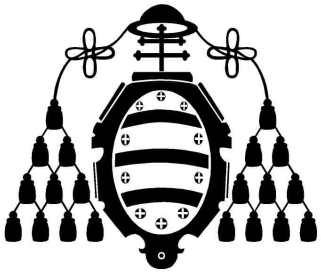
UNIVERSIDAD DE OVIEDO

Departamento de Informática



UNIVERSIDAD DE OVIEDO

TESIS DOCTORAL

Un modelo dinámico de toma de decisiones para sistemas de Inteligencia Artificial en videojuegos adaptado al estilo del jugador

Presentada por:

Sergio Ocio Barriales

para la obtención del título de Doctor en Informática

Dirigida por:

Doctor D. José Antonio López Brugos

Doctor D. Chris Jenner

Gijón, Diciembre de 2010

Agradecimientos

Cuando somos niños, es común escuchar la pregunta "¿qué quieres ser de mayor?". Aunque puede variar con el tiempo, llega un momento en el que en nuestra cabeza va tomando forma una respuesta clara a esta pregunta. A partir de ese momento, enfocamos nuestros esfuerzos en conseguir cumplir estos sueños de infancia; aunque no todos lo logramos, yo he tenido el privilegio de cumplirlos. Esta Tesis es el resultado de todos estos años de trabajo, en los que han sido muchas las personas las que me han ayudado a estar donde hoy me encuentro.

En primer lugar, he de agradecer a mis padres y mi hermana, por haber alimentado mis ilusiones, así como a mi prometida por haber estado siempre a mi lado aunque éstas nos hayan hecho estar lejos de nuestras familias, incluso en tiempos difíciles.

También debo agradecer a mi director, Don José Antonio López Brugos, por animarme y llevarme a terminar esta Tesis; gran parte de la misma está apoyada en mi experiencia profesional, la cual no hubiera sido posible sin la confianza depositada en mí tanto por mi co-director, Chris Jenner, como por Chris Preston, quienes me dieron la oportunidad de trabajar como ingeniero de IA en videojuegos. Agradezco igualmente los comentarios y opiniones de Tom Sarkanen.

Tampoco puedo olvidar la inestimable ayuda de mi amigo Richard Pearce, quien me ha ayudado a que este documento esté redactado correctamente en un idioma diferente a mi lengua materna, ni a Luis Cascante, que me ayudó a mejorar partes de este libro, así como de los artículos publicados como parte de esta investigación.

Por último, no puedo dejar de recordar a mis abuelos Roberto Barriales Ardura, que siempre supo que llegaría donde yo quisiera, y Jesús Ocio Achaerandio, quien, incluso en su enfermedad, se preocupó por mí y mis estudios; siempre estaréis en mi corazón.

Acknowledgements

When we are kids, it is common to be asked “what do you want to be when you grow up?” Even though the answer can vary as time passes, there is a moment, in our minds, when we have a clear answer to that question. From that moment, we focus our efforts on achieving our childhood dreams. Not everyone is able to realise their dreams, but I have been lucky enough to be able to do so. This Thesis is the outcome of years of hard work, and a lot of people have helped me get where I am today.

First of all, I must be grateful to my parents and my sister for having cherished my dreams, as well as to my fiancée for having stood by me all this time, away from our families, even in hard times.

I must also be grateful to my supervisor, José Antonio López Brugos, for encouraging me to finish this Thesis; much of it is based on my professional experience, which I would not have been able to get without the trust my co-supervisor, Chris Jenner, as well as Chris Preston put in me two years ago, when they gave me the opportunity to work as a game AI engineer. I thank Tom Sarkanen for his comments and tips too.

I also cannot forget to thank my friend Richard Pearce for helping me to proofread this document, and Luis Cascante, who helped me to improve parts of this book, as well as the papers published as part of this research.

Lastly, I must remember my grandparents Roberto Barriales Ardura, who always knew I could get wherever I wanted to, and Jesús Ocio Achaerandio, who, even during his illness worried about me and my studies; you will always be in my heart.

Resumen

En los últimos años, el videojuego se ha convertido en la industria más exitosa del negocio del entretenimiento. El desarrollo de un gran juego ya no es algo que pueda ser llevado a cabo por una única persona trabajando desde su habitación, sino una enorme tarea donde cientos de personas trabajan juntas para construir un producto sobresaliente.

Hoy en día, los desarrolladores buscan una característica distintiva que pueda hacer que sus juegos destaquen entre la competencia. Tradicionalmente, gran parte del trabajo se dedicaba a conseguir mejores y más deslumbrantes gráficos, pero, en un momento en el que muchos juegos comparten la misma tecnología (como Unreal Engine o Anvil), la inteligencia artificial juega un papel fundamental. Construir comportamientos creíbles y que presenten un desafío puede realmente marcar la diferencia.

El objetivo esencial de todo juego es ser divertido. Sin embargo, encontrar una buena definición de "diversión" no es una tarea trivial; cada jugador tiene sus preferencias, y diseñar una experiencia que todo el mundo pueda disfrutar no es siempre posible. La mayoría de los sistemas de IA en juegos tratan de mitigar este problema ofreciendo un tosco método de adaptación, los niveles de dificultad, en lugar de implementar sistemas completamente adaptativos. Dichos sistemas deberían ser capaces de identificar diferentes clases de jugadores, y adecuar los comportamientos de la IA apropiadamente, intentando maximizar la diversión.

La colaboración entre diferentes equipos es también esencial, y permitir que nuevas ideas puedan ser estudiadas y probadas rápidamente es fundamental.

Son varias las técnicas de IA que se han aplicado a juegos comerciales hasta el momento, pero ninguna es lo suficientemente completa para ofrecer la posibilidad de crear prototipos ágil y fácilmente, o modificar los existentes dinámicamente.

Con esta tesis, nuestro objetivo es construir una técnica multi-nivel que nos permita generar IAs autónomas, ofreciendo, a su vez, un sistema para modificar sus comportamientos añadiendo o eliminando capas de lógica de más alto nivel de forma dinámica. También intentamos producir un sistema que pueda ser utilizado

por cualquier miembro del equipo, evitando costosos trabajos adicionales de ingeniería, a la vez que los juegos se benefician de la capacidad de probar nuevas ideas velozmente.

Intención y objetivos

Esta tesis surge de la experiencia profesional del autor, y de la necesidad de mejorar ciertos procesos, así como la comunicación entre ingenieros y otros equipos para construir la mejor IA posible.

Nuestra intención es construir una IA adaptativa que pueda sorprender a los jugadores, presentando diferentes desafíos para conseguir que los juegos sean tan divertidos como sea posible. Sin embargo, tras nuestra investigación inicial, hemos llegado a la conclusión de que no existe un sistema que pueda utilizarse con este propósito de forma sencilla. Por tanto, hemos decidido separar el trabajo en dos fases, siendo la primera la presentada en este trabajo, y que está centrada en desarrollar una técnica que:

- Permita crear IAs autónomas y que no necesiten ninguna orden externa para mostrar una conducta inteligente.
- Es lo suficientemente abstracta como para poder ser utilizada en diferentes tipos de juegos.
- Siga un enfoque guiado por datos, lo cual nos permitiría construir herramientas para controlar y generar nuevo contenido sin la necesidad de ningún trabajo adicional de ingeniería.

La segunda fase de esta investigación estaría orientada al estudio de cómo este nuevo sistema puede ser utilizado para adaptar comportamientos de inteligencia artificial al estilo de cada jugador.

Aporte de este trabajo

Para alcanzar las metas presentadas en el punto anterior, hemos desarrollado una técnica novedosa, basada en pilas de árboles de comportamiento que son ejecutados de forma concurrente para producir nuevos resultados. Hemos denominado a esta técnica *Árboles de Comportamiento con ejecución basada en sugerencias* (Hinted-execution Behaviour Trees, HeBT). En esta estructura jerárquica, los niveles superiores pueden *sugerir* a los inferiores qué deberían hacer en cada momento.

Las principales contribuciones de los HeBT son:

- Permiten añadir nueva lógica a la IA dinámicamente. Los árboles de alto nivel funcionan como plug-ins, que pueden ser activados o desactivados para modificar el resultado final. Esto puede ser utilizado para adecuar la experiencia de juego a cada momento.

- El sistema puede implementarse como una librería abstracta, de tal forma que no esté atado a un determinado tipo de juego. Esto permite que una gran parte del código de IA pueda compartirse entre diferentes proyectos, disminuyendo la carga de trabajo y reduciendo riesgos, ya que la lógica habría sido probada a fondo.

- Una vez que las acciones y condiciones creadas específicamente para un juego hayan sido codificadas, la IA puede ser controlada por completo utilizando una herramienta. Este editor de HeBTs puede ser utilizado por diferentes equipos, permitiendo a todo el mundo contribuir a la IA final mostrada por el juego.

Como parte de esta tesis, hemos desarrollado un sistema de HeBT completamente funcional, un editor de árboles, un prototipo de juego y un completo ejemplo, de modo que podamos evaluar qué es lo que nuestra solución aporta a los desarrolladores de juegos.

Palabras clave

Videojuegos, Inteligencia Artificial, IA, modelado de comportamientos, árbol de comportamientos, máquinas de estados, arquitecturas multi-nivel, sugerencias

Abstract

In the last few years, the videogame industry has become one of the big players in the entertainment business. Developing a big game is no longer something a single person can do operating out of their bedrooms, but a massive project where hundreds of people work together to build an outstanding experience.

Nowadays, developers look for a distinctive characteristic that can make their games stand out. Efforts have traditionally been put into building better and shinier graphics, but, in a time where many games share the same technology (such as Unreal Engine or Anvil), AI plays a fundamental role. We think creating believable and challenging behaviours can really make a difference.

The main objective of every game is to be fun, but defining what *"fun"* means is not trivial; every player has their own preferences, and designing an experience that everyone can enjoy is not always possible. Most game AIs try to mitigate this problem by offering a coarse adaptation method, difficulty levels, rather than implementing fully adaptive systems. Such systems should be able to identify different classes of players and tailor their behaviours accordingly, trying to maximise the fun.

Also, collaboration between different teams is very important, and allowing new ideas to be prototyped and tested quickly is essential.

Several AI techniques have been applied to games so far, but none of them is complete enough to offer swift prototyping and ease of use to build new behaviours or modify existing ones dynamically.

With this thesis, our objective is to build a multi-layered technique that allows us to generate autonomous AIs, yet offering a system to modify their behaviours adding or removing higher-level layers of logic dynamically. We also aim to produce a system that can be used by any member of the team, thus removing any additional –and costly– engineering work, while games benefit from a quicker turnaround of new ideas.

Aims and objectives

This thesis emerges from the professional experience of the author, and the need of improving processes and communication between engineers and other teams to build the best possible AI.

Our aim is to build an adaptive AI that can surprise players, presenting different challenges to make games as fun as they can be. However, after our initial research we noticed there is no system that can be used for this purpose easily. Thus, we decided to split this work into two different phases, being the first the one presented in this thesis, which is focused on developing a system that:

- Allows us to create autonomous AIs that do not need any external input to behave intelligently.
- Allows behaviours to be modified dynamically, so we can create new ones just by adding additional layers of logic. These layers must be easily understandable and modifiable by non-technical staff.
- Is abstract enough that it can be used in different types of games.
- Uses a data-driven approach, so we can build tools to control and generate new content without requiring any additional engineering work.

The second phase will be aimed at studying how this new system can be applied to adapt AI behaviours depending on the style of each particular player.

Novel aspects of this work

In order to achieve the goals discussed in the previous point, we have developed a novel technique, based on stacks of behaviour trees that are run concurrently to produce new results; we have called them *Hinted-execution Behaviour Trees*.

In this hierarchical structure, higher-levels are able to communicate with lower ones by passing hints as to what they should be doing. The main advantages of HeBTs are:

- They allow new logic to be added to the AI dynamically. Higher-level trees work as plug-ins that can be enabled or disabled to modify the final output. This can be use to adapt the experience properly at any given moment.
- The system can be implemented as an abstract library, so it is not tied to a specific type of game. This also allows for a big part of the AI code to be shared among different projects, reducing the workload, and reduces some risks, as the logic would already have been tested.
- Once the different game-specific actions and conditions have been coded, the AI can be entirely controlled using a tool. This HeBT editor can be used by different teams, so everyone can contribute to the final AI.

As part of this thesis, we have developed a fully functional HeBT system, an editor, a game prototype, and a complete example, so we can evaluate what our solution can offer to game developers.

Keywords

Videogames, AI, behaviour modelling, behaviour tree, state machines, multi-level architecture, hint, HeBT, BT

TABLE OF CONTENTS

TABLE OF CONTENTS 16

TABLE OF FIGURES 21

TABLE OF TABLES 28

CHAPTER 1. INTRODUCTION 30

1.1. AI IN VIDEOGAMES 30
1.2. GAME DESIGN AND AI 32
1.3. BEHAVIOUR MODELLING 34
1.4. ADAPTATION 36

CHAPTER 2. OBJECTIVES 37

CHAPTER 3. ORGANISATION 41

3.1. STATE OF THE ART 41
3.2. PROBLEM AND SOLUTION 41
3.3. APPLYING HEBTS TO A REAL-WORLD EXAMPLE 42

STATE OF THE ART 43

CHAPTER 4. REACTIVE SOLUTIONS 44

4.1. FINITE-STATE MACHINES 44
4.2. FSM WITH EXTENDED STATES 45
4.3. STACK-BASED FINITE-STATE MACHINES 46
4.4. HIERARCHICAL FINITE-STATE MACHINES 48
4.5. CONCURRENCY IN STATE MACHINES 49
4.6. CONCURRENT, HIERARCHICAL FINITE-STATE MACHINES 50
4.7. PETRI NETS 51

CHAPTER 5. AUTOMATIC PLANNING 54

5.1. GOAP 54
5.2. HTN 58

CHAPTER 6. REACTIVE PLANNING 64

6.1. BELIEF-DESIRE-INTENTION 64
6.2. BEHAVIOUR TREES 66
6.2.1. BEHAVIOUR SELECTION AND INTERRUPTION HANDLING 68

HINTED-EXECUTION BEHAVIOUR TREES 70

CHAPTER 7. DEFINING THE PROBLEM 71

7.1. COMPARING BEHAVIOUR MODELLING TECHNIQUES 71
7.2. OBJECTIVE 74
7.3. HOW ARE BTS USED IN REAL GAMES? 76
7.4. IMPROVING PROTOTYPING 77
7.4.1. PERSONALITIES 77
7.4.2. HINTS 78
7.4.3. ADDING NEW LOGIC 80

CHAPTER 8. OVERVIEW OF THE SYSTEM 83

8.1. COMPONENTS 83
8.1.1. HINTED-EXECUTION BEHAVIOUR TREE LIBRARY 83
8.1.2. HEBT EDITOR 84
8.1.3. GAME PROTOTYPE 84
8.2. HIGH-LEVEL ARCHITECTURE 85

CHAPTER 9. BUILDING A BEHAVIOUR TREE SYSTEM 86

9.1. MANAGER AND INSTANCES 86
9.1.1. AI MANAGER 87
9.1.2. INSTANCE 87
9.2. DEFINING A TREE 88
9.3. COMPOSITE NODES 89
9.3.1. SEQUENCE 90
9.3.2. PARALLEL 91
9.3.3. SELECTOR 93
9.4. LEAVES 95
9.4.1. CONDITIONS 95
9.4.2. ACTIONS 99
9.5. FILTERS 99
9.5.1. LOOPS 100
9.5.2. CONDITIONAL EXECUTION 103

9.5.3. RESULT MODIFIERS 103
9.6. RUNNING A BEHAVIOUR 104

CHAPTER 10. EXPANDING OUR BT: HINTS 107

10.1. THE CONCEPT OF HINT 107
10.2. EXECUTION FLOW IN BEHAVIOUR TREES 108
10.3. IMPLEMENTING A HINT SYSTEM 108
10.3.1. UPDATING HINTS 109
10.3.2. REORDERING BRANCHES 110
10.4. HINTS AND CONDITIONS 114
10.4.1. HINT CONDITION 116

