavior can be drastically modified by adjusting the values returned by individual scorers. We solidified the values returned by individual scorers listed in Figure 32 after a lot of experimentation before settling on a set of scorer values that caused the defense grid controller to behave as desired.
5 Conclusions and Future Work

5.1 Summary and Conclusions

The problem addressed in this thesis is the implementation of a software architecture for modelling BTs that is both flexible and extensible. Our three demos demonstrate how different types of BTs can be built using MindSet’s components. Additionally, MindSet’s implementation is easy to extend because of the architecture’s software design.

An original contribution of this thesis is the development of a new BT architecture and corresponding BT editing software: MindSet and MindSet Editor, respectively. We have demonstrated that MindSet is flexible by showing five extensions to the base BT architecture. Additionally, we have shown that our architecture performs well in real-time and can be used to control the behavior of multiple game entities. A feature comparison among available BT editors is shown in Table 12.
Table 12. The feature comparison table from Table 3, now including MindSet Editor.

<table>
<thead>
<tr>
<th></th>
<th>Behavior Designer</th>
<th>Behave</th>
<th>Behavior 3</th>
<th>Brainiac Designer</th>
<th>Unreal Engine BT</th>
<th>MindSet Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fundamental BT components (covered in Section 2.1)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Manage collections of BTs</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Drag-and-drop interface</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Can auto-arrange components</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Add comments to provide context for individual components</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Extensible via user components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Query component</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Event-driven</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Requires specific development environment</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>BT-specific debugger included</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional aborts and restarts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>Utility selector</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Utility coefficients</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

5.2 Future Work

We now consider how our research can be continued by discussing two limitations of the implementation of MindSet, MSE, and MSAPI. We close by discussing possible enhancements.

The first limitation is the required use of some type-casting boilerplate code. Recall that AI methods defined by the user act upon an object implementing the IBehaviorTreeExecutionContext interface. This is because all implementations of the IBehaviorComponent interface require the IBehaviorTreeExecutionContext parameter for
their Process method. While this ensures that all AI methods can be used across multiple
BTs and for different types of entities, it requires that AI developers perform type-casting
to the appropriate type before beginning processing. Sample code can be found in Figure
34. We cannot use the generic IBehaviorTreeExecutionContext<TEntity, TWorld>
interface implementation because it would change the AI method signature, and MSAPI
currently only works on methods that conform to exact method signatures. Some
experimentation has been done with dynamic type-casting but a concrete solution has not
been developed yet.

```csharp
/// <summary>
/// Compute a path to a random tile on the map.
/// </summary>
/// <param name="context">The execution context.</param>
/// <returns></returns>
[MindSetAIMethod("MakePath", "Compute a path to a random tile.")]
public static BehaviorReturnCode MakePath(IBehaviorTreeExecutionContext context)
{
    // *** convert the Entity and World properties to appropriate types ***
    var e = (context.Entity as BomberEntity);
    var w = (context.World as IReadonlyMapData);

    if (_pathSolver == null)
        _pathSolver = new AStarSolver(w.TileData);

    // compute a path to a random location on the map
    var pathToFollow = context.Get(PATH_TO_FOLLOW_PROPERTY);
    var initialPosition = w.GetTileForScreenPosition(e.Position);
    var endPosition = w.GetRandomOpenTile();
    _pathSolver.Search(w.TileData, initialPosition, endPosition, ref pathToFollow);

    // set values in the execution context to begin following the new path
    context.Get(PATH_TO_FOLLOW_SEGMENT_ID_PROPERTY).Value = 0;
    context.Get(MOVING_TO_POWER_UP_PROPERTY).Value = false;

    return BehaviorReturnCode.Success;
}
```

Figure 34. Example of explicit conversion used for the Entity and World properties of the execution
context.

The second limitation is that our topological sort used for ordering data in a
MSXP file only supports detection of circular dependencies via direct references. It is
still possible to cause an infinite loop at runtime due to indirect references via query
nodes. A cursory glance through available literature did not yield any algorithms for performing topological sorts with cyclical dependency detection on indirect dependencies. Therefore, users must be careful when using query nodes to ensure that no circular dependencies between BTs are introduced.

There are also a few enhancements that can be made to the MindSet architecture that offer further research opportunities.

The first enhancement would be to combine query nodes and utility selectors into a single component: the utility query component. This component would evaluate the utility of all BTs matching some criteria specified by a query and then choose the BT with the highest utility. The utility query component would provide another powerful heuristic reasoning method for the MindSet architecture in addition to providing the future-proofing capabilities of queries. Some refactoring of the MindSet architecture and MSE is required to mark specific trees as providing utility so that the query utility component can properly assess utility.

A second enhancement is developing better runtime debugging tools. Other BT editing packages support debugging, especially those for the Unity game development platform. For an integrated development environment that executes BTs, debugging is relatively straightforward to support, but we are currently unable to offer the same functionality in MSE since the application does not execute BTs. Developers using MindSet, MSE, and MSAPI are restricted to debugging their own AI code without any exact knowledge of the entire BT traversal up to that point. If a developer sets a debugging breakpoint in one of their AI methods, when code execution reaches that point they will only be able to see all parent components leading to that method’s execution. A
tracing mechanism exclusive for debug mode would be helpful for debugging issues with BTs related to how the BT is traversed. Such a tracer would be included within a game entity’s execution context and supply information about what parts of a BT have already been visited, which could yield useful information to developers when attempting to troubleshoot bugs. The tracer would not be included in release builds.

The third enhancement would be to support conditional aborts and restarts, such as are present in other BT architectures. MindSet is limited to always starting BT execution at the root node. Conditional aborts allow BTs to dynamically respond to changes and are an optimization to prevent having to reprocess the entire BT.
References


Appendices

Appendix A. Pseudo-Code, Icons for Decorator Components

Figure A1. The succeeder decorator.

```plaintext
childStatus ← execute(child)
return SUCCESS
```

Figure A2. The failer decorator.

```plaintext
childStatus ← execute(child)
return FAILURE
```

Figure A3. The repeater decorator.

```plaintext
while i < MAX TIMES
    childStatus ← execute(child)
    if childStatus != SUCCESS and childStatus != FAILURE
        return childStatus
    i++
end-while
return SUCCESS
```

Figure A4. The repeat-until-success decorator.

```plaintext
repeat
    if i == MAX TIMES
        return FAILURE
    childStatus ← execute(child)
    if childStatus != SUCCESS and childStatus != FAILURE
        return childStatus
    i++
until childStatus == SUCCESS
return SUCCESS
```
repeat
   if i == MAX_TIMES
      return FAILURE
   end-if
   childStatus ← execute(child)
   if childStatus != SUCCESS and childStatus != FAILURE
      return childStatus
   end-if
   i++
until childStatus == FAILURE
return SUCCESS

Figure A5. The repeat-until-failure decorator.

if totalCalls < MAX_TIMES
   childStatus ← execute(child)
   totalCalls++
   return childStatus
end-if
return FAILURE

Figure A6. The count-based limit decorator.

totalTimeElapsed += timeElapsedSincePreviousFrame
if totalTimeElapsed >= TIME_INTERVAL
   childStatus ← execute(child)
   totalTimeElapsed -= TIME_INTERVAL
   return childStatus
end-if
return RUNNING

Figure A7. A timer-based limit decorator. Requires a specific amount of time to elapse before executing its child component.

coolDownTimeRemaining -= timeElapsedSincePreviousFrame
if coolDownTimeRemaining > 0
   return FAILURE
end-if
coolDownTimeRemaining += TIME_INTERVAL
childStatus ← execute(child)
return childStatus

Figure A8. A timer-based limit decorator. Will execute its child component, but must wait for a specific amount of time to elapse before being to execute its child component again.
Figure A9. A timer-based limit decorator. Requires a random amount of time to elapse before executing its child component.

Figure A10. A timer-based limit decorator. Will execute its child component, but must wait for a random amount of time to elapse before being to execute its child component again.
Appendix B. Notation Used in UML Diagrams

Figure B.1. An example UML diagram.

<<Interface>>
ICreateBehaviorDesignComponents

+Create(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : BehaviorDesignComponentBase

<<Interface>>
IReadOnlyBehaviorDesignComponents

+Load(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : BehaviorDesignComponentBase

BehaviorDesignComponentFactoryBase

+Create(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : BehaviorDesignComponentBase
+Load(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : BehaviorDesignComponentBase
+Save(writer : BinaryWriter, component : BehaviorDesignComponentBase) : void

BehaviorDesignComponentFactoryBase<T>

+CreateComponent(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : T
+LoadComponent(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : T
+Save(writer : BinaryWriter, component : BehaviorDesignComponentBase) : void

RootDesignModeFactory QueryDesignModeFactory ActionDesignModeFactory SelectorDesignModeFactory SequenceDesignModeFactory ...

...
Table B1. The list of UML diagram components.

<table>
<thead>
<tr>
<th>#</th>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interface</td>
<td>Marked with “&lt;&lt;Interface&gt;&gt;”. Only contains the signatures of methods and properties. Any class that implements the interface must implement the members of the interface that are specified in the interface definition. By convention, interfaces are prefixed with the letter “I” in code.</td>
</tr>
<tr>
<td>2</td>
<td>“implements”, “is derived from”</td>
<td>Dashed-line arrows indicate a relationship between interfaces and classes. In this instance, the arrow means “BehaviorDesignComponentFactoryBase implements the ICreateBehaviorDesignComponents interface”.</td>
</tr>
<tr>
<td>3</td>
<td>Abstract class</td>
<td>An abstract class cannot be instantiated and are either partially implemented or not implemented at all. The class name in the diagram is given in italics. Similarly, any abstract methods that require an implementation in derived classes are also given in italics. Methods that have a concrete implementation use the normal font styling.</td>
</tr>
<tr>
<td>4</td>
<td>Generic class</td>
<td>Generic classes encapsulate operations that are not specific to a particular datatype. A generic class uses the “&lt;&lt;T&gt;&gt;” suffix in its name. If there are any restrictions on the datatypes allowed to be used with the generic class, that information is written in the top-right in a dashed-line box. In this case, only classes that inherit from the BehaviorDesignComponentBase class can be used with the BehaviorDesignComponentFactoryBase&lt;T&gt; class.</td>
</tr>
<tr>
<td>5</td>
<td>Methods</td>
<td>Methods all use the following notation: “[method name] ( [parameter 1], [parameter 2], …, [parameter N] ) : [return type]”. Methods prefixed with “+” are public methods, whereas methods prefixed with “#” are protected methods. Public methods can be accessed from outside the class. Protected methods can only be accessed within the class’s own code or within the code of derived classes. Methods listed in italics are abstract methods and must be given a concrete implementation in any non-abstract derived classes.</td>
</tr>
<tr>
<td>6</td>
<td>Concrete class</td>
<td>These classes are fully implemented. If they inherit from an abstract generic class or implement a generic interface on datatype T, the type used for T is displayed at the bottom. In this case, QueryDesignNodeFactory inherits from the BehaviorDesignComponentFactoryBase&lt;T&gt; abstract generic class, and T is QueryDesignNode.</td>
</tr>
</tbody>
</table>
Appendix C. MindSet Implementation Details

We have chosen C# as the programming language used to develop the MindSet BT architecture. The reasons for our final choice follow.