CHAPTER 11. MULTI-LEVEL ARCHITECTURE 118

11.1. BEHAVIOUR CONTROLLERS 118
11.1.1. RUNNING AN HEBT 121
11.2. EXPOSING HINTS TO HIGHER-LEVELS 123
11.3. HINT NODES 126
11.3.1. PARALLELS AND HINT NODES 126
11.4. WHY ALLOW MORE THAN TWO LEVELS? 127

CHAPTER 12. APPLYING HEBTS TO A GAME 129

12.1. CHOOSING A GAME 129
12.1.1. HALF-LIFE AND THE SOURCE ENGINE 130
12.2. REPLACING HALF-LIFE'S AI SYSTEM 131
12.2.1. CREATING CUSTOM NODES 132
12.3. BUILDING AND USING HEBTS 135
12.3.1. EXPORTING A SIMPLE BT 138
12.3.2. EXPORTING CONDITION TREES 141
12.3.3. BUILDING AN HEBT 143
12.4. SETTING BEHAVIOURS 150
12.4.1. LEVEL CREATION 151
12.4.2. DYNAMIC CHANGES 153

CHAPTER 13. VISUAL EDITING 155

13.1. FUNCTIONALITY 155
13.2. MAIN WINDOW 156

13.3. ARCHITECTURE 157
13.3.1. DOCUMENTS 157
13.3.2. BEHAVIOUR TREES 159
13.3.3. NODE LIBRARIES 162
13.3.4. CONDITION TREES 165
13.4. CREATING A NEW TREE 166
13.4.1. LOW-LEVEL TREES 167
13.4.2. HIGH-LEVEL TREES 169
13.5. GAME COMMUNICATION 170
13.6. FUTURE WORK 174

CHAPTER 14. WHY HEBTS ARE GOOD FOR AI PROGRAMMING: A PRACTICAL EXAMPLE 176

14.1. PROTOTYPING NEW IDEAS 176
14.2. BASE BEHAVIOUR 177
14.2.1. COVER BRANCH 178
14.2.2. PATROL BRANCH 180
14.2.3. ATTACK BRANCH 181
14.3. HIGH-LEVEL TREES 183
14.3.1. MODIFYING AIS KNOWLEDGE 184
14.3.2. BUILDING OUR HIGH-LEVEL LOGIC 186

CHAPTER 15. CONCLUSIONS AND FUTURE LINES OF RESEARCH 190

15.1. RESULTS 190
15.2. FUTURE RESEARCH 192
15.2.1. ADAPTATION TO PLAYERS 192
15.2.2. GROUP BEHAVIOURS 194
15.2.3. MOVING TO A PRODUCTION ENVIRONMENT 196
REFERENCES 197

TABLE OF FIGURES

Figure 1. Structure of a game AI system 31
Figure 2. The autonomous AI system is controllable, in most games, via a high-level interface..... 33
Figure 3. This multi-layered structure allows a higher-level tree to indicate the layer below how it should behave..... 38
Figure 4. Basic finite-state machine to control a guard NPC..... 45
Figure 5. Each state in an FSM with extended states works as a small FSM with three states..... 46
Figure 6. Adding a new state to an FSM can start causing some design problems . 47
Figure 7. A stack-based FSM, showing the state of its stack when 'Suspicious' is the active state..... 48
Figure 8. HFSM, where the state 'suspicious' has been expanded to a new FSM..... 49
Figure 9. A Petri net that models our simple concurrent behaviour 52
Figure 10. Our Petri net once one of the AIs takes control over the mounted gun. 52
Figure 11. Different plans can be created depending on the costs of the actions used 57
Figure 12. Task network used in our example 62
Figure 13. Simple BDI representation of our guard AI..... 65
Figure 14. Simple behaviour tree describing a 'Melee attack'..... 67
Figure 15. Development process with regular Behaviour Trees 75
Figure 16. A good system should be able to let both designers and engineers modify and prototype behaviours..... 76

Figure 17. The usage of personality traits can help us modify the normal execution of a tree 78

Figure 18. A simplified vision of our guard's AI 79

Figure 19. When a selector receives a hint, it reorders its branches 80

Figure 20. A simple tree can be used to modify a main behaviour 81

Figure 21. If the AI is not suspicious, or not enough time has passed, then we reorder the selector so attack is the last available option 82

Figure 22. High-level architecture of the system 85

Figure 23. We represent each agent as an AIInstance, being those controlled by the AIManager 86

Figure 24. Relations of the manager with the rest of the library 87

Figure 25. Connections of an AIInstance to the rest of the library 88

Figure 26. A view of the main classes in charge of defining a behaviour tree 89

Figure 27. An example sequence 91

Figure 28. Example of a parallel node 92

Figure 29. A simple selection allows the AI to choose whether it has to reload its weapon 94

Figure 30. Design of a condition tree 96

Figure 31. An example of a complex condition 97

Figure 32. Example of a precondition 98

Figure 33. Example of an assertion 98

Figure 34. Filters that are available in the library 100

Figure 35. A filter working in a real-world example 100

Figure 36. Process involved in the creation and initialisation of an AI 105

Figure 37. A BT is always executed starting from its root. The execution flow is determined by the type of nodes used to build the tree 105

Figure 38. Structure of our improved selectors 109

Figure 39. Hints are sent to BTs, which then notify all the selectors 110

Figure 40. A behaviour tree can be seen as a structure made up of sequences which are run by selectors 114

Figure 41. Basic sequence structure, where actions are preceded by a collection of preconditions 115

Figure 42. A bad design can lead to useless hints 116

Figure 43. Using hint conditions we can overcome problems caused by preconditions 117

Figure 44. Behaviour controllers maintain a hierarchy of behaviour trees that will be run concurrently 119

Figure 45. Structure of a Hinted-execution Behaviour Tree 121

Figure 46. HeBTs are run from the top-down. This means low-level trees have been modified by the time they get executed 122

Figure 47. Depending on the order the children of a parallel get run, different hints would be sent to a lower-level tree, so we would get different results from the final behaviour 127

Figure 48. A simple BT that will make an AI face the player indefinitely 134

Figure 49. Our first BT will make entities face us, but they will maintain their positions 134

Figure 50. Condition tree we will export to Lua 141

Figure 51. A complex behaviour tree that will make our agents behave autonomously 144

Figure 52. High-level tree that will make our AI be a kamikaze 148

Figure 53. After receiving a 'do not cover' hint, our tree moves its cover branch to the last, and least priority, position 149

Figure 54. High-level tree that will make our AI be a coward 150

Figure 55. Precondition used by our "cover" branch 150

Figure 56. Adding an npc_thesis to a level in Hammer 152

Figure 57. Main window of HeBT Editor 156

Figure 58. Options present in the main menu 157

Figure 59. Structure of a document 158

Figure 60. New nodes can be added to a BT just by drag-and-dropping them from the library 158

Figure 61. Structure of a behaviour tree in the editor 159

Figure 62. Simple tree we will serialise 161

Figure 63. Example node library 164

Figure 64. Some nodes expect some parameters (properties) that can be defined via the editor 165

Figure 65. Structure of a condition tree in the editor 165

Figure 66. Interface of the condition tree editor 166

Figure 67. The editor allows us to create low-level or high-level trees 167

Figure 68. A simple selector, as shown by the editor 167

Figure 69. Naming a selector's branch will expose a new hint to higher-levels 167

Figure 70. Branches that have exposed a new hint are marked in a different colour in the editor 168

Figure 71. Exposing new hints using hint conditions is as simple as adding one of these nodes to a condition tree and defining the hint we want to check 168

Figure 72. We must define which hint a condition hint will check 169

Figure 73. Once a hint is defined, the editor will reflect the change 169

Figure 74. The node library our high-level tree would use. The hints exposed in the base tree are automatically added to the library 169

Figure 75. Hint nodes can send different states of a hint 170

Figure 76. Structure of our communication server 170

Figure 77. Debugging a tree in the editor 174

Figure 78. Base behaviour tree used in our example 178

Figure 79. Precondition that will decide whether to make the AI run to a cover position 179

Figure 80. Precondition to prevent AIs from looking for cover positions when they are already hidden 179

Figure 81. AI running away while trying to cover 179

Figure 82. We have modified our main selector so it is able to receive COVER hints from higher-levels 180

Figure 83. Patrol branch 180

Figure 84. Assertion used by our patrol branch 181

Figure 85. Still unaware of our presence, this agent is just following its patrol route 181

Figure 86. Our attack branch 182

Figure 87. Assertion controlling the execution of the attack branch 183

Figure 88. An AI trying to find a cover position, but still facing us. We can also see another AI at the back, ready to start its attack behaviour, as it has just spotted us 183

Figure 89. Basic idea behind the disguise system 186

Figure 90. Our first complex high-level tree 187

Figure 91. An AI ignoring us after being hinted to PATROL 188

Figure 92. Final high-level tree that models the new feature..... 189

Figure 93. Condition that will decide whether to clear the PLAYER_DISGUISED flag 189

Figure 94. Base BTs controlling the different classes of AI in our army 195

Figure 95. Tyrant high-level AI that never allows individuals to retreat..... 195

TABLE OF TABLES

Table 1. Actions defining the AI controlling our guard 56
Table 2. Plans created by SHOP for our problem 62
Table 3. Comparison of the different behaviour-modelling techniques studied 72
Table 4. Possible results of the example sequence 91
Table 5. Execution of our sample parallel node 93
Table 6. Possible results of our selector example 94
Table 7. Results obtained using the different operation modes of condition nodes 99
Table 8. Possible results of a basic loop node 101
Table 9. Possible results of a conditional loop node 102
Table 10. Possible results of a "Run until succeeded" node 102
Table 11. Possible results of a "conditional execution" node 103
Table 12. Possible results of a "Not" filter 104
Table 13. Possible results of an "Ignore failure" filter 104
Table 14. Some examples of how a list of branches is split into two lists, depending on priority 111
Table 15. Process followed to build a tree from a Lua script 140
Table 16. HeBTs get the highest score in our study, mainly for the very good extensibility they offer 191
Table 17. Scores obtained by our base behaviours for the different classes of players 193

Chapter 1. INTRODUCTION

The Oxford Dictionary defines 'game' as "an activity engaged in for amusement" or "a form of competitive activity or sport played according to rules". Game, a fundamental part of human experience, has been present in every culture, since the very beginning. Maybe because of that, the birth of technology caused the creation of new ways to play: videogames.

Early videogames were first created in the 1970s. Since then, their industry has been experiencing an exponential growth, far outpacing movies and music (Wallop, 2009; Bangeman, 2008) in the last years.

Complexity has increased in a similar manner. From games such as SpaceWar! (Russell et al., 1962) or Pong (Atari, 1972), to the massive universes presented in recent games such as Red Dead Redemption (Rockstar, 2010) or Assassin's Creed 2 (Ubisoft, 2009), the evolution of computers has provided creative teams with new tools to achieve their visions.

Over many years, most of the technical advances were used to produce better and more realistic graphics, but we have got to a situation in which improvements in graphics are not that significant from one year to the next. Games today really need an outstanding characteristic to lead them to success, and AI is the perfect candidate to achieve this (Nareyek, 2004).

1.1. AI in videogames

Game AI systems differ from traditional, academic ones in the objective they are trying to achieve (Baekkelund, 2006). Academic research usually looks for problem solvers, the so-called Good Old-Fashioned Artificial Intelligence (GOFAI) (Haugeland, 1985), while in games we are trying to build an apparent intelligence (Bryant, 2006). Games need to create the illusion of human behaviour (Spronck, 2005), that is, present an agent which appears to behave intelligently when seen from the player's point of view.

In the end, we are trying to simulate different behaviours and the way agents decide how to respond to different stimuli appropriately, providing a rich gaming experience.

So, as AI is a fundamental part of gameplay, what AI engineers are trying to build is not a generic solution to abstract problems, but something focused on playing a very specific role in a particular game. That said, although the problem that AI is trying to solve is very dependent on the game it is being built for, some systems share a basic common structure, as shown in Figure 1.

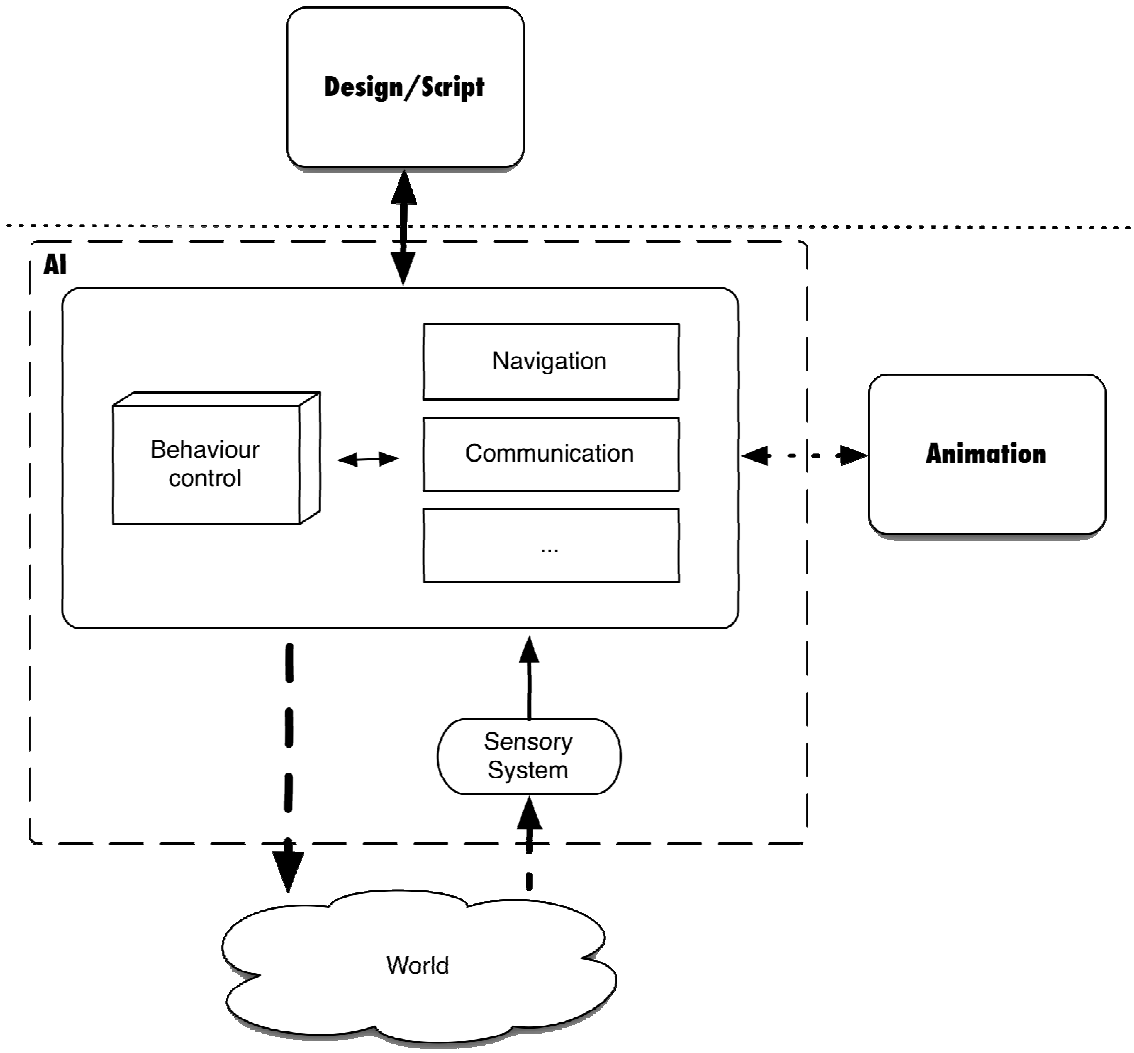


Figure 1. Structure of a game AI system

In the average real-time videogame we normally have a set of agents/NPCs[1] that interact with the world of the game. These agents:

- Learn facts about the state of the game. This is usually called a **sensory system**.
- Show complex behaviours and decide what should be done in every moment. These actions will modify the state of the world and use the functionality provided by other lower level components, such as:
    - Moving from one position to another, intelligently avoiding obstacles and using near-optimal paths in the worst scenario. This **navigation system** can be broken down into different parts, such as a *pathfinder* or a *locomotion system*.
    - Communicating and sharing information with other agents, so co-ordinated behaviours can take place.
- Play different animations to transmit information about their state to players.