Being an object-oriented programming language, C# supports the concept of interfaces. An interface contains only the signatures of methods and properties; that is, it specifies “what” something does rather than “how”. In order to implement an interface, a class must provide an implementation for all members listed in the interface definition.

An interface-based design allows many object types to provide certain functionality. New class definitions just have to conform to the contract defined by the interface. The new class can then be used alongside existing classes that require an instance of the interface. This enables easy expansion of the BT architecture. Interfaces also allow only specific functionality to be exposed to different parts of the architecture. The usefulness of this will become more apparent as we delve into the implementation.

C# supports strongly-typed function pointers, known as delegates. A delegate enforces a method signature that consumers of the library must use when defining their own AI routines. This is analogous to interfaces and class implementations. A delegate can be invoked just as if we were calling a normal method.

Generics allow classes or methods to work with multiple datatypes without having to implement a method or class for each datatype. This reduces code duplication.

C# is a reflective programming language; that is, it is possible for C# programs to “observe or change [their] own code … even at runtime” (Malenfant, Jacques, & Demers, 1996). Reflection in C# enables the retrieval of information describing assemblies, datatypes, methods, and properties. The information is declared using attributes, which
provide a powerful method of associating declarative information with C# code. Once associated with a code element, the attribute can be queried at runtime by using reflection. A more detailed explanation of the reflection techniques used will be covered when discussing MSE and MSAPI.

UML diagrams will be provided to illustrate the relationships between class definitions. A quick summary of the notations used in the UML diagrams is provided in Appendix B.

After outlining MindSet, its editor, and its API, we explain how all the pieces are fit together so that AI developers can use our architecture for coding behaviors for game entities.

**Appendix C1. MindSet**

The MindSet component class library provides implementations for runtime instances of the BT architecture previously described. The classes belong to one of the following functional groups: encapsulating properties and metadata, execution context implementations, and components.

Each class used for BT metadata declarations contains some boilerplate code to effectively organize that data and make consuming it convenient for the AI programmer. We describe these classes in detail in Appendix C1.1. We then use those classes in our implementation of execution contexts, which is outlined in Appendix C1.2. Appendix C1.3 presents implementations of components which are the building blocks for constructing a BT.
In Appendix C1.4, we use classes from the previous three groups as we describe the process of constructing a BT and defining metadata for that BT. Finally, in Appendix C1.5 we describe the classes used to store multiple behavior trees. These classes allow BTs to reference each other directly via reference nodes or indirectly via query nodes.

**Appendix C1.1. Encapsulating Properties and Metadata**

BTs act on game entities and modify the game state. The data representing the game entities must be effectively organized and managed. The most important requirement for our data management system is that BTs must be flexible and be able to work with any datatype. C# generics are especially helpful here.

The PropertyDictionary class is an abstraction of a table for looking up objects based on string keys. An object can be numbers, text, or any complex type, which fulfills our requirement. All string keys must be unique across all elements stored in an instance of the PropertyDictionary class or a runtime exception will occur. Properties and methods useful for read-only access to the PropertyDictionary class are defined within the IReadOnlyPropertyDictionary interface. UML diagrams displaying this relationship are given in Figure C1, while descriptions of properties and methods implemented by PropertyDictionary are given in Table C1 and Table C2.
Figure C1. UML diagrams for the TypedKey, DataWrapper, and PropertyDictionary classes.

Table C1. Descriptions of properties implemented by PropertyDictionary.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Returns the number of elements stored in PropertyDictionary. Is read-only.</td>
</tr>
</tbody>
</table>

Table C2. Descriptions of methods implemented by PropertyDictionary.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsKey</td>
<td>Evaluates the existence of an element with the specified ITypedKey key object. Returns TRUE if the element exists; otherwise, FALSE.</td>
</tr>
<tr>
<td>Get</td>
<td>Retrieves an element matching the specified ITypedKey key object. A runtime exception is thrown if the element does not exist, either because no element exists with the key name or because an element with key name exists but it has a different datatype than the one specified by ITypedKey.</td>
</tr>
<tr>
<td>TryGet</td>
<td>Retrieves an element matching the specified ITypedKey key object, and stores it in the outgoing value parameter. Returns TRUE if the element exists; otherwise, FALSE. This method is more efficient than Get when it is more likely the element does not exist in the collection. This is because evaluating a boolean value is faster than catching and handling an ‘item not found’ exception that the Get method would throw.</td>
</tr>
<tr>
<td>Set</td>
<td>Updates an element matching the specified ITypedKey key object. If the element does not exist, it is added to the collection. If an element with the same key name but different datatype already exists within the collection, an exception is thrown.</td>
</tr>
<tr>
<td>Remove</td>
<td>Removes an element matching the specified ITypedKey key object. Returns TRUE if the element was successfully removed from the collection; otherwise, FALSE. Throws an exception if the ITypedKey key name exists but with a different datatype.</td>
</tr>
<tr>
<td>Clear</td>
<td>Removes all elements from PropertyDictionary.</td>
</tr>
</tbody>
</table>
The rationale for the separate read-only access interface is to ensure that consumers of the BT metadata are not able to modify that metadata. BT metadata can only be created and modified prior to associating it with a specific BT.

PropertyDictionary is able to store objects of any type. This explains why the class methods use generics but the class itself is not a generic class. Rather than requiring users of this class to remember the datatypes of the values they store or having to perform type-checking repeatedly at runtime, we also cache information about the stored value’s datatype. This caching is done via the use of the generic ITypedKey interface and its implementation: TypedKey. The interface declares the Name property, which is the unique key used to identify a stored value within an instance of PropertyDictionary. By using this generic interface to access and modify values stored in PropertyDictionary, we obtain strong typing. If an ITypedKey used to access PropertyDictionary finds a matching string key but the datatype associated with that string key does not match, a runtime exception will occur. This use of generics is novel in that type information is stored in the string identifier rather than being stored separately.

A type constraint exists on the ITypedKey interface and TypedKey class: it must be a reference type, signified by the “class” keyword. The reasons for this constraint will be explained momentarily.

The generic DataWrapper class is used as a work-around for a C# language feature that would adversely affect the runtime performance of PropertyDictionary: boxing and unboxing. Microsoft documentation (Microsoft, n.d.) states:
Boxing is the process of converting a value type to the type System.Object
or to any interface type implemented by this value type. When the
Common Language Runtime boxes a value type, it wraps the value inside
a System.Object and stores it on the managed heap. Unboxing extracts the
value type from the object. Boxing is implicit; unboxing is explicit. The
concept of boxing and unboxing underlies the C# unified view of the type
system in which a value of any type can be treated as an object.

The problem is that boxing and unboxing are computationally expensive processes
compared to simple assignments. When a value type is boxed, a new object must be
allocated and constructed. We would have to do this if we were setting the value for a
value type in the PropertyDictionary. Note that reference types do not require boxing or
unboxing because they are already stored on the managed heap. The type-cast required
for unboxing is also relatively expensive.

An example is given in Figure C2. The first line of code declares an integer \(i\) with
value 123. The second line of code declares an object \(o\) that references \(i\). We say that \(i\)
has been \textit{boxed}. Note that the value of \(o\) is not 123, but is actually a pointer to a location
in heap memory that is storing the value 123. The third line of code retrieves the value
referenced by \(o\) from the heap. We say the value 123 has been \textit{unboxed}.
Figure C2. A simple example of boxing and unboxing from MSDN.

In order to avoid performing these boxing and unboxing operations on any value types it stores, PropertyDictionary requires all stored values to be reference types. The DataWrapper class is such a reference type. Note that a type constraint exists on the DataWrapper class: it must be a value type, signified by the “struct” keyword. The value type is accessed via the Value property of the DataWrapper class. This enables the storage of value types within PropertyDictionary. Essentially, we perform the boxing and unboxing that C# would typically do for us; however, our approach is significantly better. Instead of performing a memory allocation every time the value type is accessed, we only perform one: when the DataWrapper class is instantiated.

Appendix C1.2. Execution Contexts

Recall that a BT is not exclusively used by a single entity and that an execution context is used to encapsulate entity- and environment-specific data. The execution context is accessed and modified by individual components as execution moves through
the BT. Our implementation for execution contexts builds upon the interfaces and classes described in the previous section. As shown in Figure C3, we provide a base interface for execution contexts that is type-agnostic. A generic interface and concrete class implementing that generic interface is also supplied. The properties and methods exposed by the objects are detailed in Table C3 and Table C4. The concrete class uses the PropertyDictionary class described in the previous section as a backing store for execution context variables.