As a separate, higher-level layer, we also find an interface with scripts and designers. Whether or not an AI is going to provide a fun experience depends, to a great extent, on them.

1.2. Game design and AI

Videogames are a type of software that benefits from changes, and prototyping is necessary to develop fun. Building a sophisticated and fun AI that is capable of being the outstanding characteristic we are looking for requires collaboration between programmers and designers (Orkin, 2002). Also, many different ideas should be tried, and the best ones should be kept for the final product. The question is: how can we optimise this process?

[1] Non-Playable Character, a common name for autonomous entities in videogames.

Almost every game requires this kind of communication that basically converts the AI into an intelligent black box, as shown in Figure 2, which designers can control to build the game experience they are looking for.

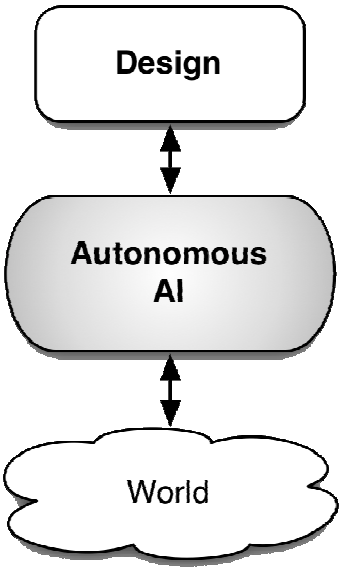


Figure 2. The autonomous AI system is controllable, in most games, via a high-level interface.

Depending on the grade of control offered to designers (Snavely, 2006), we can classify systems as:

- **Brute-force systems**, where every action and reaction the AI undertakes is hard-coded in the game.
- **Data-driven systems**, where non-engineers take an active part in the creation of the AI, not only providing programmers with feedback, but also modifying systems themselves. Data-driven systems are often referred to as *scripted systems*.

Videogames are very complex systems, and changes in an advanced stage of their development are very risky –and can be very costly-, but sometimes they are necessary in order to improve the game experience. While brute-force systems would require potentially dangerous changes in the code base, data-driven systems by-pass this problem, and allow a better usage of resources.

A second classification can be focused on analysing the amount of functionality exposed to scripts (Rabin, 2004). In this case we can speak of:

- **Level 0**, where everything is hard-coded in the source language. It is the equivalent to the brute-force systems.
- **Level 1**, where data in files specify stats (such as personality traits) and locations of agents.
- **Level 2**, where the AI is completely controlled by scripts during cutscenes.
- **Level 3**, where tools or scripts describe lightweight logic.
- **Level 4**, where heavy logic resides in scripts that rely on core functions written in the source language.
- **Level 5**, where everything is coded in scripts.

In order to optimise the process of creating an AI system, we must decide how to divide responsibilities, as well as take into account the strengths of each component in the team. Exposing too much functionality can overwhelm designers, affect performance and produce static (or too predictable) behaviours (Laird, 2002), whilst not exposing anything at all could require a lot of changes, longer iteration times and increasing risks.

The ideal solution should offer autonomous AIs that can behave smartly without any external feedback, but that can also obey the designer's orders (Tozour, 2002).

1.3. Behaviour modelling

From a high level, we could consider a game AI as a collection of NPCs, or AI instances, that interact among them and with players to give the illusion of being alive. These actions performed by agents in a logical way are known as behaviours.

Behaviours work as managers of the resources available to the AI. They get information from other subsystems, process it, and decide how to act.

Many AI techniques have been used in commercial videogames to model and control behaviours. Among them, we can find solutions as diverse as Finite-State

Machines (FSM) (Buckland, 2005; LaMothe, 2002), Behaviour Trees (Isla, 2005), or automatic planners such as GOAP, (Yue & de-Byl, 2006). Each solution has its strengths and weaknesses, and not all of them are suitable for every type of game.

Another possible classification of behaviours can come from their level of predictability; that is how good they are at surprising the player with unexpected actions. We can say a behaviour is static if it always produces the same set of actions. On the other hand, a dynamic behaviour is not as foreseeable, as it might choose to resolve the very same problem in many different ways.

While dynamic behaviour may appear to be desirable, it might not be as good for linear games, or those that need to follow a very strict sequence of events: in these cases, a more predictable behaviour could offer a better experience. However, how predictable must behaviours be?

To fully understand this, we must study the problem from two different points of view. Let us use a simple example of a situation we can encounter in a hypothetical game. In this game, our AI controls guards that try to stop the player from stealing a valuable object.

From the players' point of view, our guard is an obstacle to get its objective, so they expect the guard to confront them. They have faced a guard previously in the game, and they know they will be attacked. So, armed to the teeth, they decide to step forward and kill him.

Let us stop at this point and analyse the situation. Our agent has just perceived the threat, and must decide what to do:

- As designers, we want the AI to be an **obstruction**: something players must beat in order to continue playing. This means the behaviour must be **predictable**, because the agent cannot just run away, as it would ruin the game experience.

- As players, however, we would like the AI not only to be a hurdle, but we want to **have fun**. Players would like to face an intelligent enemy, whose actions were not seen before, and thus shows an **unpredictable** behaviour.

How can we get a behaviour that is predictable enough for designers, yet dynamic and fun for players? In fact, this is the problem every game AI should always try to solve.

1.4. Adaptation

AI is a key part of many games, where their challenge comes from trying to beat the AI. Games are not a static activity, but an interactive one, which requires players to be profoundly involved in them. Ultimately, the consequence of this is that even if two players enjoy the same type of games, depending on other factors, such as how challenging a game is, they can be more or less attracted to the same product.

This is something that has been known since the very beginning of gaming, which rapidly spread a coarse or manual method of adaptation: **difficulty levels**. These can be implemented in many different ways. For example, we could make a game easier by reducing the amount of damage AIs can tolerate or by giving players more powerful weapons.

Building a groundbreaking AI requires a much more sophisticated approach; players can find a game much more appealing if they become part of it, and really think they have mastered it; so, rather than modifying values artificially, we could adapt our AIs' behaviours and tailor them to the individual requirements and preferences of each player.

This would also allow our AI to evolve as the game progresses, learning and improving itself (Charles & Black, 2004), presenting a better challenge and improving gaming experiences.

The question is, how can be create dynamic, adapted AI systems?

Chapter 2. OBJECTIVES

This thesis emerges from the need to answer the questions presented in the previous chapter, which came from professional experience and months of research.

Our initial objective was to build AI systems that can be adapted to players. We started our research looking for ways to model players and gather information so behaviours can be enhanced using this information, just to realise there was plenty of research about the topic (for example, Drachen et al., 2009; Houlette, 2004), but there was not an easy method to make AI systems use all this data.

So, at an early stage, we decided the best way to approach this work was to separate the research into two different phases, which were creating a behaviour modelling technique that allows dynamic changes to behaviours, and a player modelling system that makes use of these capabilities to produce final, adapted AI behaviours. Due to the massive amount of work this would require, we decided to focus this thesis on the former, and build a behaviour modelling and control system that would allow us to continue our research on player adaptation in the near future.

Our system is built around a simple, manual adaptation method: we do not look to adapt behaviours to players, but to allow development teams to improve AI behaviours dynamically, so they are better suited to create the best possible experience.

Our main objective is to develop a behaviour modelling technique that:

- Is capable of producing dynamic behaviours, but that can be easily controlled to produce more predictable results, if the game requires it.
- Allows prototyping, and provides enough tools to modify or create new behaviours without the need of any additional programming work.

The solution chosen, *Hinted-execution Behaviour Trees (HeBTs)*, is a novel evolution of regular behaviour trees. In this case, we have opted to use multiple

layers of them, which interact to allow higher-level trees to modify the normal execution of the tree immediately below in the hierarchy, as shown in Figure 3.

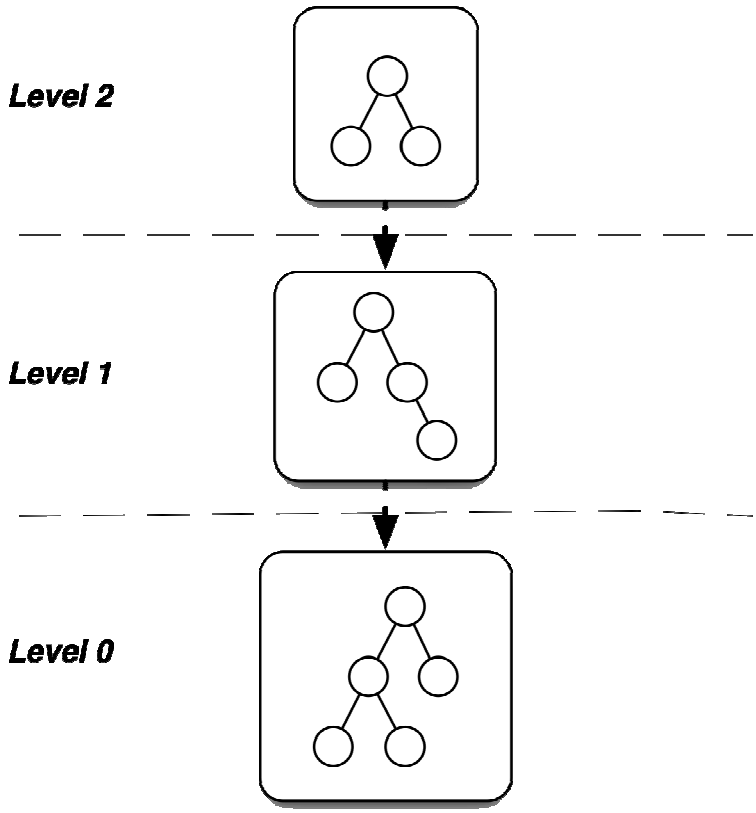


Figure 3. This multi-layered structure allows a higher-level tree to indicate the layer below how it should behave

This kind of structure would allow us to study different options to adapt the game to its players, as we will study in detail later in Chapter 15. The basic idea behind it is that we could try to adapt our behaviours using our player model just by building a new higher-level layer.

We aim to fulfil the following objectives, which come from the need to support adaptation, as well as meet the requirements of commercial game development:

1. **Autonomy:**

   1.1. AIs controlled by the system are fully autonomous. This means they will be able to respond to external stimuli appropriately.

   1.2. AIs will, at the same time, be able to accept orders from design, so the gameplay experience can be enhanced.

2. **Extensibility and reusability:**

   2.1. The system must allow designers and programmers to create new behaviours based on existing ones.

   2.2. Behaviours must be easily modifiable, so changes in the game design can be translated to them quickly.

   2.3. The system must provide its users with a set of building blocks that can be shared between behaviours.

   2.4. New functionality can be added by creating new building blocks.

3. **Ease of use**

   3.1. Behaviours must be easily understandable by both technical and non-technical members of the team.

   3.2. Behaviours must be visually debuggable, so errors in the logic can be spotted swiftly.

   3.3. Designers must be able to prototype and try new ideas easily and without requiring any additional programming work, which can be costly and risky. This will enable a more effective methodology, as final users are also involved in the development of the system.

4. **Abstraction**

   4.1. The system must be able to, potentially, control any entity in any type of game.

   4.2. Most of the functionality must be game-independent, and work as a separate library.

Due to the highly practical nature of this study, a big part of the work and research presented in this thesis comes from the experience and knowledge gained during the development of a prototype of the system. Creating a whole game from scratch is not a trivial task, so the prototype is built on top of Half-Life 2's public SDK and tools. The game's original AI was replaced by our own implementation of an HeBT

system. A complete editor (which is able to communicate directly with the game) was also implemented.

Chapter 3. ORGANISATION

In this chapter, we present an overview of the organisation of this book, which shows what kind of research and work has been done to support this thesis. We also present what steps were taken to produce it, from its design to building a fully-functional prototype.

3.1. State of the art

This work is the result of several years of research in the field of multi-agent systems, always focusing on game AI, and, particularly, how it can be improved moulding behaviours to offer better experiences.

Producing adaptive behaviours required finding a system that was easy to use and extend, so we started our research studying different solutions that have been used in commercial games.

We have classified them into three groups, and dedicated a chapter for each of them: "*Chapter 4. Reactive solutions*", "*Chapter 5. Automatic planning*" and "*Chapter 6. Reactive planning*".

3.2. Problem and solution

After studying the different options that were available, we tried to determine whether they were suitable for what we were trying to achieve or not. We then decided Behaviour Trees were almost what we needed, but they still presented some problems. We study this in "*Chapter 7. Defining the problem*".

Then, we needed to propose a high-level design for the solution, which is presented in "*Chapter 8. Overview of the System*", which served as a starting point for the system we were going to implement. As HeBTs are based on regular behaviour trees, we decided the best way to approach the problem was to build our own implementation of a BT system, which we study in "*Chapter 9. Building a Behaviour Tree system*". Then, we expanded it with our new ideas in "*Chapter 10. Expanding our BT: Hints*" and "*Chapter 11. Multi-level architecture*".

3.3. Applying HeBTs to a real-world example

Once we had defined the foundations of a Hinted-execution Behaviour Tree system, we needed to check whether commercial games would be able to use it. To do so, we implemented a game prototype, which we present in "*Chapter 12. Applying HeBT's to a game*".

One of the objectives of this thesis is to build a system anyone can use, so designers and scripters can contribute to the final AI. In order to fulfil this point, we built a fully functional editor, studied in "Chapter 13*. Visual editing*".

We also implemented a complete example, so we could study what HeBTs would contribute to games, as shown in "*Chapter 14: Why HeBTs are good to AI programming: A practical example*".

Lastly, we present a final chapter, "*Chapter 15. Conclusions and future research lines*", where we summarise what the advantages of our new system are, as well as we look at the future of this new technology, and how we could improve it.

State of the Art

In this section, we will study different solutions that have been used in games to model and control behaviours, so we provide readers with a complete view of the current state of the art in the field of this research.

Chapter 4. REACTIVE SOLUTIONS

Reactive solutions are the techniques that are the most commonly used in games. They allow behaviours to be described as a series of states that can be reached through different transitions, and can also be known as state-based solutions.

Their key characteristic is that they cannot look into the future to try and achieve a goal; instead, their execution is tied to the present, and they only modify their state to respond to changes in the environment (i.e., they work reactively).

4.1. Finite-State Machines

Many games implement AI agents as Finite-State Machines, as they are an easy way to represent how an object can change its state over time to respond to external events.

Formally (Hopcroft & Ullman, 1979), an FSM is defined as a five-tuple

$$(\Sigma, S, s_0, \delta, F)$$

where:

Σ	is a finite set of symbols denoting possible inputs;
S	is a finite set of symbols denoting states;
$s_0 \in S$	is the initial state;
δ	is the state-transition function: $\delta : S x \Sigma \rightarrow P(S)$;
$F \subset S$	is a set of final states. It can be the empty set.

We then define an FSM as a set of states and transitions between them, and they can be easily represented as directed graphs called *state transition diagrams*. The machine starts in the initial s_0 and depending on the inputs received and the transitions that had been defined, it will move to new states, until a state $f \in F$ is achieved. So, because of this structure, it is clear that only one state can be active at any time.

In order to create a new behaviour, game designers and programmers have to break down what they expect the AI to do into these elements.

For instance, let us say we would like to model a guard. We want our guard to keep his position, unless he is suspicious. In that moment he should look for threats and, in the event that they are found, eliminate them. This will produce 3 states; 'Idle', 'Suspicious' and 'Eliminate'. From 'Idle', our agent can become 'Suspicious' in case he hears/sees something), and he can get back to 'Idle' if he decides there is no threat. He can also decide there is a threat to eliminate, so we have two possible transitions from the 'Suspicious' state. If the agent was eliminating a threat and he accomplishes his goal, it will get back to 'Idle'. So we will end up with an FSM similar to that shown in Figure 4. As long as our agent stays in a certain state, it will execute the actions encoded in it.