![UML diagram for the BehaviorTreeExecutionContext<TEntity, TWorld> class.](image)

**Table C3.** Descriptions of properties implemented by the BehaviorTreeExecutionContext class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Gets the reference to the entity associated with the execution context instance.</td>
</tr>
<tr>
<td>World</td>
<td>Gets the reference to the game world object that provides methods for querying the game state.</td>
</tr>
<tr>
<td>TimeElapsed</td>
<td>Gets the amount of time that has elapsed since the previous AI step. The measurement is provided in milliseconds.</td>
</tr>
</tbody>
</table>
Table C4. Descriptions of methods implemented by the BehaviorTreeExecutionContext class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsKey</td>
<td>These methods are functionally identical to those defined for the PropertyDictionary class as they just call the corresponding method on the backing PropertyDictionary class instance used for storing execution context variables.</td>
</tr>
<tr>
<td>Get</td>
<td></td>
</tr>
<tr>
<td>TryGet</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td></td>
</tr>
</tbody>
</table>

Note that the Entity and World properties of the IBehaviorTreeExecutionContext interface class return an object instead of a specific type. This allows any datatype to be used for these two properties; however, the AI programmer must cast the returned value to the appropriate type in order to access fields and methods specific to that type. Ordinarily, a generic interface would be used as a method parameter instead, but a non-generic interface is required so that we can utilize all parts of the MindSet architecture. The main limitation is that delegates and method signatures need to match; that is, a delegate definition and any methods that act as instances of that delegate type must have an identical return type and parameter list. If any types in the method signature differ, the method cannot be used as an instance of the delegate type. Keep this in mind while we discuss individual components within the MindSet class library.

Although the generic interface IBehaviorTreeExecutionContext<TEntity, TWorld> and BehaviorTreeExecutionContext<TEntity, TWorld> implementation that act on an entity type TEntity and world object type TWorld are provided by the library, they are provided for type-casting convenience and for use as a baseline implementation of the IBehaviorTreeExecutionContext interface.
Appendix C1.3. Components

A BT is composed of one or more components. To ensure that components can all be used in conjunction with each other, all provided component types implement the IBehaviorComponent interface. Figure C4 illustrates this set of relationships: all components implement the IBehaviorComponent interface, but only a subset of components also implement the IParentBehaviorComponent interface.

![Figure C4. UML diagram for the IBehaviorComponent and IParentBehaviorComponent interfaces.](image)

The IBehaviorComponent interface declares a single method: Process. It takes an IBehaviorTreeExecutionContext as its only parameter. Once processing is complete, the result is returned as a BehaviorReturnCode: SUCCESS, FAILURE, or RUNNING. Even though all component types have the same processing method, their instantiation logic varies based on the information that the component requires to function. Figure C5 and Figure C6 show some simple examples: a selector node component only requires the collection of components that will be its children, while a repeater decorator requires a
reference to a single component and an integer representing the number of times to execute the referenced component.

```csharp
/// <summary>
/// Initializes a new instance of the <see cref="SelectorNode"/> class.
/// </summary>
/// <param name="components">Collection of individual behavior components.</param>
public SelectorNode(params IBehaviorComponent[] components) { ... }

/// <summary>
/// Initializes a new instance of the <see cref="SelectorNode"/> class.
/// </summary>
/// <param name="components">Enumerable collection of behavior components.</param>
public SelectorNode(IEnumerable<IBehaviorComponent> components) { ... }
```

Figure C5. The selector node constructors.

```csharp
/// <summary>
/// Initializes a new instance of the <see cref="RepeaterDecorator"/> class.
/// </summary>
/// <param name="component">Behavior component to wrap.</param>
/// <param name="numberOfRepetitions">The number of times to execute.</param>
public RepeaterDecorator(IBehaviorComponent component, int numberOfRepetitions) { ... }
```

Figure C6. The repeater decorator node constructor.

In order to support utility evaluation for child components, we define another interface: IBehaviorComponentWithUtilityEvaluation. In addition to the Process method inherited from the IBehaviorComponent interface, classes implementing the IBehaviorComponentWithUtilityEvaluation interface must also implement a utility evaluation function. The function returns the utility score as an integer. A UML diagram for the IBehaviorComponentWithUtilityEvaluation interface is shown in Figure C7.
We use the IBehaviorComponentWithUtilityEvaluation interface when constructing the UtilitySelectorNode component. UtilitySelectorNode requires an enumerable collection of components implementing the IBehaviorComponentWithUtilityEvaluation interface as an argument for its constructor. The constructor is given in Figure C8.

```csharp
/// <summary>
/// Initializes a new instance of the <see cref="UtilitySelectorNode"/> class.
/// </summary>
/// <param name="tieBreakingSelector">Strategy for breaking ties.</param>
/// <param name="components">Collection of behavior components.</param>
public UtilitySelectorNode
{
    ISelectSingleBehaviorComponent tieBreakingSelector,
    params IBehaviorComponentWithUtilityEvaluation[] components
}[
/// <summary>
/// Initializes a new instance of the <see cref="UtilitySelectorNode"/> class.
/// </summary>
/// <param name="tieBreakingSelector">Strategy for breaking ties.</param>
/// <param name="components">Collection of behavior components.</param>
public UtilitySelectorNode
{
    IEnumerable<IBehaviorComponentWithUtilityEvaluation> components,
    ISelectSingleBehaviorComponent tieBreakingSelector
}[
```

Figure C8. The utility selector node constructors.
Note that our implementation of the parallel sequence component uses Microsoft’s Parallel.ForEach code, which may not actually execute the tasks in parallel (Microsoft, n.d.). Instead, Parallel.ForEach distributes the collection of tasks into work items and schedules them to run on available processors. We have opted for this approach as explicitly managing threads for lightweight BT branch evaluations produces overhead that can actually degrade runtime performance if the work performed by individual threads is not intensive enough.

If we wish to create another component type in MindSet, we simply create a new class that implements the IBehaviorComponent interface. For example, the AI programmer may discover that our AI code contains many instances of method calls to play sounds. The AI programmer can create a new PlaySoundComponent and then the AI designer can modify behaviors to use that component instead, eliminating any code duplication related to sound playback. This change also allows sound playback to be modified by the AI designer outside code instead of requiring AI programmers to make the changes in AI code. Our new component may use the constructor shown in Figure C9.

```csharp
/// <summary>
/// Initializes a new instance of the <see cref="PlaySoundAction"/> class.
/// </summary>
/// <param name="sound">The sound resource to retrieve for playback.</param>
/// <param name="volume">The playback volume.</param>
/// <param name="is3D">Indicate if 3D sound is to be used for playback.</param>
public PlaySoundAction(IGetASoundResource sound, byte volume, bool is3D) { .. }
```

Figure C9. The play-sound action node constructor.

Our components are designed to be small and simple. This approach enables AI developers to compose complex behaviors using a set of common building blocks.
Table C5. Memory consumption table for individual components.

<table>
<thead>
<tr>
<th>Group Type</th>
<th>Type</th>
<th>Memory Footprint per Instance (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>ParallelSequenceNode</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>RandomSelectorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>SelectorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>SequenceNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>UtilitySelectorNode</td>
<td>16</td>
</tr>
<tr>
<td>Data accessor</td>
<td>DataWrapper</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>TypedKey</td>
<td>12</td>
</tr>
<tr>
<td>Decorator</td>
<td>AlwaysFailDecoratorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>AlwaysSucceedDecoratorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>CooldownTimerDecoratorNode</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>InverterDecoratorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>LimitExecutionDecoratorNode</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>RandomCooldownTimerDecoratorNode</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>RandomTimerDecoratorNode</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>RepeaterDecoratorNode</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>RepeatUntilFailDecoratorNode</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>RepeatUntilSuccessDecoratorNode</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>SkipOnFailureDecoratorNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>TimerDecoratorNode</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>UtilityDecoratorNode</td>
<td>12</td>
</tr>
<tr>
<td>Method</td>
<td>BehaviorAction</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>ConditionNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>UtilityBehaviorAction</td>
<td>20</td>
</tr>
<tr>
<td>Reference</td>
<td>QueryNode</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>ReferenceNode</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>UtilityReferenceNode</td>
<td>20</td>
</tr>
</tbody>
</table>

Table C5 shows how much memory is allocated when instantiating each BT component type. All the components defined by MindSet are lightweight in terms of memory footprint. The majority of memory allocated by individual components is for keeping references to objects that execute AI code.

**Appendix C1.4. Building a Behavior Tree in Code**

Our BT implementation makes use of all classes discussed thus far. To do so, it implements the IBehaviorComponent interface as shown in Figure C10. In fact, the BehaviorTree class takes an instance of a IBehaviorComponent instance as a required
constructor parameter. This makes it possible for any component type to act as a root node for a BT. Another constructor parameter is used to specify metadata for that BT, but it is optional. The constructor for the BehaviorTree class is given in Figure C11.

```csharp
/// <summary>
/// Initializes a new instance of the <see cref="BehaviorTree"/> class.
/// </summary>
/// <param name="root">Root node of the behavior tree.</param>
/// <param name="propertyDictionary">Behavior tree metadata.</param>
/// <remarks>
/// If no properties are specified, the behavior tree cannot be found by query nodes.
/// </remarks>
public BehaviorTree
(
    IBehaviorComponent root,
    IReadablePropertyDictionary propertyDictionary = null
)
```

Figure C10. UML diagram for the BehaviorTree class.

Figure C11. The BehaviorTree class constructors.

The runtime performance of a BT depends on the space and time complexities of individual components contained in that BT (Millington & Funge, 2009). If every component in the BT runs in O(1) time and requires O(1) memory, the tree will be O(n) in memory and O(n) for runtime complexity, where n is the number of components in the BT. The time complexity is O(n) because each component in the BT can be visited at most once during BT traversal.

In order to construct a BT using the MindSet architecture, child components must be instantiated before their parents, as shown in Figure C12. This order is required
because the control flow components like selectors and sequences require enumerable collections of IBehaviorComponent objects as input parameters.

```csharp
// instantiate the child components first
var hasNotComputedPathNode = new ConditionNode(BomberAIMethods.HasNoComputedPathToPowerUp);
var computePathNode = new BehaviorAction(BomberAIMethods.ComputePathToNearestPowerUp);

// instantiate the parent sequence component
var root = new SequenceNode(hasNotComputedPathNode, computePathNode);

// create the behavior tree using the parent component as the root node
// do not specify any metadata for the new behavior tree
var behaviorTree = new BehaviorTree(root);
```

Figure C12. C# code is used to construct a simple BT using MindSet.

The MindSet BT implementation facilitates the construction of BTs that model complex behavior even when components are limited to those in the base BT architecture. In particular, MindSet enables the use of a functional programming style for developing behaviors since all components require the same input and produce the same type of output: a BehaviorReturnCode value. Assuming that AI routines are coded in a modular and granular fashion, BTs can be extended with new behaviors with relatively little effort.

**Appendix C1.5. Managing Multiple Behavior Trees**

The BehaviorTreeRepository class is an important piece of the MindSet architecture. It is used for sharing single BTs amongst multiple entities; that is, the class acts as a central point of storage for BTs. The class fulfills this role by acting as a wrapper for a collection of BTs that can be retrieved via a string key. The retrieved BTs can then be used to control entity behavior. The UML diagram for the
BehaviorTreeRepository class is shown in Figure C13, while descriptions for the properties and methods defined by the class are given in Table C6 and Table C7.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Gets the number of BTs stored within the repository.</td>
</tr>
</tbody>
</table>

Table C7. Descriptions of methods implemented by the BehaviorTreeRepository class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsKey</td>
<td>Returns TRUE if a BT with the specified string key exists within the repository; otherwise, FALSE.</td>
</tr>
<tr>
<td>Add</td>
<td>Adds a BT with an associative string key to the repository.</td>
</tr>
<tr>
<td>Get (single)</td>
<td>Returns a single BT associated with the specified string key.</td>
</tr>
<tr>
<td>Get (multiple)</td>
<td>Returns an enumerable collection of BTs whose metadata satisfies some condition specified by the predicate.</td>
</tr>
<tr>
<td>TryGet</td>
<td>Retrieves a single BT from the repository with the specified string key, storing it in the outgoing value parameter. Returns TRUE if a BT with that key exists; otherwise, FALSE. This method is more efficient than Get when it is more likely the BT does not exist in the collection. This is because evaluating a boolean check is faster than catching and handling exceptions.</td>
</tr>
<tr>
<td>Clear</td>
<td>Removes all BTs from the repository.</td>
</tr>
</tbody>
</table>

A Remove method is not provided in the BehaviorTreeRepository class because BTs can depend on each other via reference nodes. Removing a BT from the repository would cause a runtime exception if execution of another BT subsequently tried to access the removed BT.
Much like the IReadOnlyPropertyDictionary interface and PropertyDictionary class, the IReadOnlyBehaviorTreeRepository interface is restricted to exposing only methods that do not modify the collection of BTs. This restriction ensures that parts of the architecture do not have write access to that repository.

Appendix C2. The MindSet Editor

When developing AI, the responsibilities are typically split between two parties: the AI designer and the AI programmer. The AI designer specifies a high-level description of the behaviors entities will exhibit: what an entity should do and under what circumstances. The AI programmer implements those behaviors in code based on the design. Given what has been discussed so far, it is possible for BTs to be defined entirely in code using the MindSet library. However, the result is less than satisfactory.

For example, consider Figure C14 below, where a BT has been defined in code that specifies the behavior that a bomberman entity will exhibit during gameplay. The specific code is not as important as the amount of it: creation of the BT involves a complex series of instantiations. There are a few problems with this development approach.
// Create behavior tree used to control bomber behavior
if(!m_BehaviorTreeRepository.ContainsKey(MAIN_BEHAVIOR_TREE))
{
    var panicActionNode = new BehaviorAction(BomberAIMethods.Panic);
    var panicTimerWrapper = new TimerDecorator(2000, panicActionNode, "panicTimeElapsed");
    IBehaviorComponent panicLock = new LockDecorator(panicTimerWrapper, PANIC_LOCK_PROPERTY);

    var queryNode = new QueryNode(m_BehaviorTreeRepository, BomberAIMethods.ChoosePathFindingAlgorithm,
    BomberAIMethods.GetPathFindingBehaviorTree);
    IBehaviorComponent computeMovement = new DiagnosticDecorator(queryNode, QUERY_NODE_COUNT_PROPERTY);

    var moveActionNode = new ReferenceNode(m_BehaviorTreeRepository.Get(FOLLOW_PATH_BEHAVIOR_TREE));
    IBehaviorComponent movementRoot = new DiagnosticDecorator(moveActionNode, FOLLOW_COMPUTED_PATH_COUNT_PROPERTY);

    var canPlantBombConditionNode = new ConditionNode(BomberAIMethods.CanPlantBomb);
    var isNearOtherPlayerConditionNode = new ConditionNode(BomberAIMethods.IsNearAnotherPlayer);
    var plantBombActionNode = new BehaviorAction(BomberAIMethods.PlantBomb);

    IBehaviorComponent bombingAction = new SequenceNode
    {
        new DiagnosticDecorator(canPlantBombConditionNode, CAN_PLANT_BOMB_COUNT_PROPERTY),
        new DiagnosticDecorator(isNearOtherPlayerConditionNode, IS_NEAR_PLAYER_COUNT_PROPERTY),
        new DiagnosticDecorator(planBombActionNode, PLANT_BOMB_COUNT_PROPERTY)
    };

    var randomBombingTimerWrapper = new RandomTimerDecorator(1000, 3000, bombingAction, "bombTimeElapsed", "");
    IBehaviorComponent bombingRoot = new DiagnosticDecorator(randomBombingTimerWrapper, RANDOM_TIMER_COUNT_PROPERTY);

    var mainBranch = new SelectorNode(computeMovement, new SequenceNode(movementRoot, bombingRoot));
    var rootNode = new SequenceNode(panicLock, mainBranch);
    BehaviorTree behaviorTree = new BehaviorTree(rootNode, ROOT_NODE_COUNT);
    m_BehaviorTreeRepository.Add(MAIN_BEHAVIOR_TREE, behaviorTree);
Firstly, BTs written in code are difficult to understand because the structure of the BT is not immediately obvious when looking at the code. Often, the only way to picture a BT defined in code is to run through the creation flow manually and then draw the BT separately. Obviously, the graphic becomes outdated as soon as the BT code is changed.

Secondly, code changes are inflexible. If the AI designer wishes to make minor changes to the behavior and does not have the programming skills needed to make the change, the assistance of the programmer is required. This results in a lengthy feedback loop.

Thirdly, building BTs programmatically is prone to error due to mistakes not being found until runtime in the form of exceptions. If information required for the BT to function is not present or is invalid, in most cases a runtime exception will occur. These types of errors are not easy to detect just by looking at a block of BT construction code.

Lastly, code changes require deployment for testing. Hard-coded behavior trees require the game software to be recompiled anytime changes are made. For large games, the recompilation process could take hours, which makes it impossible to quickly deploy changes.

MindSet Editor (MSE) has been developed to solve these problems. MSE is a graphical editor that is used for creating and editing BTs. It operates outside of game code, but it uses any AI code that programmers have exposed to MSE. The MSE software was developed using Windows Forms.

In Appendix Appendix C2.1, we discuss editor features and the underlying code that makes it all possible. In Appendix Appendix C2.2, we explain how BT data is managed by MSE.
Appendix C2.1. Editor Features and Internals

The MSE application window is split into panes. Each pane is responsible for displaying different information to the user. The panes and descriptions of them can be found in Figure C15 and Table C8, respectively. Pane management is performed using the third-party WeifenLuo.WinFormsUI.Docking library, which can be obtained from https://github.com/dockpanelsuite/dockpanelsuite. Document panes containing BT data cannot be manipulated, but the other individual panes can be docked, hidden, or floated.

One architectural decision for the code was forced by the pane management library: each pane must be a separate view. This presents a problem since different views must be able to communicate with each other. To prevent high coupling between views, the observer design pattern is used. Per Head-First Design Patterns (Freeman, Freeman, Sierra, & Bates, 2004):

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The observer pattern is also referred as the publish-subscribe pattern. The object that changes state sends a message through a publisher and then any components that subscribe to that message type will receive a copy of that message.
Figure C15. A sample view of MSE.

Table C8. Descriptions of panes in MSE.

<table>
<thead>
<tr>
<th>#</th>
<th>Pane</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project Explorer</td>
<td>Shows all BTs included within the current project. BTs are organized into folders. Any assemblies imported by the user are shown within the grey References folder.</td>
</tr>
<tr>
<td>2</td>
<td>Behavior Tree View</td>
<td>Shows the current BT being edited. Multiple BTs can be open at the same time and are accessed by clicking on the appropriate tab. The selected component is highlighted in yellow in the Behavior Tree View.</td>
</tr>
<tr>
<td>3</td>
<td>Properties</td>
<td>Shows information related to the BT component current selected by the user. If no component is selected, nothing appears in this pane.</td>
</tr>
</tbody>
</table>
To implement the publish-subscribe pattern, an event aggregator is placed at the root-computer-program window level and all views are created with a reference to the event aggregator. When an event occurs, a message is published which is then routed through the aggregator. Examples of events include a project being opened or the MSE user switching between BT views. Other views subscribe to that event via the aggregator and receive their own copy of the message to process, as illustrated in Figure C16.

![Figure C16. Example of an event message being published and then routed through the event aggregator to all subscribers.](image)

Although there is no direct coupling between publishers and subscribers, there still exists semantic coupling via event arguments classes. This coupling occurs because modifying an event arguments class requires modifying whatever classes publish that event and whatever classes subscribe to that event. Fortunately, our events are simple and granular, and there is usually only a single class that depends on a particular event occurring. Due to this simplicity, the problems associated with this type of coupling are rare.

The basis of MSE’s extensibility is the ability to import assemblies. The assemblies must define specific types of C# classes in order for importing to be successful. Those class types will be discussed at length in Section Appendix C3. Any number of assemblies can be imported into a MSE project.
The first step in creating a BT is to add the components. The root node is always available when a new BT is created and it cannot be deleted. Right-clicking any component opens a context menu. If the component has cardinality value ExactlyOne or AtLeastOne, the “Add Child…” option will appear. This option is unavailable when right-clicking a component with cardinality value ExactlyOne that already has a child component. Selecting “Add Child…” opens a dialog box that displays a list of types of components that can be created to the user. Upon selecting a component type from the list, that component is added to the BT as a child of the right-clicked component.