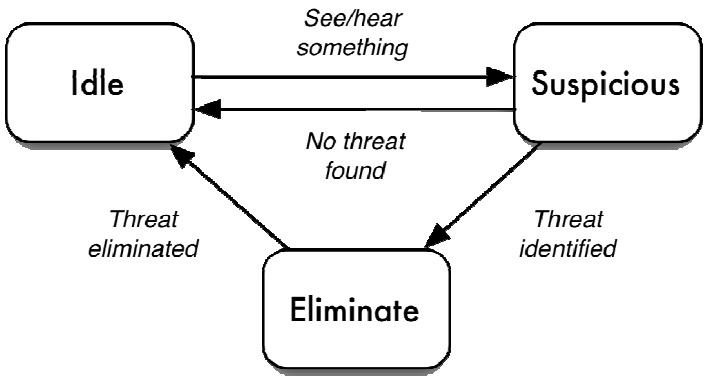


Figure 4. Basic finite-state machine to control a guard NPC

Finite-state machines set the basis for every other state-based technique, which are improvements to this basic system used widely in computer engineering. While they are easily understandable and very powerful, they do not really scale well (Diller et al., 2004). Working with large machines is complicated, as adding or removing states or transitions can affect many parts in the machine (depending on the existing connections among them).

4.2. FSM with extended states

An FSM, by definition, will keep executing its active state until a transition is triggered, which would make the machine run a new state, and so on. Sometimes it

is desirable to execute some actions just when accessing the state, and others when the FSM is leaving it (Fu & Houlette, 2004).

In effect, each state works as a little FSM with three states: 'On enter', 'On exit' and 'Running', as shown in Figure 5. This could be viewed as a special case of Hierarchical-State machines, studied later in this chapter.

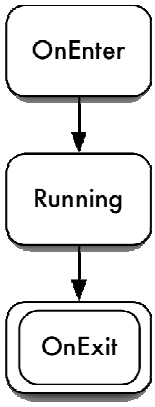


Figure 5. Each state in an FSM with extended states works as a small FSM with three states

4.3. Stack-Based Finite-State Machines

One of the problems FSM present, particularly a non-deterministic one (i.e. an FSM where two or more states offered transitions to get to a common one), is the impossibility to create a valid transition that guides the execution back to its previous state. Continuing with the example of a 'guard AI, we could face this problem if we added a new 'Patrol', just as we illustrate in Figure 6.

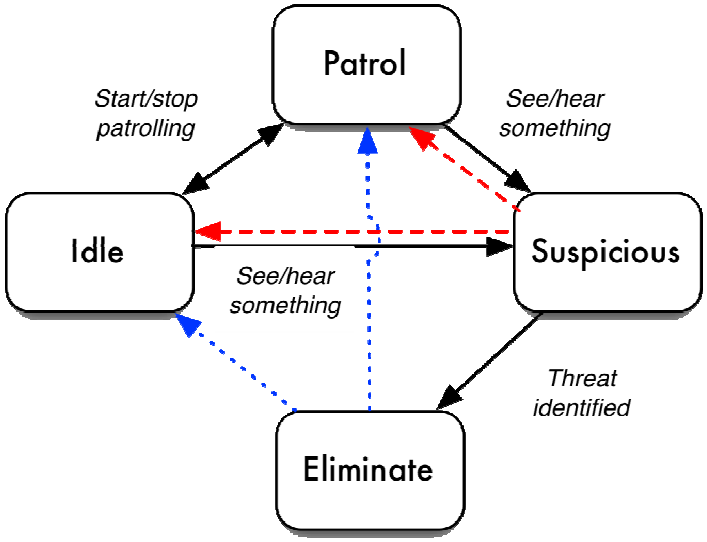


Figure 6. Adding a new state to an FSM can start causing some design problems

In this case, adding transitions to decide what to do after 'Eliminate' or when the 'Suspicious' state does not lead to 'Eliminate' is not trivial anymore: some sort of memory is needed.

Adding a state stack solves the problem. In a stack-based FSM, transitions do not necessarily have to produce an immediate change of state. Instead, we find three different possibilities (Tozour, 2004):

- The target state is **pushed onto the top** of the stack.
- The current state is **popped off** the stack.
- The current state is popped off the stack and immediately **replaced** by the target one. A stack-based FSM providing only these transitions would behave as a regular state machine.

With this new set of rules, the guard's behaviour can be modelled as shown in Figure 7.

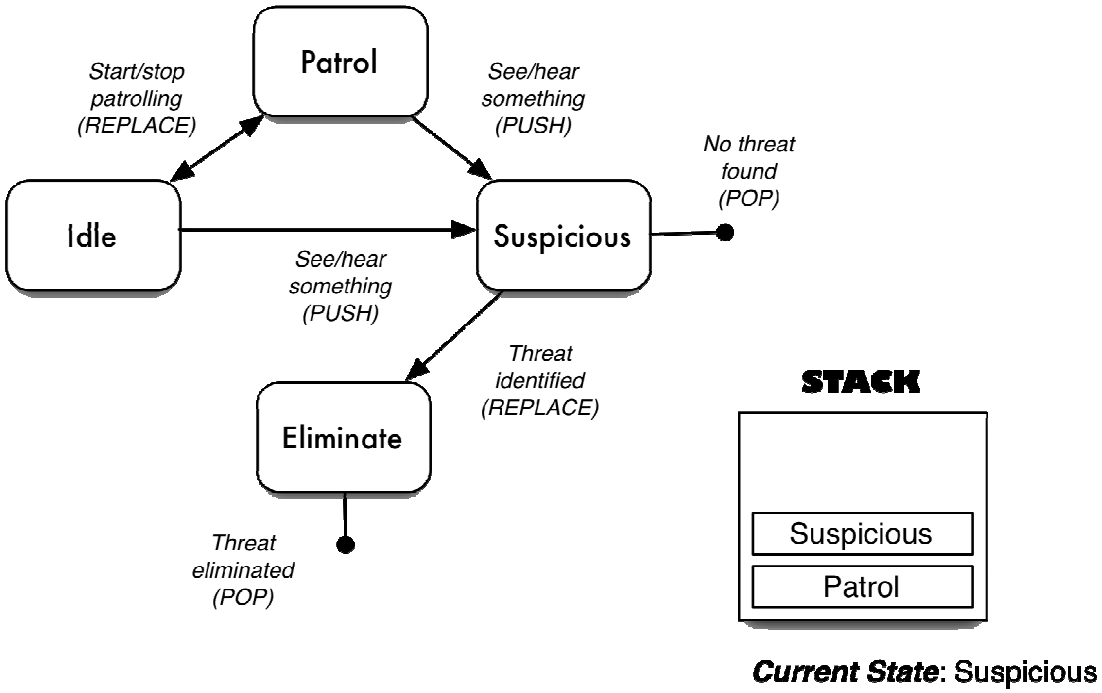


Figure 7. A stack-based FSM, showing the state of its stack when 'Suspicious' is the active state

4.4. Hierarchical Finite-State Machines

As stated before, while FSMs are a very powerful tool, their biggest disadvantage is that they do not scale well. Complex behaviours require state machines that grow rapidly, using a lot of states and even more transitions.

Hierarchical Finite-State Machines (HFSM), first presented as *statecharts* (Harel, 1987), try to deal with this complexity problem grouping states, so some transitions can be shared (*generalised transitions*). They were designed as a solution to manage complex systems via a graphical interface that allowed zooming in and out of groups of states (*super-states*).

Expanding this idea, behaviours can be broken down into a set of main states, which are then recursively subdivided into their own FSMs (see Figure 8). The biggest advantage resides in the separation of the logic of the different states that can deal with their own data now. These improved HFSMs are also known as *Behavioural Transition Networks* (BTNs).

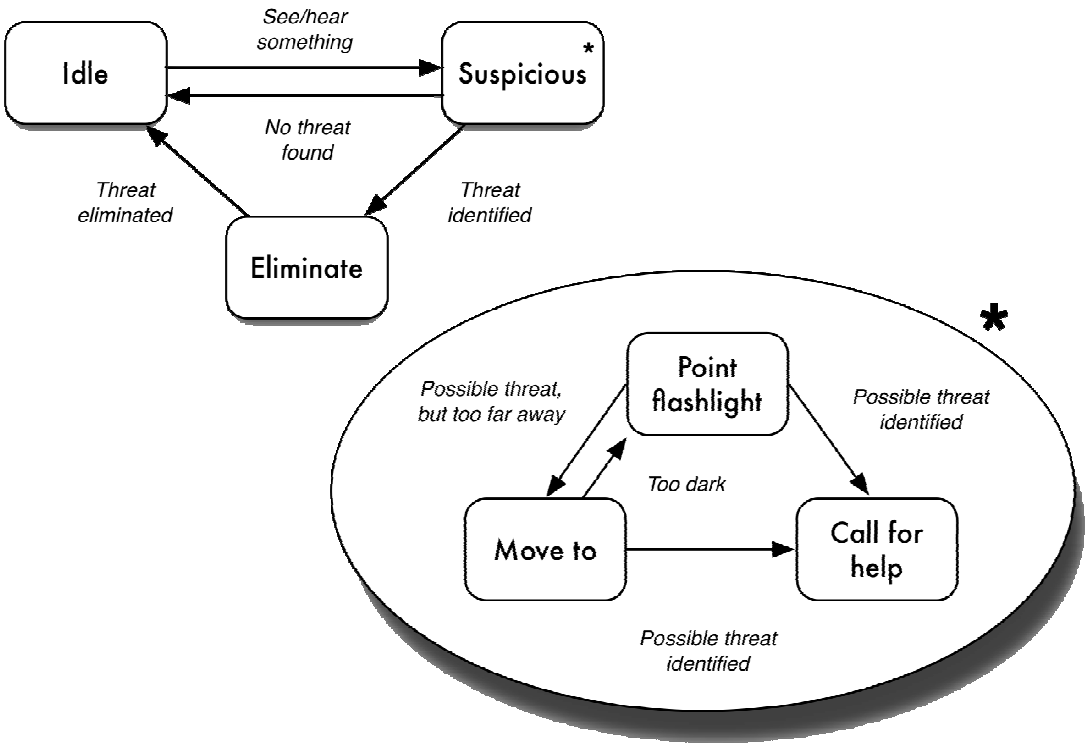


Figure 8. HFSM, where the state 'suspicious' has been expanded to a new FSM

BTNs are made of *nodes* (actions the agent can perform) and a set of *transitions* (Houlette et al., 2001). There is an initial node and, as with the rest of the state-based solutions presented so far, only one action can be active at a time. Each transition defines a decision process as a logical function, whose activation results in a change of node. As they are hierarchical, any node can be described as a new BTN.

The use of the hierarchy allows different team-members to focus on different parts of the behaviour, so these parts can be tweaked individually. Any change to enhance a sub-behaviour only affects that particular sub-FSM, and will not require any change in higher levels. Also, complex states can be re-used among different state machines, so creating new behaviours is highly simplified.

4.5. Concurrency in state machines

When it comes to modelling a problem using a reactive solution, sometimes we could find we need to have different components working together to achieve our

goal. This concurrency means allowing some form of communication between different machines, so they can be synchronised (Alur et al. 1999).

Traditional FSM are not prepared to support this communication, so extensions to the model have to be used. The simplest of them consists of building a message system, so states can send and receive events (Rabin, 2002).

The biggest challenge is to synchronise access to shared resources correctly (Champandard, 2010-1; 2010-2). To do so, we could use traditional synchronisation primitives, such as semaphores (Downey, 2005).

4.6. Concurrent, Hierarchical Finite-State Machines

HFSMs try to simplify regular FSMs adding the possibility to group states into super-states. However, they are not very well suited for situations where we need to take into account various different aspects of the environment. HCSMs (Cremer et al., 1995) try to solve this problem presenting hierarchies of concurrent state machines.

HCSMs drop the concept of state, and make an HCSM the basic element to build more complex machines. Each HCSM is basically defined by a list of child machines, a set of transitions, an activity and a pre-activity function, a set of input variables, a set of outputs, and a set of variables (also called *control panel*).

A *leaf* HCSM is the basic one, and it does not have any children or transitions. Apart from this, we can also have *sequential* HCSMs, where only one machine is active at a time and execution is transferred between machines via transitions, and *concurrent* machines, where all of them are run at the same time, and there is no transfer of execution.

One of the differences between an HCSM and a standard FSM is the usage of *activity functions*. These functions are in charge of producing some output values for the machine. If we are using a *sequential* machine, the output will normally be the one from the active child; on the other hand, a *concurrent* machine would have to combine the outputs of its children. Pre-activity functions are in charge of converting the parent machine's input and pass it to its children.

HCSMs also have a control panel, which is a set of variables that can be modified dynamically to obtain different outputs. For example, machines can send messages to one another and modify these values.

The algorithm to run an HCSM is shown below:

```
ExecuteHCSM(hcsm)
{
  ExecutePreActivity(hcsm.pre_activity_function);
  transition_to_fire := SelectTransitionToFire(hcsm);
  if (transition_to_fire) then
    hcsm.active := transition_to_fire.to_state;
    ActivateState(hcsm.active);

  for each active child m of hcsm do
    ExecuteHCSM(m);

  return(ExecuteActivity(hcsm.activity_function));
}
```

HCSM are more indicated to control behaviour in a simulation environment, rather than defining agent behaviours. For instance, they were used in Valve's Left 4 Dead (Valve, 2008), as part of their AI Director. This system allowed orchestrating the gameplay, generating different situations, depending on the state of the game, to make it more fun and interesting.

4.7. Petri nets

Petri nets are another possibility, when it comes to modelling these complex systems. A Petri net is an abstract, formal model of information flow (Peterson, 1977). They can be represented using graphs, which contain two types of nodes called *places* and *transitions*. Nodes are connected using directed arcs, and there is a restriction that only allows places to be connected to transitions, and transitions to places, but no node can be connected to another of the same type.

The execution of a Petri net is controlled by the position and movement of tokens (D'Angelo, 1983). These markers reside in the *places* of the net, and are moved around it by transitions. These can only be fired if they are enabled. In order for a transition to be enabled, it needs that all its inputs, that is, all the places that are connected to it, have a token on them. This way we can model concurrent behaviours which are synchronised.

For example, let us add a simple concurrent behaviour to our guards. We want them to be able to attack using a mounted gun, but only one is available: we are facing a mutual exclusion problem. We can solve this by using P/V operations, and model it using the Petri net as shown in Figure 9.

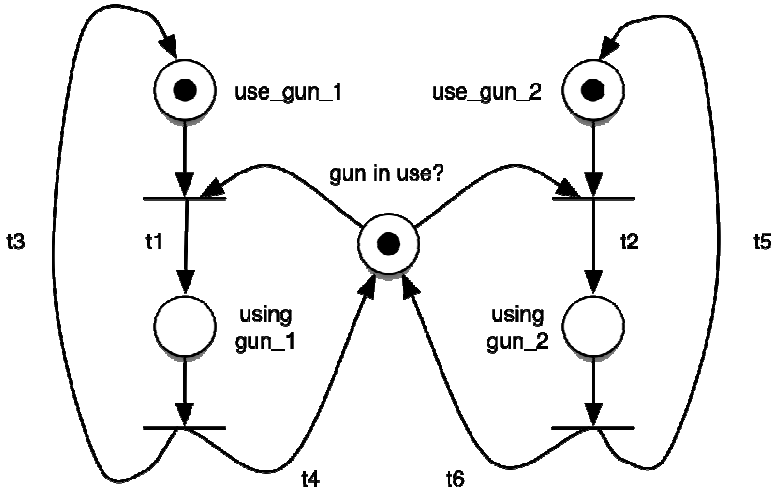


Figure 9. A Petri net that models our simple concurrent behaviour

This net is quite simple. In our initial state, both AIs decide to use the mounted gun. This is represented by the tokens present in "*use_gun_1"* and "*use_gun_2"*. The mounted gun is free, and we represent this as a token in "*gun in use?*". At this point, both *t1* and *t2* are enabled (all their inputs have a token on them), so we can fire either. Let us say *t1* is the chosen one: after it is executed, our net will look like that shown in Figure 10.