Once components have been created, they can be edited. After selecting a component via a mouse-click, the component’s properties are displayed in the Properties pane. Individual values can be modified via the property grid. Each row in the property grid displays the name of the property and the property’s current value.

Components can also be moved via drag-and-drop. The user must click the mouse to select a component, hold the mouse button down, and then drag the component into the desired position. It is not possible for components to overlap on the screen. During the movement process, the component stays at its current position while the user moves a grey ‘ghost box’. The ghost box will turn red if it overlaps with an existing component. Components are restricted to being moved horizontally. This restriction ensures that parent-child relationships are preserved in the graphical representation: parents always appear above their children on the Y-axis. There is also an auto-arrange function available. The algorithm, derived from previous work, cleans up the arrangement of components such that parents are centered over their child components (Walker, 1991; Lim R., 2014).
Components can be deleted. This is done by opening the right-click context menu and choosing the “Delete” option. Deleting a component will also delete all of its children.

For simplicity and ease of development, BT data has been separated into three distinct parts: functional data, relational data, and spatial data. All three parts are associated via a globally unique identifier (GUID) that is assigned to a specific component. The GUID is a randomly-generated 128-bit integer. The three parts are kept separately in simple lookup tables, all of which use the GUID of the lookup key. These types of data structures make it easier to manage manipulation of BT data because we do not need to preserve a tree structure at design-time while the BT is being modified. Additionally, C# libraries do not provide a built-in tree structure, so using the lookup tables saves a non-negligible amount of development effort.

Functional data is information that determines how components function at runtime. This type of data may include the name of a C# method to call, or the set of numeric parameters that controls a timer. The root node is always included in the lookup table by default.

Relational data describes the parent-child relationships between components. For each component GUID in the lookup table, there is an associated enumerable collection of GUIDs that correspond to the component’s children. The collection may be empty. A reverse lookup is also used for caching the child-parent relationship between components.

Spatial data gives the positions of axis-aligned bounding boxes (AABBs) corresponding to components within the BT. Here, an axis-aligned bounding box is a 2D rectangular region in the graphical output with edges running parallel to the X and Y
axes. We use a *quadtree* data structure in order to store spatial data for components. A quadtree is a tree data structure in which each internal node represents some AABB and has exactly four children. Each child corresponds to a quadrant in its parent node’s region. The regions associated with nodes can be recursively subdivided into smaller and smaller regions. Objects represented as AABBs in 2D space can be stored in a quadtree in the smallest quadrant the object fits into. If an object cannot fit entirely in a quadrant, it is stored in the quadrant’s parent.

Using the quadtree, we can determine which component lies beneath a user’s mouse-click in \( O(\log n) \) time, where \( n \) is the number of components in the BT. Suppose we are given a user’s mouse-click at position \( p = (x, y) \). We begin our search through the quadtree starting at the quadtree’s root node. At each search level, we first check if there are any objects stored within the current node whose AABB contains \( p \). If such an object exists, we return that object and stop processing. If not, we check which of the current node’s quadrants contains \( p \), and then continue our search in that quadrant. This process is repeated until the object is found, or until a leaf node of the quadtree is reached and a final search does not find the object. The algorithm’s efficiency comes from being able to discard entire subsets of objects as invalid candidates because they are stored in quadrants that do not contain \( p \).

Figure C17 depicts the underlying quadtree node boundaries for a BT. We see how quadtree nodes are subdivided based on the position and size of components contained in those nodes. For example, we see that the quadtree boundary extends just past the edge of the BT on the left of the ‘SHOULD PANIC?’ condition component and
below the ‘PLANT BOMB’ action component. Additionally, quadtree nodes are only split into smaller rectangular regions if a BT component is contained within that region.

In order to properly construct a logical tree structure using the data stored in our lookup tables, the root node is always assigned the zero-GUID: zero represented as a 128-bit integer. The lookup table containing relationship data can then be used to construct the tree.

Individual BT documents are also uniquely identified by GUID in MSE. These document identifiers are used by reference components to link to other BTs in the project.
Figure C17. Example of the AABBs of individual quadtree nodes.
Appendix C2.2. Managing Behavior Tree Data

In order to organize information about BTs for a particular project, MSE uses three different file types: MindSet Behavior Trees (MSBTs), MindSet Editor Projects (MSEPs), and MindSet eXported Projects (MSXPs). The relationships between the three file types are shown in Figure C18 below.

An MSBT file contains information about a single BT. This includes components, the parent-child relationships between them, the AABBs associated with components, and some editor-specific information such as labels and comments.

An MSEP file contains paths to all assemblies imported by the user as well as links to all individual MSBT files included in the project.

An MSXP file is a single file containing all information in an MSEP file and all MSBT files that are linked to the MSEP file. The MSXP file is imported into game software at runtime.

Figure C18. The relationships between different MSE file types.
The data in a MSXP file is arranged such that a collection of BTs can be constructed on-the-fly while reading the file contents. This arrangement accounts for dependencies between components via parent-child or sibling relationships and dependencies between BTs via reference nodes.

Breath-first traversal is used to determine a bottom-to-top, left-to-right ordering of components of a single BT written to the MSXP file. Such an ordering is illustrated in Figure C19 below, where the numbers of the nodes indicate the traversal order. Leaf components at the deepest level of the BT come first in the ordering according to their sibling relationship from left to right, followed by all nodes at the next level up, and so on. Component relationship data is written to the MSXP file before the data for individual components. This arrangement allows the structure of the BT to be known before beginning to construct components. A BT is built from the bottom up.

Figure C19. The ordering scheme used for writing node data to a MSXP file.

We write BTs to the MSXP using an ordering determined by a topological sort. The topological sort requires knowledge of the BT project and of all reference nodes in individual BTs. For example, consider the scenario in Figure C20 below: A has a reference node pointing to B and B has a reference node pointing to A. If a BT references
itself or a circular dependency exists between a set of BTs, then it is possible for the execution of those BTs to result in an infinite loop at runtime.

![Diagram](image)

**Figure C20.** Example of a circular dependency between BTs via reference nodes.

```c
// GetErrors method
// id : identifier of the BT to process
// dGraph : collection of dependencies between BTs (the dependency graph)
// resolvedIDs : topological ordering of identifiers associated with BTs with
// resolved dependencies
// visited : collection of identifiers associated with BTs already visited
// trail : list of node identifiers in the current recursion trail
// returns errors
add id to trail

bool inProcess ← false
bool alreadyVisited ← visitedNodes[id] ?? false
if alreadyVisited
  if inProcess
    if id == trail[0]
      add 'detected reference to self' to errors
    else
      add 'circular dependency exists' to errors
    end-if
  end-if
else
  if visited does not contain id
    add id to visited
    visited[id] ← true

  // for all BTs that depend on the current BT, get their dependency errors
  // recursively store the topological ordering in resolvedIDs
  foreach dID in dGraph[id]
    add GetErrors(dID, dGraph, inout resolvedIDs, visited, trail) to errors
  end-if
  visited[id] ← false
  add id to resolvedIDs
end-if
return errors
```

**Figure C21.** The algorithm for determining if any circular dependencies exist between BTs within the same repository.
The algorithm given in Figure C21 implements a topological sort and is used to create an ordering for BT document GUIDs. This ordering ensures that BTs that do not depend on other BTs will be written to the MSXP file before the BTs that depend on them. For example, consider the example is given in Figure C22 below, where a collection of BTs A, B, C, D, and E are defined. Note that B depends on A and C, and E depends on C. The topological sort produces an ordering that will cause BTs A and C to be written to the MSXP file before BTs B and E.

Figure C22. Example of ordering for BTs determined using topological sort.

The algorithm also detects circular dependencies. The ordering is computed as part of MSE’s validation process immediately prior to writing the BTs to a MSXP file.

Once the ordering has been determined, we validate it as well as individual BTs. Any validation errors produced are displayed to the user in the Project Export Error List pane. Examples of validation errors include: a project containing no BTs; one or more BTs share the same filename, which would cause lookup collisions at runtime; a BT with no components defined other than the root node, which executes no behavior;
component that requires child components to function has no children, which creates an
invalid BT; a BT contains a reference to itself or a circular dependency exists between a
set of BTs, resulting in infinite loops; or individual property values in a component are
invalid.

Validations are performed whenever the user initiates the project export process.
We do not perform validations whenever the user changes something in the project
because doing so can be expensive in some cases. It could also annoy users if warnings
and errors keep popping up while they are in the midst of building a BT. MSE only
checks whether a project is fully valid when exporting it.

Once the validation process is complete, individual validations are categorized
based on their severity and are shown in the Project Export Error List pane, as depicted in
Figure C23. There are three validation categories from least severe, to most: message,
warning, and error. A message is purely informational and does not adversely affect
anything. A warning informs the user of a problem that may affect usability within MSE,
but correcting the problem is optional. An error indicates that something is wrong with
the BT. If any BT returns an error, then no MSXP is written to disk. The user must
correct all errors before a MSXP file can be successfully exported.

![Project Export Error List pane.](image)

Figure C23. The Project Export Error List pane.
The pane displays all problems in the order they were encountered during validation. Icons are used to indicate the level of severity. A helpful description of the validation problem is given along with the filename for the offending file. Double-clicking on a line corresponding to a component in a BT will shift window focus to the offending component. If the file containing the component is not currently open, MSE simply opens the file prior to shifting window focus. Clicking the buttons at the top of the pane can show or hide specific severities.

More information regarding validations for specific component types will be discussed alongside the implementations for design-time components.

Appendix C3. The MindSet API

The MindSet API (MSAPI) is an application programming interface. It is a layer of abstraction between the MindSet library and MSE. MSAPI enables the development of AI code as plugins for MSE.

MSAPI includes the following object types: attributes, components, factories, and transforms. These object types are discussed in Appendix C3.1 through Appendix C3.4, respectively. Objects of all types are external to MSE, which allows users to create components and then import them into MSE. An example of such an external object will be given in Appendix C3.5. Finally, in Appendix C3.6 we outline steps required for outfitting AI code with MSAPI objects to that the AI code can be used by MSE.

As in Appendix C1, UML diagrams are given throughout this section to show relationships between class definitions.
Appendix C3.1. Attributes and Metadata

The approach used to make AI code usable by BTs created in MSE is based on attributes. An attribute encapsulates some declarative information that can be attached to program elements such as classes, enumerated types, or methods. AI programmers specify attributes above a class definition. We say that class definition has been tagged with the attribute. The declarative information is fixed for all instances of that class type at runtime. MSE retrieves that information using reflection and dynamically creates instances of those program elements when users are building their BTs in MSE.

A variety of attribute classes are declared that are used with specific portions of MindSet. The following attributes must be used in order to properly expose AI functionality to MSE: MindSetAIClass, MindSetAIMethod, MindSetAIComponent, MindSetAIComponentFactory, and MindSetAITransform.

MindSetAIClass can only tag class types. The tagged classes will be accessible to MSE as ‘AI method’ classes.

MindSetAIMethod can only tag class methods. It is used to denote which methods inside a class are exposed to MSE for use as an action, condition check, or query method depending on the method signature. The attribute also contains DisplayName and Description properties that will show information about that method in MSE. The class that the AI method belongs to must be tagged with the MindSetAIClass attribute, otherwise MSE will not load it.

MindSetAIComponent can only tag class types. The tagged classes will be accessible to MSE as BT component classes so long as the tagged class inherits from the BehaviorDesignComponentBase abstract class. The attribute defines a persistence
identifier, display name, description, cardinality, and visibility flag for a component type. More information on MindSet components is given in Section Appendix C3.2.

MindSetAIComponentFactory can only tag class types. The tagged classes will be accessible to MSE as BT component factories that are responsible for creating, loading, and saving data for individual BT component types. The attribute defines the component type that the factory is responsible for. Factories are described in Appendix C3.3.

MindSetAITransform can only tag class types. The tagged classes are used during BT instantiation while reading a MSXP from disk at runtime to convert design-time representations of BT components to their runtime equivalent. The attribute defines the component type that the transform is responsible for. Transforms are described in Appendix C3.4.

Attributes are used so that classes do not have to be instantiated at runtime in order to retrieve the above information. We simply query for the existence of the appropriate attribute on a class type, method, or enumeration type value during the assembly import process. The use of attributes also ensures that the information for a specific programming element is constant amongst all instances of that element.

There are two attributes in MSAPI that are optionally used to declare metadata for BTs: MindSetAIBehaviorType and MindSetAIBehaviorTreeTypeCustomProperties.

MindSetAIBehaviorType can only tag enumerated types. It is used to signify that the enum declares a set of behavior tree types.

MindSetAIBehaviorTreeTypeCustomProperties can only tag values in an enumerated type. The attribute acts on a class type that inherits from the
BehaviorTreeCustomPropertyBagBase class, which acts a wrapper for a list of BehaviorTreeCustomProperty instances. Those instances are implementations of the PropertyDescriptor abstract class and are used to encapsulate information required for editing properties in the Properties pane in MSE. The enumeration value’s type must have the MindSetAIBehaviorType attribute declared so that is properly exposed to MSE.

Sample code is given in Figure C24. In this example, the BehaviorTreeType enum has been tagged with the MindSetAIBehaviorType attribute, so MSE will display it as a possible selection for behavior tree type. The PathFinding enum value has been tagged with the MindSetAIBehaviorTreeTypeCustomProperties attribute, and so MSE will display a ‘Requires Power Up’ checkbox in the property grid pane while the root node of a BT with PathFinding type is selected.

More descriptive use cases for the individual attribute types are discussed in Appendix C3.5 and Appendix C3.6.
/// <summary>
/// An enumeration of the available behavior tree types.
/// </summary>
[MindSetAIBehaviorTreeType]
public enum BehaviorTreeType
{
    /// <summary>
    /// The behavior tree is used for path-finding.
    /// </summary>
    /// <remarks>
    /// Includes metadata specified by the PathFindingBTCustomProperties class.
    /// </remarks>
    [MindSetAIBehaviorTreeTypeCustomProperties(typeof(PathFindingBTCustomProperties))]
    PathFinding,
    /// <summary>
    /// The behavior tree is used for performing movement.
    /// </summary>
    /// <remarks>
    /// Does not specify any metadata.
    /// </remarks>
    PerformMovement
}

/// <summary>
/// The metadata associated with behavior trees used for path-finding.
/// </summary>
public class PathFindingBTCustomProperties : BehaviorTreeCustomPropertyBagBase
{
    public PathFindingBTCustomProperties() : base(new List<BehaviorTreeCustomProperty>()
    {
        new BehaviorTreeCustomProperty
        {
            "REQUIRES_POWER_UP", // the name used to identify the property
            typeof(bool), // the datatype of the property
            null, // a type converter
            // optional parameter that specifies the property's display name
            new DisplayNameAttribute("Requires Power Up"),
            // optional parameter that specifies the description for the property
            new DescriptionAttribute("Does a power-up exist on the field?"")
        }
    })
    {
        // No other initialization required
    }
}

Figure C24. The declaration of an enum type and corresponding metadata class used to identify BT types.
Appendix C3.2. Components

The data representation for components is different in MSE from in the MindSet runtime library. Recall that components in MindSet are lightweight: the only requirement is that a component class implements the IBehaviorComponent interface, which itself only contains a specification for the Process method. Since MSE will not be running BTs, it uses a different representation for components during design-time. MSAPI provides design-time representations for all components provided by the MindSet library.

We include extra information as part of the component data at design-time: editor-specific properties to enhance readability and provide context in MSE; versioning capabilities to support enhancements to components without breaking backwards compatibility with older versions of components; and mechanisms for enforcing constraints so that things do not break at runtime. Additionally, we must be able to validate components for correctness prior to exporting MSEPs to MSXPs. We must also be able to map to AI program constructs like methods that would normally be passed around MindSet directly using class instances and delegates. These mappings must be able to create instances of those AI constructs.

Rather than bloating runtime component classes to also work within MSE, we instead create separate classes to represent the components at design-time, as shown in Figure C25. Properties shared amongst all design-time components are listed in Table C9. All these components have the MindSetAIComponent attribute declared for them so that MSE can display important contextual information to the user. Neglecting to declare the attribute would prevent the component type from being properly exposed to MSE.
Figure C25. UML diagram for the BehaviorDesignComponentBase class and inherited types.

Table C9. Properties defined by the BehaviorDesignComponentBase class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistGuid</td>
<td>A read-only GUID property used to identify the specific component type. It must be unique across all component types.</td>
</tr>
<tr>
<td>Name</td>
<td>A read-only property that is the underlying class name of the component type.</td>
</tr>
<tr>
<td>Description</td>
<td>A read-only property that lists a description of the component’s type.</td>
</tr>
<tr>
<td>ChildNodeCardinality</td>
<td>A read-only property that specifies one of three values listed in an enumerated type that determines the required number of child components (if any) for the component type: None, ExactlyOne, or AtLeastOne. It is used for validation during the project export process. If a BT contains a component that does not meet its cardinality constraint, the project export cannot proceed.</td>
</tr>
<tr>
<td>ID</td>
<td>A read-only property that is the identifier assigned to the specific instance of a component upon creation. This identifier is unique among all components within a BT.</td>
</tr>
<tr>
<td>VersionNumber</td>
<td>A read-only property that returns an integer specific to the component type. It is used for versioning components and maintaining backwards compatibility as new component-specific properties are added.</td>
</tr>
<tr>
<td>NodeIdentifier</td>
<td>A modifiable text property. The text stored in this property is displayed in the heading line of the component in MSE. This property is not exported.</td>
</tr>
<tr>
<td>Comments</td>
<td>A modifiable text property. The text stored in this property is displayed beneath the component’s heading text in MSE. This property is not exported.</td>
</tr>
</tbody>
</table>

The BehaviorDesignComponentBase abstract class provides properties that are used by MSE for managing components. Some properties are used only within MSE and are not needed for runtime usage. In those cases, the properties are not included as part of the project export process. An individual component type derived from the BehaviorDesignComponentBase class can declare its own properties unique for that component type.
For any components that must hold mappings to AI methods, we have chosen to represent those mappings as space-delimited strings that hold the full AI class name including namespaces and the AI method name. The format of such a string is “Namespace1.Namespace2.[…].NamespaceN.ClassName MethodName”. This scheme for AI method mappings is required for creating instances of delegates: in order to create delegate instances, C# requires that the delegate type, class name, and method name be provided. After creating a delegate instance, a BT can then execute the method represented by the delegate.

The string mappings are created using TypeConverters derived from a generic abstract class: DelegateTypeInstanceTypeConverterBase. A derived class specifies a delegate type for the generic parameter as pictured in Figure C26. MSE retrieves methods from AI code matching the method signatures of any delegate types. This approach also allows developers to create custom components with their own delegate types.

MSE treats decorators like any other component type. A DecoratorDesignNodeBase class is provided for convenience. This class derives from BehaviorDesignComponentBase. In MSE, decorator components are identified by a ribbon-and-bow icon.
Figure C.26. UML diagram for the DelegateTypeInstanceTypeConverterBase<T> class and inherited classes.

 DelegateTypeInstanceTypeConverterBase<T>  
 +GetStandardValuesSupported(context : ITypeDescriptorContext) : boolean  
 +GetStandardValuesExclusive(context : ITypeDescriptorContext) : boolean  
 +GetStandardValues(context : ITypeDescriptorContext) : StandardValueCollection

 ActionMethodSelectionConverter  

 ConditionCheckMethodSelectionConverter  

 QueryMethodSelectionConverter  

 QueryResultSelectionMethodSelectionConverter  

 UtilityEvaluationMethodSelectionConverter
All component types have a public Validate method that performs basic validation common to all component types. A warning is given if a component has not been assigned an identifier by the user. The identifier is a design-time property that is displayed in the header line of individual components. An error is given if a component’s cardinality condition is not met. The cardinality condition is determined by an enumeration type value None, ExactlyOne, or AtLeastOne. The value indicates the number of child components that must be linked to the current component.