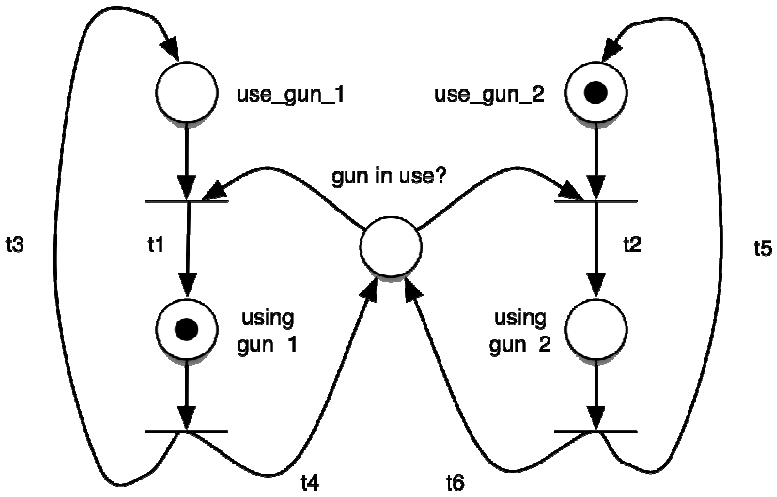


Figure 10. Our Petri net once one of the AIs takes control over the mounted gun

We can see tokens in "*use_gun_1"* and *"gun in use?"* have been removed, while a new one has been created on *"using_gun_1"*, indicating our first AI has taken control over the gun. Because of these changes, only *t3* and *t4* are available: *t2* is no longer enabled and thus our second AI cannot use the mounted gun. When the first AI decides it wants to release the gun, we will get back to the initial net (Figure 9), and both can compete for the resource once again.

Chapter 5. AUTOMATIC PLANNING

While state-based solutions represent a very good way to organise, describe and maintain behaviours, these can become predictable, because their logic remains static. This can be disguised as long as multiple choices are provided to AIs, but it would represent a lot of design (and possibly programming) work, as all the possible variations of a behaviour should be taken into account beforehand (Young et al., 2004).

On the other hand, goal-driven NPCs, and more precisely, those using automatic planners to achieve their goals, present much more dynamic behaviours with less input from designers (Wallace, 2004). If we define a goal as a condition an agent wants to satisfy (Orkin, 2004-1), planners will build a list of actions to take in order to transform the AIs perception of the state of the world from A to B. This means planning will not just decide what to do, but how to do it (Champandard et al., 2009).

5.1. GOAP

Goal-Oriented Action Planning (GOAP) is a decision making architecture that defines the conditions necessary to satisfy a goal, as well as the steps required to accomplish it in real time. This technique was first used by "F.E.A.R." (Monolith, 2005; Orkin, 2006), and several other games, such as "S.T.A.L.K.E.R." (GSC, 2007), "Fallout 3" (Bethesda, 2008), "Empire: Total War" (Creative Assembly, 2009) or "Just Cause 2" (Avalanche, 2010) followed.

GOAP is similar to STRIPS, an automatic planner developed at Stanford University in the 70's (Nilsson & Fikes, 1971). STRIPS is made of two components: goals and actions. It uses a symbolic representation of the world, so it is capable of achieving a desired state of the world (goal) by applying the correct actions. These are defined as preconditions, an 'Add' list and a 'Delete' list. These two lists represent the modifications the action applies to the world, first deleting what is in its 'Delete' list and then adding the knowledge present in its 'Add' one.

As with any architecture implying planners, GOAP requires a different way to approach the decomposition of behaviours, as solutions are not built by programmers, but generated automatically from the declaration of the different actions that will help to transform the world state.

To illustrate this, and continuing with the example presented in previous sections, we could model the behaviour of our guard using a top-down approach:

- First, the goal of the agent must be described. In this case, the guard's goal is to maintain that there are no threats in sight. It can be expressed as:
    - **Not Present(Y)**, where Y is the threat we are trying to eliminate.
- Now, we just have to declare some actions, as many different possibilities as we want to provide the AI with. So, in order to eliminate a threat, the agent can either kill it or frighten it away.
- We must now provide our guard with more actions to break down the ones presented in the previous point. So, for instance, a threat can be killed if the agent has a weapon, or it could be scared away by switching the alarms on, and so on.

This process will produce the set of actions shown in Table 1.

Action	Preconditions	Effects
Kill threat(X,Y) – Makes X kill the threat Y	WeaponAimed(X, Y)	Not Present(Y), Not WeaponAimed(X, Y)
Frighten threat away(X,Y) – Makes X frighten Y away	Present(Y)	Not Present(Y)
Aim weapon(X,Y) – Makes X aim its weapon at Y	Present(Y)	WeaponAimed(X, Y)
Investigate threat(X,Y) – Makes X search and find threat Y	Suspicious(X,Y), Detected (Y)	Present(Y), Not Suspicious(X,Y), Not Detected(Y)
Become suspicious(X,Y) – Makes X become suspicious of threat Y	Not Suspicious(X,Y),Detected(Y)	Suspicious(X,Y)

Table 1. Actions defining the AI controlling our guard

Plan formulation is the key to get a more emergent behaviour (Wallace, 2006), which is less predictable, and works autonomously, without any additional programming work. It, in fact, consists of a search through a search space of different states of the world, which are a result of applying the different actions available. Actions can be assigned different costs (O'Brien, 2002), so the actual plan varies depending on those values. Figure 11 shows how two different plans can be achieved, depending on the aggressiveness of the agent, providing the initial state of the world is 'Detected(Threat), Not Suspicious(Guard)'.

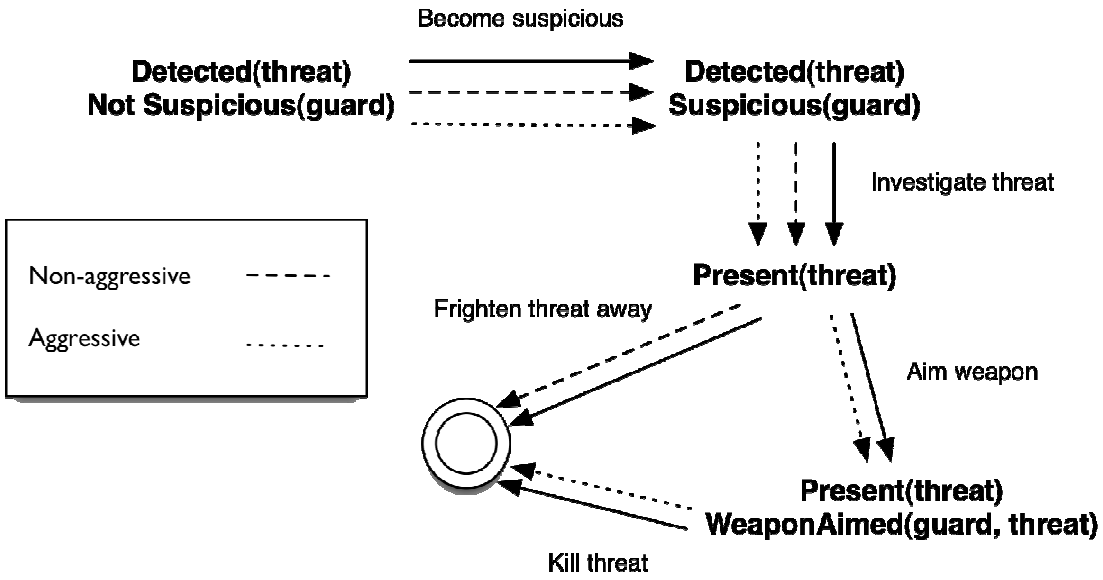


Figure 11. Different plans can be created depending on the costs of the actions used

We can obtain different plans, all of them valid to satisfy the AIs goal, as long as we define a valid initial state of the world. Once it is defined, the search consists of creating a tree in which each node represents a state of the world. Each node will be expanded using every action that is applicable, with the knowledge we have at that point, and this process will carry on until the goal state is found.

The true power of automatic planners is not very clear in the example we have been using in this thesis, because of its simplicity. However, this is very noticeable when complex behaviours are built, or new conditions or functionalities need to be added to them.

Let us expand the example of our guard. Until now, we have been working knowing that our agent will always have a weapon. Let us say he has not, but he has to get one from the world to attack his threat. In a state-based solution, we would have to modify the actual structure of our FSM-like model to reflect this: this means, new nodes and transitions would have to be added. Using GOAP, we would only have to add new actions and modify some preconditions to make it work. Let us say now that we are in a late stage of the project, and designers have decided our guard can also use melee attacks, which do not need weapons, but just some proximity to the target. Again, our structures would have to be modified and tested, whereas GOAP could use the new attack method in its plans simply by defining the appropriate

actions. Not happy with this, our designers decide to add a mounted gun near our post, and they want our guard to be able to use it. Just by adding some more actions, the new gun would be available to be used in plans, whereas an FSM at this point would start to become hard to maintain and debug.

Although this architecture leads to very desirable behaviours (as they are quite dynamic, and AIs can perform actions they were not directly programmed to do), it also has some drawbacks:

- Planning could become bottlenecked (Kelly et al., 2008), depending on the frequency with which agents decide to re-plan. Depending on the number of actions, the search space can grow very quickly, leading to combinatorial explosion. Algorithms like A* (Hart et al., 1968) can help pruning the tree of different possibilities.

- Sometimes, it could be possible that an agent cannot satisfy its goal. With the information provided in the example presented, the agent's goal should always be to not have any threat in sight… but what should it do if it does not detect any threat at all? Using an FSM, the NPC would simply be in an 'Idle' state, but behaviours like these can be harder to describe using GOAP.

5.2. HTN

Hierarchical Task Networks (HTN) are another form of planning, where relations between actions can be expressed as task networks.

A task network is composed of tasks, where each one is described by a name and a list of arguments, and a set of constraints. Tasks can be of three different types:

- *Goals*, which are desired states of the world.

- *Primitive*, which represent concrete actions, and thus can be performed directly.

- *Compound*, which need to be subdivided, and represent changes that cannot be expressed by simpler tasks.

The input of an HTN planner is a triple P={d, I, D}, where *d* is a task network, *I* represents the initial state of the world and *D,* planning domain is a pair D={Op, Me}, where *Op* are operators, which denote the effects of primitive tasks, and *Me* is a set of methods, which indicate how compound tasks must be solved. Methods allow us to decompose complex tasks into their own task network, so a method *M(α, d)*, where *α* is a compound task and *d* is a task network expresses that in order to achieve *α*, *d* must be achieved.

HTN planning is done by reduction. This means, that the planner will try to decompose all the compound tasks into primitive ones using the available methods, and then execute each of these using the required operators (Erol et al., 1994). A plan will thus be a list of actions to satisfy the task network, providing we are at the initial state.

Let us express the example used in previous sections as an HTN network. In this case, our initial network would be composed by just one task 'Eliminate Threat(X)'. As it is a compound task, we need to define some methods to decompose it.

To define our example, we will use the notation (based on Lisp's S-Expressions) used by SHOP (Simple Hierarchical Ordered Planner) (Nau et al., 1999), which is a domain-independent, total-order HTN planning system. This representation follows the standardisation of planning languages called PDDL, Planning Domain Definition Language (Ghallab et al., 1998).

In SHOP, we can define knowledge using Horn clauses with a Lisp-like syntax. This knowledge includes states, which are atomic expressions or axioms, which have the form **(:- head tail1 tail2 ... tailn)**. Axioms express that *head* is true if any of its tails are true. Let us express the knowledge of being suspicious (that is an axiom), as we know our guard will be suspicious if he hears a sound or perceives movement.

```
(:- (suspicious ?x)
      ( (in-sight ?x ?y) (heard ?x ?y) ) )
```

We can also create an axiom to determine when a threat can be frightened away, and whether an agent is aiming his/her weapon at a target.

```
(:- (can-frighten-away ?x)
      ( (< (height ?x) 150) ) )

(:- (is-weapon-aimed ?x ?y)
      ( (< (distance-to ?x ?y) < 100 ) (< (angle (create-vector ?x ?y)
(heading ?x) ) 5 ) ) )
```

A method is defined as **(:method h C T)**, where *h* is the compound task the method decomposes, *C* is a set of preconditions and *T* is a task list. A method expresses that if the preconditions in *C* are met, then *h* can be accomplished executing tasks in *T* in order.

In our example, we can subdivide our main task, 'Eliminate threat', into two different ones: 'Kill threat' and 'Frighten threat away', so we would have two different methods to satisfy our main task.

```
(:method (eliminate-threat ?y)
          ()
          '( (prepare-weapon ?x) (!shoot ?x ?y) ) )

(:method (eliminate-threat ?y)
          ( (can-frighten-away ?y) )
         '( (!frighten-away ?x ?y) ) )
```

To clarify the syntax of a method, we can express the first method shown above, as a method to solve a compound task called 'eliminate-threat', which has no preconditions (), and which has two child tasks that must be executed to resolve the problem: 'prepare-weapon' and 'shoot'. Primitive tasks are preceded by a '!'. 'Prepare-weapon' is a compound one, and we will have to define a method to satisfy it. The ' operator preceding the list of tasks to be executed expresses that the lisp interpreter shall not evaluate the expressions.

SHOP's syntax allows methods to define various branches (i.e. condition + tail). We will use this to define our third method:

```
(:method (prepare-weapon ?x)
         ( (not (is-weapon-aimed? ?x ?y)) )
         ( (!aim ?x ?y) )
         (is-weapon-aimed? ?x ?y) )
         () )
```

This method, more complex than the rest of the examples, expresses that it is defined to solve the task 'prepare-weapon' and that has two branches (that is why we have four blocks of expressions). Blocks are grouped in pairs, so the first block indicates preconditions, and the second is a list of actions. This means the method above will either aim the weapon (!aim) if it was not aimed or do nothing () if it was already aimed.

An operator is defined as **(:operator h D A)**, where *h* is the primitive task it defines, and *D* and *A* are deletion and additions of symbols contained in *h*. It is similar to an action in STRIPS, as its meaning is that in order to achieve the task, the atoms in *D* must be removed from the state of the world, and those in *A* must be added. Let us, then, define operators for our primitive tasks:

```
(:operator (!frighten-away ?agent ?target)
             ( (suspicious ?agent) (heard ?agent ?target) (in-sight ?agent
      ?target) )
             ( (alarm ?on) ) )
```

Just to make things clearer, the operator above expresses that it is defining the actions to execute for the primitive task named '!frighten-away'. The following block of expressions is a list of knowledge to delete from the state of the world (in this case, our agent will stop being suspicious and will no longer have any auditory or visual stimulus. The last block is a list of knowledge to add, so it will trigger the alarm.

```
(:operator (!shoot ?agent ?target)
             ( (suspicious ?agent) (heard ?agent ?target) (in-sight ?agent
      ?target) )
             () )


(:operator (!aim ?agent ?target)
             ()
             ( point-weapon-at(?agent ?target) ) )
```

With this information, our problem would be defined. The expanded task network the planner would use is shown in Figure 12.

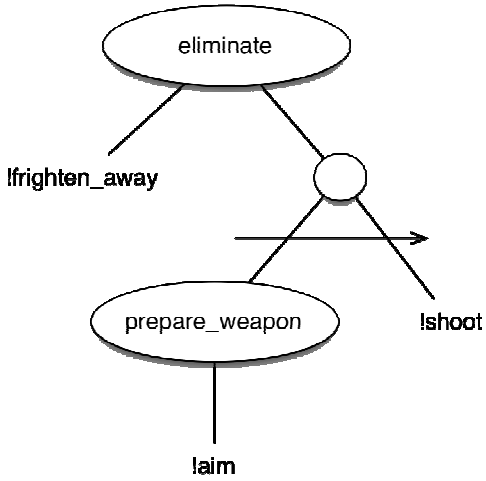


Figure 12. Task network used in our example

Depending on what the initial state of the world is, the planner would provide us with different plans. Table 2 shows some of these plans:

Initial state	Plans
((height ?threat '130) (heard ?agent ?threat))	((!frighten-away ?agent ?threat))
	((!aim ?agent ?threat) (!shoot ?agent ?threat))
((height ?threat '190) (in-sight ?agent ?threat))	((!aim ?agent ?threat) (!shoot ?agent ?threat))
((height ?threat '190) (in-sight ?agent ?threat) (is-weapon-aimed ?agent ?threat))	((!shoot ?agent ?threat))

Table 2. Plans created by SHOP for our problem

The main difference between GOAP and HTNs is that the former is focused on actions, whilst the latter are centred in tasks, which means that Task Networks work trying to choose a strategy rather than the actions needed to satisfy it (Hoang et al., 2005).

Another advantage of HTNs is that they can work with more complex conditions. GOAP requires the state of the world to be expressed as atomic pieces of knowledge (Orkin, 2004-2). This means GOAP uses a large bit array, where each situation can be present or not present. For HTNs, however, we can have complex conditions : they use an infinite set of symbols, while STRIPS-like planners have a finite plan-space (Lekavý & Návrat, 2007).

Chapter 6. REACTIVE PLANNING

Automatic planners also have some drawbacks. They only produce plans that are valid on their creation, but do not care about their execution or considering possible invalidations over time. A possible solution to this problem would be to re-run the planner every time a plan fails, or even periodically, so we get better solutions over time (Beaudry et al., 2005), but this is not very appropriate in terms of performance.

On the other hand, *reactive planners* build or change their plans in response to the shifting situations at run time (Firby, 1987; Wilkins et al., 1995). This is much better suited for the kind of problems we have to face in game development.

6.1. Belief-desire-intention

The BDI is based on two philosophical theories about human reasoning: Intentional Systems (Dennett, 1987), and Practical Reasoning (Bratman, 1987). They describe entities as rational selves with their beliefs and desires that behave following intentions, conceived as partial plans (Georgeff & Lansky, 1987). It has become a very important model for implementing practical reasoning agents (Rao & Georgeff, 1991; Rao & Georgeff, 1995; Georgeff et al., 1999; Hardland et al., 2002; Ambroszkiewicz & Komar, 2006).

In fact, we can match the concepts used in a classic planning problem to those used by BDI models, where:

- A **belief** represents information about the state of the world.
- A **desire** is a goal the AI must achieve.
- **Intentions** are the plans that have been calculated, and which the agent has committed to follow.

Apart from these components, a BDI system, or reactive planner, will also have a library of plans (Bratman et al., 1988), which describe how to satisfy desires depending on some preconditions.

When a BDI system decides to fulfil a goal, it selects a plan rule from the library, and puts it in its database of intentions. The execution of this plan can lead to new goals, or to the choosing of alternative plans to satisfy its desires, if its current plans fail at any time. This process is repeated until a plan succeeds completely or there is nothing else the agent can do (Sardina & Padgham, 2007).

The main difference with *automatic planners* is that, in this case, we are not building a full plan beforehand, but we are selecting smaller objectives or choosing alternative solutions as we execute our overall plan (intention).

Let us represent the example we have been using so far built as a BDI system (see Figure 13):

- Our list of **beliefs** represents the state of the world as we currently perceive it. For example, we know there is an enemy on sight, we have a weapon, our current health level is optimal or we have ten bullets left.
- Our internal **desire** is to not be threatened.
- Providing our desire and beliefs, our agent's **intention** is to eliminate its enemy.

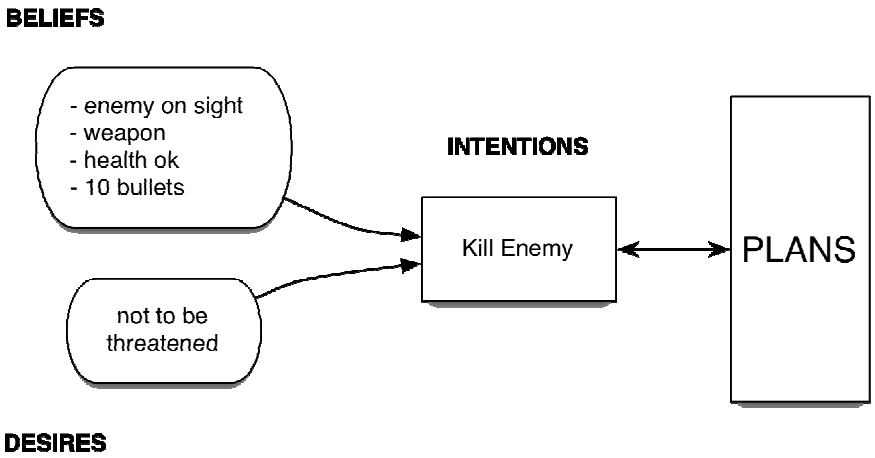


Figure 13. Simple BDI representation of our guard AI

If we start running our current *intention*, we can choose a plan from our library, such as "*fire at enemy*". We can break this up into smaller tasks; for example "*get in range*", "*aim at enemy*" and "*pull the trigger*". Let us say now we have run this plan a couple of times, our enemy is still alive, and we have run out of ammunition; this

would make us choose a new plan to "*reload*" our weapon, which we would execute, and, once it has been completed, we will keep attacking our enemy until we have killed it. So, our long-term goal is fixed, but our plan varies over time, reacting to new situations.

Although the initial framework presented is good to model basic human behaviour, it has been subject to various modifications and extensions to enhance it (Norling, 2004; Castelfranchi et al., 2006; Guerra- Hernández et al., 2004). Black & White (Lionhead, 2001) presented one of the most advanced game AIs to date, and it was based in a BDI architecture, with some extensions, such as the use of reinforcement learning (Evans, 2002).

6.2. Behaviour trees

While hierarchical, state-based techniques are good solutions, their semantics are not as powerful as they could be. Important constructions such as selections or sequences of actions produce many sub-FSMs, states and transitions. Behaviours presented in some current games imply working with very complex systems that can become hard to maintain or modify (Isla, 2008). Also, changes in design could require extensive programming work.

Behaviour trees (BT) are a means to avoid this kind of problems, providing a means to describe sophisticated behaviours through a simple hierarchical decomposition using basic building blocks (Champandard, 2008), like the one shown in Figure 14. This offers development teams several advantages:

- Transitions are not explicitly declared, but are controlled by special nodes in the tree (meta-nodes) like *sequences*, *selectors* or *parallels*. This makes these structures much clearer and understandable.

- Trees can be easily extended as needed, just implementing new blocks, and without the need to modify any other part of the code.

- Trees, or sub-trees can be easily reused, allowing developers to swiftly create variations of behaviours.

- Designers can be provided with tools to create, modify or experiment with behaviours with no programming work involved.
- Tree design is flexible enough to allow all logic to be data driven.
- Unlike FSMs and HFSMs, behaviour trees allow two states to be running concurrently.

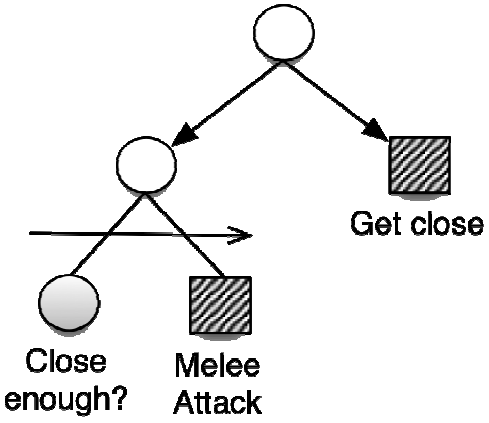


Figure 14. Simple behaviour tree describing a 'Melee attack'

BTs have become a widely used technique, and they are present in many commercial games like Halo (Bungie, 2001), Spore (Maxis, 2008) or Prototype (Radical, 2009), so their usefulness has been proved in real products.

Some games create special branches controlled by scripts (Garces & Champandard, 2009) or allow designers to modify the order of execution of their trees, letting them choose what branch must be active at any given moment.

Behaviour trees can also allow multi-level architectures to be created. If we change the granularity of the structure, we can start considering complete branches as sub-behaviours. These sub-trees can be separated from the main one and become leaf nodes: we could have different views of the behaviours at different granularity levels. This way, we could create a system where AI engineers are in charge of maintaining and generating a library of complex sub-behaviours, and designers use those to easily create new full behaviours for their AI characters.

This thesis presents a new technique that is based on behaviour trees, so we study them in more detail in Chapter 9.

6.2.1. Behaviour selection and interruption handling

These structures can be used to model and select behaviours, so effectively, we have a behaviour tree where its branches are trees as well. The root of our tree is usually a selector that chooses the best sub-behaviour among its children. The problem with this architecture is related to behaviour interruption and resuming, as, plain behaviour trees do not support it.

Some solutions have been presented to solve this problem, among them we find MARPO (Laming, 2009). This technique, used in GTA: Chinatown Wars, uses a three-stack approach, where the stacks hold information about "*long term*", "*reactive*" and "*immediate*" behaviours; only one stack is executed at each moment, but, if the behaviour is interrupted by a higher-priority one, the stack is unloaded (just keeping a reference to the root of the behaviour that was being run) and replaced by the new behaviour. After the new tree has been run, the previous, lower-level behaviour is restarted.

A similar, although more sophisticated, approach is used by Behaviour Multi-queues (Cutumisu & Szafron, 2009), where three sets of queues are used. In this case, behaviours are marked at initialisation time as "*independent*" or "*collaborative*", which indicate whether more than one agent is required to perform the action. They are also classified as "*proactive*", which are started by an NPC spontaneously, when the agent has no active behaviours; "*latent*", which are run when an even cue is fired; "*reactive*", which are run when the agent discovers another NPC he can collaborate with.

When a behaviour is created, it is placed on the appropriate queue; there is one queue for proactive independent behaviours, and an arbitrary number of them for collaborative (proactive or reactive) and latent (independent, collaborative or reactive) behaviours. Different types of behaviours have different priorities (which are known by the queues), so some can interrupt others. The main difference with MARPO is that while the approach used in that case forced to restart our previous behaviour, once we decided to resume it, behaviour multi-queues can continue it in the point where it was left at.

As we will show later, we have opted for a behaviour restart approach in our solution (see 11.1.1).

Hinted-execution Behaviour Trees

In this section, we will study our novel solution to in detail, from its design to its final implementation. We will also present how it can be applied to real games, and what benefits it offers.

Chapter 7. DEFINING THE PROBLEM

In previous chapters we studied of the different systems used by videogames to model and control their AI behaviours. Each of the techniques presented so far have different characteristics, and thus some are better at solving some problems, but worse at others. We classified these algorithms and structures into two big groups, plus a third one which represented mixed solutions.

Recalling what the goal we set in the first chapters of this thesis was, we are trying to achieve is a system that, in short:

- Produces dynamic, but controllable (and fun), behaviours, which will allow us to tailor the experience to different classes of players.
- Can make prototyping and team collaboration easier.

In this chapter we will analyse the information gathered during our research and present a new technique that allows us to achieve our objectives.

7.1. Comparing behaviour modelling techniques

The main question we must answer is: can the solutions presented so far provide us with enough flexibility to fulfil our goals?

In order to do so, we have chosen a range of characteristics where a comparison is useful for our objective of building complex autonomous, yet controllable behaviours. The different aspects studied are:

- **Simplicity**, or how easy to implement the system is.
- The **degree of control** designers have over the system, with no need for programming work.
- How well the system **scales** when the problem grows.
- Whether or not aspects of the system will be **re-usable**.
- The **unpredictability** of the behaviours modelled by the system.

- How easily behaviours can be **extended** or modified to generate new behaviours or adapt existing ones to different situations.

Each characteristic is scored with a value in the range [0…5], and an overall (total) result is also provided. We have filled in the scores using our own expert judgement, and drawn up Table 3.

Technique	Simplicity (programming)	Modifiable by designers? (no coding involved)	Scalability	Re-usability	Behaviour unpredictability	Behaviour extensibility	Overall result (total)
FSM	5	1	0	0	0	0	6
Stack-based FSM	4	1	2	0	0	0	7
HFSM	3	2	2	2	2	1	12
HCSM	2	3	2	2	4	1	14
Petri nets	4	2	1	0	0	0	7
GOAP	2	3	3	3	4	2	17
HTN	1	2	4	4	4	2	17
BDI	2	2	3	3	4	3	17
BT	3	4	3	4	3	3	20

Table 3. Comparison of the different behaviour-modelling techniques studied

From the results we can extrapolate the following conclusions:

- **FSMs** are the easiest architectures to implement, but they do not scale well, are difficult to modify by designers, and offer no re-usability. The complexity of their notation is very low, so they are easy to understand, even for non-technical members of the team, and while there is a small chance of producing bad emergent behaviours, this implies that they are usually very predictable. They are not very powerful, because, although they can solve almost any problem, creating complex behaviours, or extending existing ones, would require a lot of modelling work.

- **Stack-based FSMs** are quite similar to FSMs, but slightly more difficult. They are more powerful, as they allow for more complex solutions to be created more easily. As with regular FSMs, they produce very predictable behaviours and extending existing ones is not a trivial task.

- **HFSMs** are the best option when it comes to using an FSM-based solution. They have medium-complexity; their behaviours are a little less predictable and scale better. They also allow some re-usability and extending existing behaviours is easier, as we do not have to add as many transitions (most of them could be hidden inside a super-state). Mainly, their results in the study indicate they are a good technology to use.

- **HCSMs** are harder to implement. They are more modifiable by designers, as they could set different values in their control panels. They are similar to HFSMs, but their behaviours are more unpredictable, as they use activity functions to generate output, which have results that can be more difficult to predict. HCSMs are normally used as directors, to help and orchestrate complex situations.

- **Petri nets** are a good way to represent concurrent behaviours. They are not very hard to implement, as long as we just use their simplest version, but they suffer from the same problems as FSMs, so they get a very similar overall score.

- **GOAP** and **HTNs** are a big step in terms of unpredictability, although they can also lead to undesirable behaviours. Their complexity is greater, but they scale quite well. They are both very powerful, but they require extensive work to deal with dynamic changes (they need to re-plan when their solutions are no longer valid). Designers could modify behaviours through high-level interfaces that allow for things such as choosing what actions are available for a certain type of NPC, but any important change would require programming work.

- **Belief-Desire-Intention** architectures can be hard to program. They can receive some input from designers, but they produce pretty unpredictable

behaviours. They could be compared to automatic planners, but deal with dynamic changes better.

- **BTs** are not very hard to implement. They are easy to understand, and designers could even create new behaviours, if they are provided with some tools. They deal with dynamic events very well, if they are correctly implemented. They scale well, and allow for a high degree of re-usability, although their solutions are predictable unless a lot of time is spent creating their structures. Extending existing behaviours is easier in this case, as we would only have to add or remove nodes or branches in our tree, which can be done using a tool; however, this could introduce new bugs, increasing the risks, as we could be adding new unpredicted situations, or making ones that worked correctly before stop doing so.

In general, BT's are the best technology for our purpose: building autonomous, yet controllable behaviours. However, there are still some points where they can be enhanced. For example, automated planners produce more unpredictable solutions with less tuning work. Also, although designers can create and modify BT's at will, this is not always the best approach. Wrongly designed trees can lead to strange situations, such as our guard fleeing even though it has full health and a very powerful weapon, so testing new ideas is not always easy.

7.2. Objective

From the conclusions outlined in the previous point, we can say Behaviour Trees seem to be closest to what we are looking for. They are also quite understandable (provided we have the right tools to visualise them), and extendable, as new behaviours can be created from other ones or expanded adding new building blocks.

Although they look like a good solution, the biggest problem with Behaviour Trees is that they can become too bloated to be controlled by non-technical staff. Modifying a BT without the appropriate knowledge can cause odd bugs and situations. What we would normally want is to have an engineer who can look after the behaviours, and improve or change them based on the requests that come

from the design team. This, however, adds some problems to the development process. Figure 15 shows how the information flows in this scenario.