After performing the basic validations, Validate then calls DoCustomValidation. DoCustomValidation is responsible for performing validations specific to the component type. It is an abstract method, so all component types must provide their own implementation for it.

Any validation errors produced by the Validate and DoCustomValidation methods are merged into a single collection. The collection of validation errors is then displayed in the Project Export Error List pane.

**Appendix C3.3. Factories**

After design-time component classes have been defined as described in Appendix C3.2, MSE must be able to create new instances of these components. MSAPI delegates these tasks to *factory* classes, so named because they are implementation of the *factory method design pattern* (Freeman, Freeman, Sierra, & Bates, 2004):
The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

MSE also requires the ability to load components from a binary file on disk and write component data to a binary file. We extend our factory classes to also implement a variant of the *strategy design pattern* (Freeman, Freeman, Sierra, & Bates, 2004):

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Our variant of the strategy design pattern ensures that read- and write-logic is implemented for individual component types. The read-logic is another example of the factory design pattern. We are just creating a new instance of a component based on information from a file rather than creating a new instance of that component with default initial values. The factory classes are responsible for all reading and writing of BT component data to binary streams, which include MSBT and MSXP files.

An individual factory class definition is responsible for exactly one component type and it must inherit from the BehaviorDesignComponentFactoryBase class. Additionally, it must declare the MindSetAILComponentFactory attribute on the class definition. This attribute is used for forcing a factory class to handle precisely one component type. A UML diagram for these classes is given in Figure C27.

In our implementation, we have combined the factory and strategy patterns into a single class. Rather than requiring developers to create three classes that handle creating, loading, and saving components separately and properly outfit each of those three classes
with the appropriate attribute, we instead opted for an approach requiring a single class and single attribute. This keeps creation and persistence logic localized to a single class in the code. Factory classes should only be providing implementations of Create, Load, and Save and not holding state. This approach simplifies some of the bootstrapping code in MSE because we only have to instantiate a single class instead of three.
Figure C27: UML diagram for the BehaviorDesignComponentFactoryBase class and inherited classes.

```
<<Interface>>
ICreateBehaviorDesignComponents
+Create(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : BehaviorDesignComponentBase

<<Interface>>
ILoadBehaviorDesignComponents
+Load(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : BehaviorDesignComponentBase

BehaviorDesignComponentFactoryBase
+Create(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : BehaviorDesignComponentBase
+Load(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : BehaviorDesignComponentBase
+Save(writer : BinaryWriter, component : BehaviorDesignComponentBase) : void

BehaviorDesignComponentFactoryBase<T>
+CreateComponent(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : T
+LoadComponent(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : T
+SaveComponent(writer : BinaryWriter, component : T) : void
+create(documentID : uniqueIdentifier, nodeID : uniqueIdentifier) : BehaviorDesignComponentBase
+Load(reader : BinaryReader, documentID : uniqueIdentifier, nodeID : uniqueIdentifier, versionNumber : uint) : BehaviorDesignComponentBase
+Save(writer : BinaryWriter, component : BehaviorDesignComponentBase) : void

RootDesignNodeFactory  QueryDesignNodeFactory  ActionDesignNodeFactory  SelectorDesignNodeFactory  SequenceDesignNodeFactory  ...
```

[T : RootDesignNode]  [T : QueryDesignNode]  [T : ActionDesignNode]  [T : SelectorDesignNode]  [T : SequenceDesignNode]
The three methods provided by each factory class – Create, Load, and Save – are declared as abstract methods and so must be given an implementation in any derived classes. The method implementations are all expected to fulfill the following functions:

The Create method is called when adding a new component to a BT in MSE. It instantiates a new component associated with the specified document and with the specified identifier. All other properties are initialized with default values according to the component specification.

The Load method is called while loading BT data from a MSBT file. It instantiates a component and returns it based on data read from an open binary input stream. It is also responsible for loading component-specific information given the parameters passed into the method. The version number parameter is used for managing backwards compatibility for components.

The Save method is called while saving BT data to a MSBT file. It writes component-specific data to an open binary output stream. The data written for components always corresponds with the newest version.

A generic BehaviorDesignComponentFactoryBase<T> class is also defined that provides stronger data-typing for the component type the factory handles, but it simply wraps the methods defined by the non-generic BehaviorDesignComponentFactoryBase class. The type parameter T must be a class that inherits from the BehaviorDesignComponentBase class. The non-generic class is required because MSE maintains a collection of factory classes and they must all share a class type in order to be stored in the same collection.
Appendix C3.4. Transforms

Once the MSEP file has been exported to a MSXP file as described in Appendix C2.2, the MSXP file can be read from disk into memory for use at runtime. However, the data contained within the MSXP file is still a design-time representation of the BT. In fact, the same factory classes used to load data into MSE is used to load MSXP data for individual components at runtime. The data must first be converted from its MSAPI representation to its equivalent implementations in the MindSet library. This is accomplished via transforms.

A set of transforms are defined by MSAPI, one for every component type provided by the library. This set demonstrates an implementation of the one-way data mapper design pattern, analogous to a mathematical function: given a design-time MSAPI component class instance and other parameters as input, the output of the transform is a MindSet library component class instance that can be used for executing AI behaviors at runtime.

All transforms inherit from the DesignNodeToRuntimeNodeTransformerBase<T> class, where T is a class that inherits from BehaviorDesignComponentBase. The UML diagram for the DesignNodeToRuntimeNodeTransformerBase<T> class and a subset of derived classes is shown in Figure C28. Methods defined for the DesignNodeToRuntimeNodeTransformerBase<T> class are listed in Table C10. Individual transform classes are responsible for exactly component type, enforced by the MindSetAITransform attribute declared on the transform class definition.
Figure C28. UML diagram for the DesignNodeToRuntimeNodeTransformerBase<T> class and inherited classes.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoTransform</td>
<td>An abstract method that must be implemented by any derived classes. It is responsible for performing component—specific transformation logic.</td>
</tr>
<tr>
<td>Transform</td>
<td>This method performs some preliminary validation prior to running the design-time component through the DoTransform method. There are two validations. The first performs type-matching to ensure we’re accepting the correct design-time component type for the transformation. The second verifies that the node has the correct number of children based on the cardinality enumeration value specified for the component type: either None, ExactlyOne or AtLeastOne.</td>
</tr>
</tbody>
</table>

Much like factories, a non-generic base class is provided so that MSAPI can work with all transforms identically and store them within the same collection.

Because DoTransform executes the transformation logic as part of the public method Transform, both methods share the same set of parameters: the component to apply the transformation to, the component’s child nodes, the read-only instance of a repository containing behavior trees loaded from disk up to that point, and a collection of AI and BT metadata retrieved via reflection that is required for some transformations. The AI and BT metadata includes behavior tree type enumeration types tagged with the
MindSetAIBehaviorTreeType attribute, AI method class types tagged with the
MindSetAIClass attribute, and a lookup table for BTs by their document GUID.

Recall that a MSXP file arranges the data for a BT such that relationship data is
given before data for individual components. Also recall that child components are
written before their parents as depicted in Figure C19 from earlier. This enables on-the-fly BT creation in $O(n)$ time, where $n$ is the number of components in the BT. As
components are read from the MSXP file, they are cached for quick retrieval by their
component GUID. Because all component relationship data is completely known prior to
constructing components, components with children can just reference the cache for the
appropriate components as shown in Figure C29.

![Diagram](image)

Figure C29. Demonstrates component creation done in $O(n)$ time given a BT with $n$ components.

Most transformations are relatively straightforward since components have fixed
implementations: a binary stream for a given component type always adheres to the same
data specification. For example, a query node always expects two space-delimited strings
corresponding to a query method and selection method defined in the AI assembly. In
this case, the transform is responsible for creating instances of delegates corresponding to
the two method strings.

The transformation process for root nodes is different from all other component
types. This is because there is no analog for root node in the basic MindSet architecture.
MSE uses the root node to store metadata for the BT. Metadata is any dynamic collection of properties describing the BT.

When saving root node data to a MSBT or MSXP file, the root node factory writes type information for the behavior tree type enumeration value as a space-delimited type-and-value string to the open binary stream first. We then write the number of properties contained in the property bag. Finally, for each property stored in the property bag, we serialize the property value to a byte array then write the property name, byte array length, and the byte array to the binary stream.

When loading root node data from a MSBT or MSXP file, the root node factory first reads the space-delimited type-and-value string from the open binary stream. This string represents the behavior tree type enumeration value. Upon assigning that enumeration value to the root node via a property setter, we do a search for the MindSetBeaviorTreeTypeCustomProperties attribute on the value. If such an attribute is found, we instantiate the appropriate BehaviorTreeCustomPropertyBagBase class as dictated by the attribute. Next, the number of properties to read is extracted. For each property to be read, we read the property name and the number of bytes that represent the property value. Finally, we read that many bytes from the stream and deserialize the byte array, storing the result in the corresponding property bag entry. This process requires that individual property values are serializable.

The root node transform acts on the data loaded in the previous step. The output of the transform is an instance of the BehaviorTree class. Recall that the constructor of the BehaviorTree class takes an IBehaviorComponent instance and an optional property dictionary parameter. At this point in the transformation process, all other components in
the BT have been constructed. The ChildNodes parameter will contain a single component corresponding to the component placed directly beneath the root node in MSE. If a property bag has been defined for the root node, it is converted to a PropertyDictionary class instance. The component and the PropertyDictionary are then used to create the final BehaviorTree class instance.

**Appendix C3.5. Extension Capabilities**

MSE imports the MSAPI assembly by default for all projects. Note that this DLL import is not displayed within the project’s References folder in the Project Explorer pane. This is done to eliminate clutter in the UI. Another reason it is not displayed is because it cannot be removed from the References folder.

The base classes for components, factories, and transforms are all defined as public classes within the MSAPI assembly. This lets AI developers define their own component types that can be imported into MSE for use in their BTs. In fact, the procedure is identical to how we have declared the appropriate component, factory, and transform classes within MSAPI.