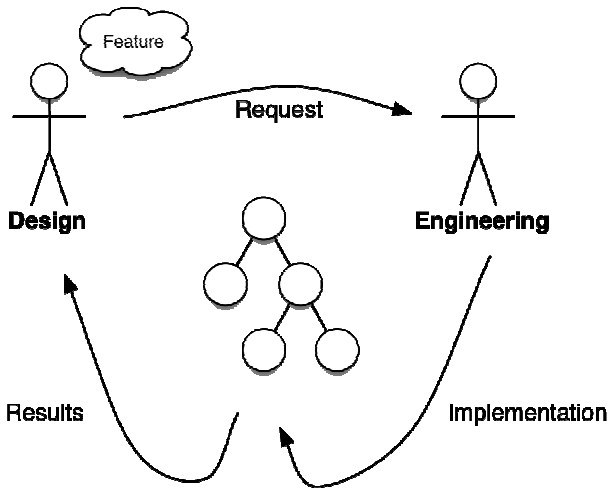


Figure 15. Development process with regular Behaviour Trees

In this case, every new feature designers would like to have, or try, must be requested from an engineer. That is where the first problem is: designers do not have direct control over the system, but they delegate the work to engineers. They will then implement the changes, put them in the game, and, after some time, they will be able to check the results. Finally, whether the feature should be kept, needs polishing or should be discarded will be decided. Let us say the feature was correct, but it needed some changes. In that situation, a new request would be created, and the cycle would start over and over again. There is a clear waste of time in our workflow.

Allowing prototyping is the key here. A good system should let designers modify behaviours directly, as shown in Figure 16, but in a safe way. That is, we do not really want correct and fully tested trees to be modified directly, as it is too risky; instead, we want a way to alter the execution of those trees, in this way we can be adding new, evolved behaviours.

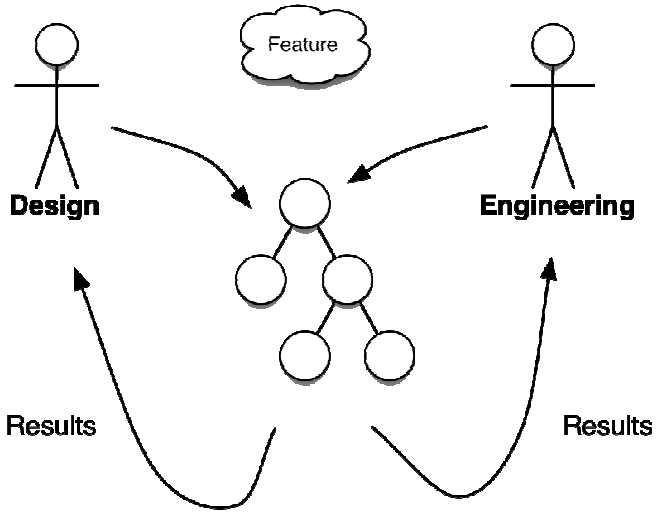


Figure 16. A good system should be able to let both designers and engineers modify and prototype behaviours

7.3. How are BTs used in real games?

Behaviour trees can be used in various ways. AI programming, like any other engineering work, is an art, and so different engineers can come up with different solutions, even when they are using the same tools. Some games use BTs just as decision makers, whereas others, such as *Driver: San Francisco* (Ubisoft, 2011), also use them as a tool to describe behaviours.

This thesis is focused on producing a solution that allows new behaviours to be easily developed and tested, without breaking the existing system. Before we can define a way to do that, we must analyse how a BT works in a real game, and what a modification might suppose.

To illustrate the process, we will expand the example we have been using so far: our guard's AI. Just as a reminder, this type of entity is, basically, an agent that can patrol while it is idle, but it can also inspect things when it is suspicious. If it detects an enemy it will attack it, but it will also try to take cover if it is in danger.

Let us say our project is just a few months away from being released; however, at this point, some tests have shown that the game is too simple and not great fun, so the design team comes up with a new idea: they want players to be able to take on

the aspect of the enemies they eliminate, so other foes will not recognise them as threats straight away.

A game based on BTs will require us to modify the trees to add this new logic. Even if it seems like a pretty straightforward change, this can end up creating new bugs, and making things stop working as they used to do: our original behaviour tree can be quite complex (depending on its size), and choosing the right places to add the new logic could not be so trivial anymore. Also, normally this will require an engineer to be in charge of the changes; they will even probably be the ones to implement the actual modifications.

On top of that, we must bear in mind that after adding the new behaviour, designers might decide the idea is not good enough, and that it does not work (in terms of gameplay values), so we could, potentially, be wasting many hours of engineering, which is quite inconvenient for the project itself. So, how do we enhance the process?

7.4. Improving prototyping

Looking at our problem, we can notice the solution comes by leaving our trees intact, as we know they have been tested and tweaked thoroughly, and modifications are dangerous. But, not modifying our trees means the behaviour will not change… unless we can re-arrange the way the tree is executed.

7.4.1. Personalities

A simple way of changing the normal flow of execution on a tree is the inclusion of a set of parameters (Ellinger, 2008) that can be modified, so selectors and/or conditions in our tree behave differently. We will call them 'personality traits' as we would be able to generate different *personalities* just by having different sets of values for them. A good example of this could be an *aggressiveness* trait, such as the one presented previously in 5.1. In this case, it could just be a boolean value, where 'true' means the AI must be aggressive towards enemies, while 'false' indicates the agent is just coward, and it must try to hide at all cost. Figure 17 shows an example of this.

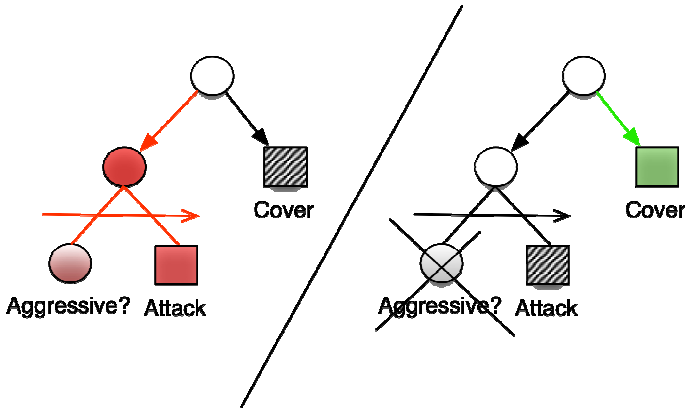


Figure 17. The usage of personality traits can help us modify the normal execution of a tree

This is the kind of structure we would be looking for, if we were allowed to modify our original tree. It just does not provide us with enough power in our case, though, due to the restriction we have imposed at the beginning of this section, that is, the invariability of the trees. So, using traits or values **we cannot add new logic**.

7.4.2. Hints

Let us think differently. How can we alter the results without changing the structure or a tree? The answer is, by means of messages. We will call these messages *hints*.

As introduced in 6.2, a behaviour tree is just the organised sum of several building blocks, one of which are *selectors*. A simple selector will execute its branches **in order**, until one of them succeeds (or all of them fail). So, priority, and thus, execution flow, comes from this order.

Back to our guard's AI, we can simplify its behaviour, so a big selector basically forms it, choosing among different sub-behaviours, which are, as we have just said, prioritised. Figure 18 illustrates this.

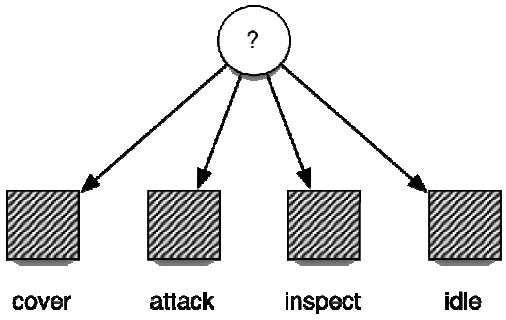


Figure 18. A simplified vision of our guard's AI

In the figure, we can see our AI will first check if it has to cover, then it will try to attack, but if it does not have a target, it will try to look for one. If no threat is visible, then it can just be idle.

Selectors are really the key to modify the behaviour of an AI, as they are the 'intelligent bits' inside a BT. Just like in real life, making a decision requires inspecting different options and checking the pros and the cons of them… but just as in the real world, a decision can also be affected by other people's opinions or thoughts, which may bias our verdict towards a particular choice.

That is exactly what a hint represents: a way to let the selector know what a higher-level entity thinks about what it should do. But, as in real life, hints are not always *positive*, that is, we are not always told what to do, but what **not** to do.

A simple example could be recreating our 'aggressiveness' trait without modifying our tree. Let us say we did not think about the possibility of having two AIs that only differ in this parameter, and we want to test if this provides players with a greater variety of situations. We want to create a kamikaze AI, which never tries to hide; to do so we are going to pass our selector a 'DO NOT COVER' hint. This is letting the selector know it should really avoid covering and it should attack its target, rather than being a coward.

However we still want our AI to be able to respond properly to other events, that is, if we want it to keep being autonomous: we only need to be able to affect its priorities. Effectively, this means we are reordering its branches, as shown in Figure 19.

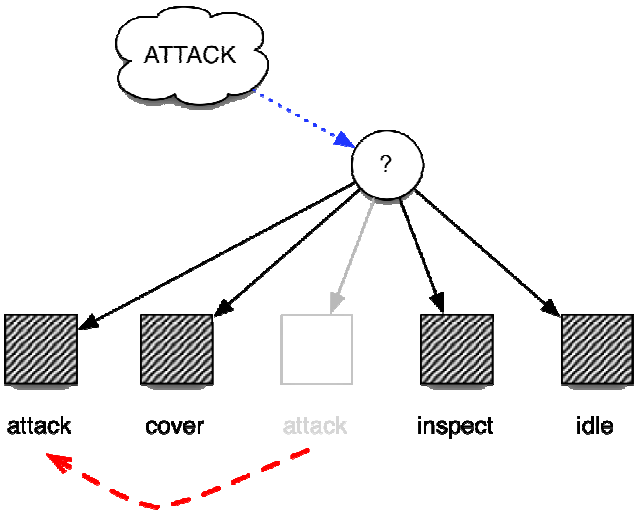


Figure 19. When a selector receives a hint, it reorders its branches

The effects of these new priorities are that, no matter how low its health is, covering will be our AI's last resource.

As we said before, we still want our AI to be independent, and this is a very good example to illustrate that. If, for any reason, our AI cannot attack (e.g. it has run out of ammunition), the logical choice will be to cover: we have transformed our coward AI, whose first option was to flee, into a guard that will always try to eradicate its enemies, no matter how tough they look.

It is important to note that we have not really touched the main tree at all, and that going back to the original behaviour is as easy as taking the hint away. This means this solution satisfies our first condition.

7.4.3. Adding new logic

To some extent, this example is adding new logic, although it is not anything we could not have added with a simpler system, such as a set or personality traits. This kind of systems does not work when the new logic is more complex, such as adding the disguise system.

Now, if we remember the way a BT is built, we can get to a solution to this. Why do we just not create a simplified tree where the leaf nodes send hints to our main tree? Just by implement this new building block, and replacing the action nodes in our main tree with these new **hint nodes**, we could overcome this limitation.

So, continuing with the example, let us define the new feature. Basically, what we want to do is prevent AIs from attacking us, in case we are disguised; also, we want AIs to start attacking us if we have been near them for too long. Analysing the problem with our hint system in mind, we can see that what we want is to build a tree with two branches: "do not attack" and "we have an intruder, just attack".

In order to describe what we want our AI to do first, which is to avoid attacking at all cost, we again use a *negative hint*: that is "do NOT attack". On the other hand, if what we want is just to get back to the normal behaviour, we have to stop sending hints, which we have represented as 'CLEAR HINTS' in Figure 20. As a second option, we could also want to make our AI more aggressive sending a *positive* hint: "attack".

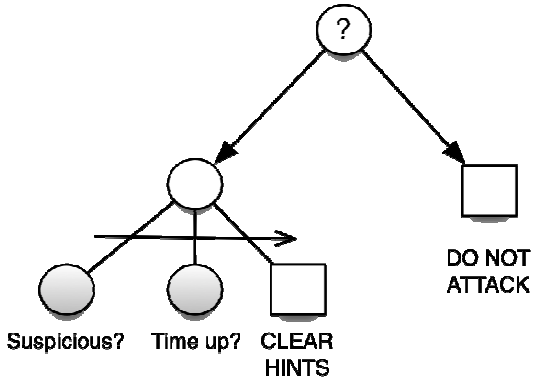


Figure 20. A simple tree can be used to modify a main behaviour

Providing the AI with a higher-level tree like this, we will make the main selector reorder its branches as shown in Figure 21, in case the AI has not discovered the deception. In that case, attacking will be the AI's last option. The tree will go back to normal as soon as the conditions in the first branch are met.

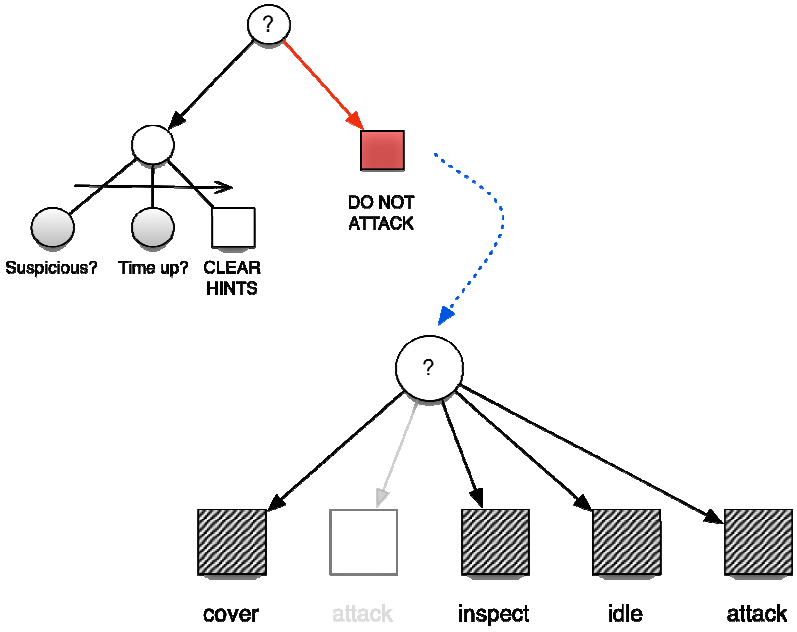


Figure 21. If the AI is not suspicious, or not enough time has passed, then we reorder the selector so attack is the last available option

The key to these simplified trees is that they are very easy to understand and, as we are not changing the original behaviour, we can build or modify them very quickly. In fact, just with a little training, any member of the team can create their own trees and start experimenting with new ideas, which is the core work of this thesis.

So far we have just presented the basic idea behind Hinted-execution Behaviour Trees (HeBTs). A fully functional prototype has been developed as a fundamental part of this thesis and, in the following chapters, we will study, based on the experience gained during its implementation, how an HeBT system works, and how this technique provides game studios with a good solution, that will allow them to produce fun AIs maximising the use of resources.

Chapter 8. OVERVIEW OF THE SYSTEM

In the previous chapter, we presented our novel solution, Hinted-Execution Behaviour Trees, to tackle problems derived from the complexity and dynamic nature of videogames.

Game programming is an agile process, and needs practical solutions, rather than theoretical ones, so we decided, in the earliest stages of this research, that we had to take a pragmatic approach, and base our study on experimenting with a real game AI.

In this chapter, we will present an overview of the prototype developed as part of this work, from a high-level point of view, showing how the different components of the system work together to satisfy the goals of this thesis.

8.1. Components

Our system is made up of three main parts or components: an HeBT library, a tree editor, and a game prototype, built on top of Half-Life 2's public SDK. We will describe them roughly in this section, and explain each of them in detail in the following chapters.

8.1.1. Hinted-execution Behaviour Tree library

One of the objectives presented in Chapter 2 was that our system has to be abstract enough to support any kind of game. Because of this, the library is highly templatised, written in C++, and it contains all the elements that can be common to different games; this way, we can just re-use the library, implementing only game-specific objects, such as action nodes.