First, we create a new component’s runtime class. This class must implement the IBehaviorComponent interface which supplies the Process method. Properties are added to the class as required.

Once we have completed development of the runtime class, we create a new component’s design-time class. It must inherit from the BehaviorDesignComponentBase abstract class and be tagged with the MindSetAIClass attribute to declare a persistence GUID, display name, description, and cardinality. The new component class
must implement the DoCustomValidation method in order to perform component-type-specific validation. More complex components may require the use of TypeConverters to properly display information in MSE’s Properties pane.

Next, we create a new factory class. This class must inherit from the non-generic BehaviorDesignComponentFactoryBase class or the generic BehaviorDesignComponentFactoryBase<T> class, where T is the design-time component type developed in the previous step. Using the generic class is the recommended approach as it removes the need for some boilerplate code associated with verifying the correct type of component is being handled by the factory. An implementation for the Create, Load, and Save methods must be given. The factory class definition must also be tagged with the MindSetAIComponentFactory attribute.

We then create a new transform class. This class must inherit from the non-generic DesignNodeToRuntimeNodeTransformerBase or the generic DesignNodeToRuntimeNodeTransformerBase<T> class, where T is the design-time component type used for the factory in the previous step. Using the generic class is the recommended approach as it does some preliminary validation prior to performing the transform: namely, that the component being run through the transform is the correct type. The Transform method must be implemented. The transform class definition must be tagged with the MindSetAITransform attribute.

Finally, we import the assembly containing the component design-time class, factory class, and transform classes into MSE. MSE requires that the three classes be declared within the same assembly. This prevents users from dividing the class definitions between different assemblies and then forgetting to import one of them.
Once the above steps are completed, the component will become available for use within MSE.

When loading a MSXP file from disk inside an AI application, any custom factory and transform classes must be added to the BehaviorTreeNodeTransformRepository instance passed into the MSXP loader. If at any point during the transformation process a component type is not recognized, a runtime exception will be thrown.

Appendix C3.6. Setting Up AI Code to Work With MindSet Editor Using MindSet API

BTs must have behavior code to execute if they are to be useful. The process for exposing AI code to MSE requires a new set of classes and methods tagged with the appropriate attributes. This is similar to the process for creating a new component type.

We create a new static class that will contain a collection of AI code. The use of a static class to contain AI methods is done to prevent AI code from being written as instance methods. The AI code should only act on whatever data is passed to it via function parameters and should not hold entity state. Entity state should instead be stored within execution contexts. The class definition must be tagged with the MindSetAIClass attribute.

Once we have created the static class definition, we create AI methods as static methods inside that class. The method signature depends on the type of AI method being written as shown in Figure C30 below. In order to be displayed in MSE, an AI method must be tagged with the MindSetAIMethod attribute. If the method is not tagged, it will
not be available for use in MSE; however, that code can still be used internally by other AI methods.

Finally, we compile the AI library into a DLL and then import the DLL into MSE. Assuming the above steps have been performed correctly, the AI code will be available for use by MSE.

We have required the usage of attributes to properly expose classes and methods to MSE instead of automatically pulling all types and methods from an AI assembly that are compatible with MSAPI. This is because some AI code may be used to perform common boilerplate functions and are not meant to be executed on their own. Not tagging these methods hides them from MSE while keeping them available for the internal use of AI code.
namespace MindSet_AIDev.Entities
{
    
    public static class BomberAIMethods
    {
        ...
        
        /// <summary>
        /// Predicate method used to determine if a behavior tree is used for computing
        /// paths.
        /// </summary>
        /// <param name="info">Behavior tree metadata to evaluate.</param>
        public static bool ChoosePathFindingAlgorithm(IBehaviorTreeInfo info)
        {
            // "MindSet_AIDev.Entities.BomberAIMethods ChoosePathFindingAlgorithm"
        }
        
        /// <summary>
        /// Predicate method used to determine which path-finding behavior action to use.
        /// </summary>
        /// <param name="filteredResults">Collection of path-finding behaviors.</param>
        /// <param name="context">Execution context.</param>
        public static BehaviorTree GetPathFindingBehaviorTree
        {
            IEnumerable<BehaviorTree> filteredResults,
            IBehaviorTreeExecutionContext context
        }
        {
            // "MindSet_AIDev.Entities.BomberAIMethods GetPathFindingBehaviorTree"
        }
        
        /// <summary>
        /// Indicates if the entity should display panicking behavior.
        /// </summary>
        /// <param name="context">Execution context.</param>
        public static bool ShouldPanic(IBehaviorTreeExecutionContext context)
        {
            // "MindSet_AIDev.Entities.BomberAIMethods ShouldPanic"
        }
        
        /// <summary>
        /// Compute a path to a random tile on the map.
        /// </summary>
        /// <param name="context">Execution context.</param>
        /// <returns></returns>
        public static BehaviorReturnCode MakePath(IBehaviorTreeExecutionContext context)
        {
            // "MindSet_AIDev.Entities.BomberAIMethods MakePath"
        }
        
        ...
    } // end of BomberAIMethods class definition
}
} // end of MindSet_AIDev.Entities namespace scope

Figure C30. A static class is defined containing AI methods. Displays corresponding space-delimited mapping strings.
Appendix C4. Reading a MSXP File into Memory

A MSXP file loader is provided by MSAPI so that developers do not have to write their own. The loader takes as one of its parameters an instance of an object implementing the IReadOnlyBehaviorTreeNodeTransformRepository interface. BehaviorTreeNodeTransformRepository is MSAPI’s default implementation of this interface. The UML diagram is given in Figure C31. Properties defined by the BehaviorTreeNodeTransformRepository are listed in Table C11 and its methods are listed in Table C12.

```csharp
///<Interface>
interface IReadOnlyBehaviorTreeNodeTransformRepository

+BehaviorTreeTypes : IEnumerable<Type>
+AIMethodClasses : IEnumerable<Type>

///<Interface>
interface IReadOnlyBehaviorTreeNodeTransformRepository

+ComponentLoaders : IReadOnlyDictionary<Guid, ILoadBehaviorDesignComponents>
+Transforms : IReadOnlyDictionary<Guid, DesignNodeToRuntimeNodeTransformerBase<T>>

BehaviorTreeNodeTransformRepository

+AddBehaviorTreeNodeEnum(behaviorTreeType : Type) : void
+AddAIMethodClass(aimethodClass : Type) : void
+AddComponentLoaderAndTransform(loader : ILoadBehaviorDesignComponents, transform : DesignNodeToRuntimeNodeTransformerBase<T>) : void
```

Figure C31. UML diagram for the BehaviorTreeNodeTransformRepository class.

Table C11. Properties defined by the BehaviorTreeNodeTransformRepository class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BehaviorTreeTypes</td>
<td>The collection of behavior tree type enumeration types.</td>
</tr>
<tr>
<td>AIMethodClasses</td>
<td>The collection of class types that contain AI methods.</td>
</tr>
<tr>
<td>ComponentLoaders</td>
<td>A lookup table of component loader classes identified by</td>
</tr>
<tr>
<td></td>
<td>component persistence GUID.</td>
</tr>
<tr>
<td>Transforms</td>
<td>A lookup table of component transform classes identified by</td>
</tr>
<tr>
<td></td>
<td>component persistence GUID.</td>
</tr>
</tbody>
</table>
Table C12. Methods defined by the BehaviorTreeNodeTransformRepository class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddBehaviorTreeTypeEnum</td>
<td>Adds type information corresponding to an enumerated type that defines behavior tree types. The type being added must be an enumerated type and be tagged with the MindSetAIBehaviorTreeType attribute.</td>
</tr>
<tr>
<td>AddAIMethodClass</td>
<td>Adds type information corresponding to a class that defines AI methods. The type being added must be tagged with the MindSetAIClass attribute.</td>
</tr>
<tr>
<td>AddComponentLoaderAndTransform</td>
<td>Adds instances of a component loader and transform for a specific component type to the repository. The component loader class type must be tagged with the MindSetAIComponentFactory attribute and the transform class type must be tagged with the MindSetAITransform. Also, the component loader and transform must both act on the same component type.</td>
</tr>
</tbody>
</table>

The component loaders and transforms defined in the MSAPI assembly are all included within BehaviorTreeNodeTransformRepository by default. The methods exist so that user extensions to MSAPI and AI code can be plugged into the MSXP loading process as demonstrated by the code sample in Figure C32. If the user neglects to do this, runtime exceptions will be thrown when trying to process the missing types.

```csharp
public protected BehaviorTreeRepository Read(ContentReader input)
{
    // add the behavior tree type enumeration type and AI class type to the transform repository
    var nodeTransformRepository = new BehaviorTreeNodeTransformRepository();
    nodeTransformRepository.AddBehaviorTreeTypeEnum(typeof(BomberBehaviorTreeType));
    nodeTransformRepository.AddAIMethodClass(typeof(BomberAIMethods));

    // read the contents of the MSXP file
    return input.ReadBehaviorTreeRepository(nodeTransformRepository);
}
```

Figure C32. Example of user-defined types being plugged into the transform repository so that the MSXP file loader can correctly process them.
The IReadOnlyBehaviorTreeNodeTransformRepository interface is used by the MSXP loader to ensure that developers cannot add new type information to the transform repository while MSXP loading is in progress.
Appendix D. Source Code Metrics

<table>
<thead>
<tr>
<th>Project</th>
<th>Number of lines of code (obtained from Visual Studio code metrics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MindSet</td>
<td>1580</td>
</tr>
<tr>
<td>MindSet API</td>
<td>1567</td>
</tr>
<tr>
<td>MindSet Editor</td>
<td>6865</td>
</tr>
<tr>
<td>Bombercube</td>
<td>2551</td>
</tr>
<tr>
<td>Pocket Critters</td>
<td>1132</td>
</tr>
<tr>
<td>Defense Grid</td>
<td>376</td>
</tr>
<tr>
<td>Total</td>
<td>14071</td>
</tr>
</tbody>
</table>