We use a Lua interpreter to import and build trees easily, and we have also added a communication layer, based on sockets, that can be used to control our AIs from the tree editor, as we will see later in this thesis.

8.1.2. HeBT Editor

Another important goal of this work is to generate a solution that is easy to use. Without such an editor, it would not be possible for non-technical staff to generate new behaviours, and the technique itself would lose most of its power. Because of this, an editor has been created in parallel to the HeBT library.

The tool has been implemented using C# and WPF. It can be personalised using different configuration files, based on XML, so it can be reused in different games. These configuration files allow us to generate libraries of conditions and actions to be used by our behaviours.

Also, the editor is capable of communicating directly with the game, so behaviours can be changed on-the-fly. This eases the work, as changes can be tested just by clicking a button.

Finally, the editor is also capable of showing, in real time, how the tree is working, so it can be debugged effortlessly. To do so, the editor will show the whole tree and highlight the nodes that are currently running. This is especially important when we start using hints, as it allows us to see how our higher-level logic is affecting the execution of the original tree.

8.1.3. Game prototype

In order to get some valid results, we decided it was important to apply our new technique to a real game. Building a game from scratch is getting more and more complicated, even more when we are talking about 3D environments, and it would have required quite a long time. Instead, we opted to implement a modification for a commercial game. In our case, we chose Half-Life 2, as all its AI is written in C++, and it was relatively easy to replace their systems with our own.

As the system has been designed to be game-independent, we have only had to create some functionality that is specific to Half-Life, and the kind of game we were trying to create. Specifically, we have implemented some actions, conditions, and some communication commands, so we are able to instantiate these new objects.

8.2. High-level architecture

Although we have subdivided the system into three components, we can differentiate between two main parts: the game side (library and game prototype) and the tools side (editor). Figure 22 shows a high-level idea of the structure of the system.

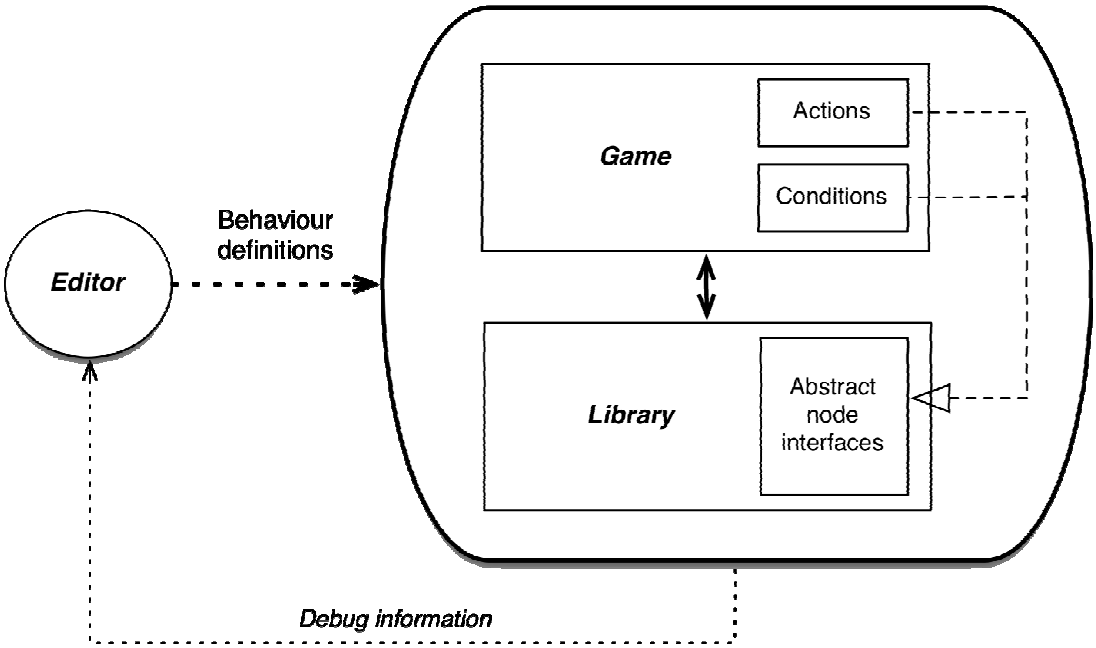


Figure 22. High-level architecture of the system

Our system is data-driven; this means, the data generated by the editor is used by the game to produce the final behaviours. The library holds most of the logic, and exposes some interfaces so any game that uses it can implement specific logic that cannot be shared. These specific implementations are held inside the game. The game side can also send debug information to the editor if it is required.

We will study each of these components, and their relations, in the following chapters.

Chapter 9. BUILDING A BEHAVIOUR TREE SYSTEM

Hinted-execution Behaviour Trees are an extension of the regular BTs. With this in mind, the first logical step was building a behaviour tree system we could then upgrade. Our BT library has been built trying to keep every component as abstract as possible, as one of the objectives of this thesis is to generate a game-independent solution.

In this chapter, we will study this part of the library in detail: what main components it offers, what type of nodes are available, etc.

9.1. Manager and instances

When building an AI library, the first thing that has to be defined is how each AI instance will be represented, and how we are going to manage those AIs. In the end we do nothing but control these entities, so we must be able to keep track of them and provide some functionality to set or change their behaviours. All of this is centralised in two objects, shown in Figure 23: *AIManager* and *AIInstance*.

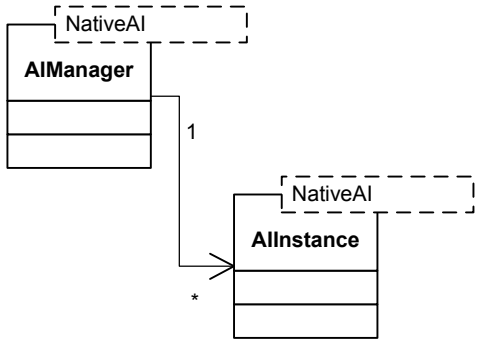


Figure 23. We represent each agent as an AIInstance, being those controlled by the AIManager

Our *AIInstances* work, to some extent, as wrappers of the native AI type of the final game[2]. This means we could potentially apply our system to any existing application.

[2] For the sake of simplicity, we will avoid marking classes as templates in the rest of the figures provided.

9.1.1. AI Manager

The manager acts as a hub for the rest of the library. It contains other smaller managers –Lua and communications– and provides us with basic functionality to create, delete and get AI instances. Figure 24 shows how the manager is connected to other components of the system.

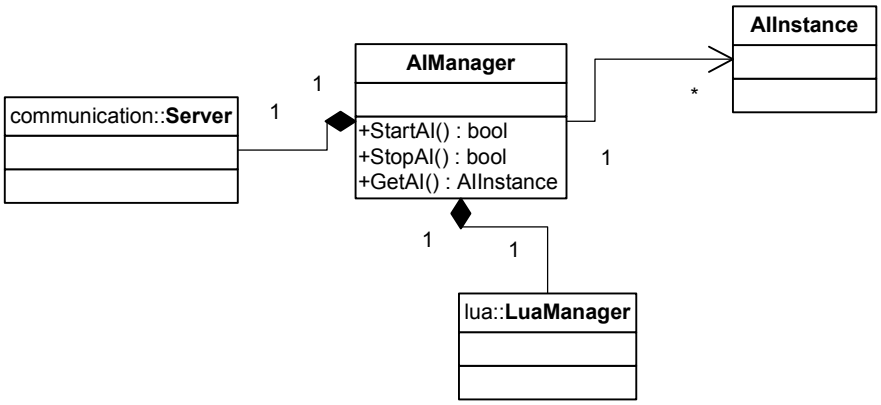


Figure 24. Relations of the manager with the rest of the library

9.1.2. Instance

As we have already said, each AI instance represents an agent in the world, and because of the design we have used, it acts as a wrapper of a native entity. Instances keep a pointer to this game entity, as well as a pointer to the behaviour tree that is controlling it. Finally, it contains a blackboard, which we have built as a list of pairs key/value, and that acts as a simple memory, allowing the AI to store and access small pieces of data that can be useful for their behaviours. Figure 25 shows how an instance fits in the library.

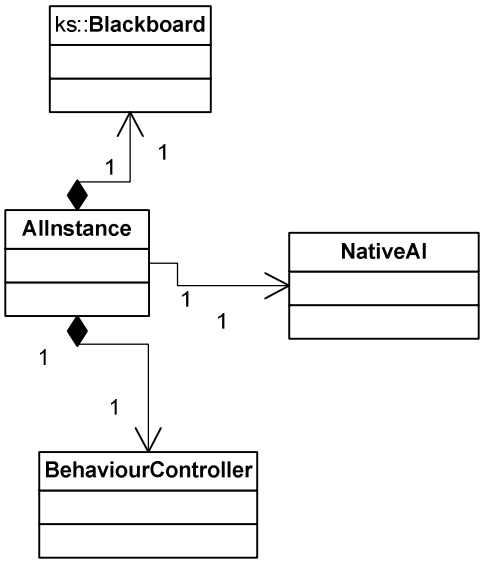


Figure 25. Connections of an AIInstance to the rest of the library

As we can see in the figure, we do not really reference a behaviour tree directly. Instead, we use a *BehaviourController*, which is one of the keys of the hint system, and will be studied in depth in 11.1, but, for now, we can say it contains a list of trees that are being executed simultaneously to build the final behaviour.

Instances are our interface with our agents, and thus they provide methods to set their behaviours. Now, we must study how these behaviours are created.

9.2. Defining a tree

A behaviour tree is just a structure where each node can have *0..n* children. Execution of these trees begins at their root at every step. This root node will then run its logic and execute one, or several, of its children, and so on.

Because of the latent execution present in BTs, we can say each node has two possible states: *STOPPED* and *RUNNING*. We have added two additional methods to our nodes, so we can execute some logic that is only run when it is started or stopped.

A node will be stepped only if it is running. When a node is run, it must return its state after being updated. A node can *SUCCEED*, *FAIL*, report it is still *IN PROGRESS* or abort its execution because of a serious *ERROR*.

From these initial requirements, a *Composite* pattern (Gamma et al., 1994) seems appropriate. This way, every node will have a common interface. We will also implement a Visitor pattern, as it will allow us to process our tree in different ways without the need of modifying the structure itself.

Several nodes inherit from this abstract node. Some of them are strict composite nodes, such as *selectors*, *sequences* and *parallels*, others are leaf nodes (i.e. nodes with no children), and finally we have decorators or *filters*, which are nodes that following a *Decorator* pattern, modify the standard behaviour of a node. Figure 26 shows the basic structure of this part of the library.

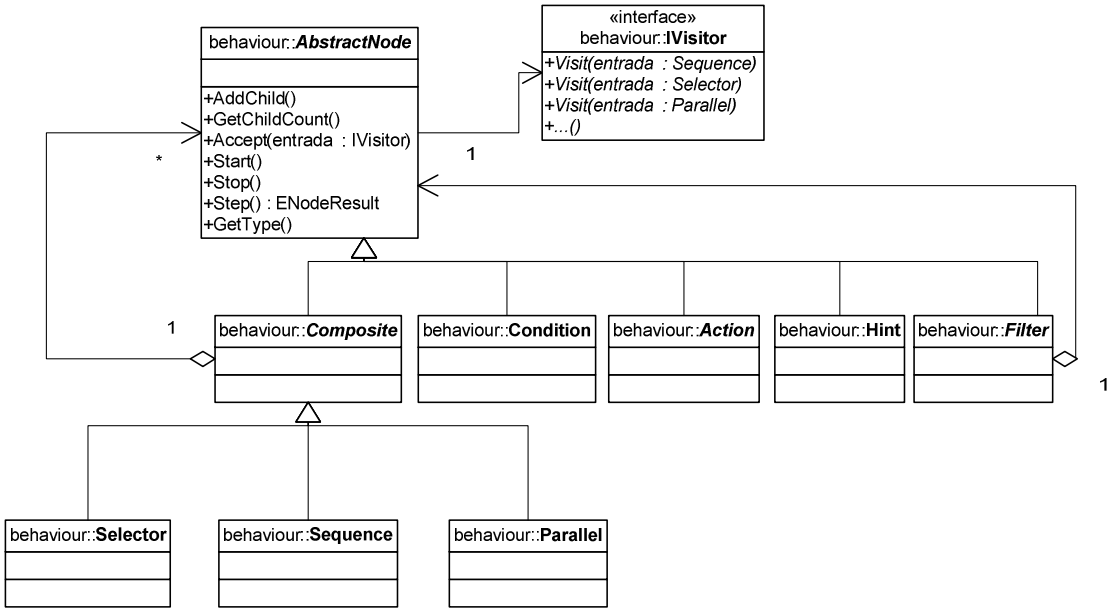


Figure 26. A view of the main classes in charge of defining a behaviour tree

We will detail how each node works in the following sections.

9.3. Composite nodes

Composite nodes are those that have *1..n* children. Note the range starts at 1, indicating these nodes cannot be used as leafs in the tree. They control the execution flow in the tree, and we will call them *metanodes*, as they basically allow us to create more complex structures.

9.3.1. Sequence

Sequences are the simplest of the composite nodes. As their name indicates, they allow sequential execution to be added to a tree. They will execute their child branches in order, bailing out prematurely if one of the tasks fails. Translating this to actual code, sequences look like this:

```
1  template< class AIInstance >
2  ENodeResult Sequence< AIInstance >::Step()
3  {
4         ENodeResult eResult = NR_IN_PROGRESS;
5
6         ASSERT_STR( m_currentSubtask != m_children.end(),
7                     L"Trying to update an invalid subtask" );
8
9         eResult = ( *m_currentSubtask )->Step();
10        if ( eResult == NR_SUCCEEDED )
11        {
12               // Stop current task
13               ( *m_currentSubtask )->Stop();
14
15               // Jump to next task
16               ++m_currentSubtask;
17               if ( m_currentSubtask != m_children.end() )
18               {
19                      // Start new task and return that we're in progress
20                      if ( ( *m_currentSubtask )->Start() )
21                      {
22                             eResult = NR_IN_PROGRESS;
23                      }
24                      else
25                      {
26                             eResult = NR_ERROR;
27                      }
28               }
29               else
30               {
31                      // Succeeded
32                      m_currentSubtask = m_children.end();
33                      eResult = NR_SUCCEEDED;
34               }
35        }
36
37        return eResult;
38 }
```

Let us analyse this snippet. We have already said composite nodes cannot work as leaf nodes, so the first thing a sequence is checking (in this case, it is asserting it in lines 6-7) is that we actually have some children. We keep an iterator to the current subtask, and we initialise it when we start the node, so, in case we are running the node, the iterator should be valid.

After this first check, we are updating the current branch (line 9). The result is stored in *eResult*, so in case the child node fails (or is erroneous), the sequence will return that failure to its parent and bail out. Otherwise we stop the task that has

just finished (line 13) and increment the iterator (line 16). If we have not processed all our children, we will start the new one (line 20). On the other hand, if we have already executed all of them, it means the sequence has succeeded (lines 32-33), and the sequence will end.

Let us examine a real-world example (Figure 27).

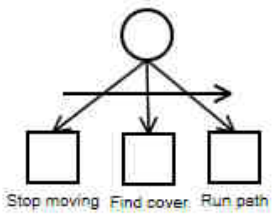


Figure 27. An example sequence

In this case, we are trying to model a simple "cover" behaviour, in which we want our AI to stop moving, then find a cover and, finally, run to the cover it has just found. Table 4 shows the results of the execution of this behaviour, depending on the results returned by each of the branches of the tree.

Stop moving	Find cover	Run path	Sequence result
Succeeded	Succeeded	Succeeded	Succeeded
Succeeded	Succeeded	Failed	Failed
Succeeded	Failed	-	Failed
Failed	-	-	Failed

Table 4. Possible results of the example sequence

9.3.2. Parallel

Parallels allow the concurrent execution of their children. A parallel node will update all its children in each step, and it will only bail out if any of them fail or succeed.

Although concurrency could be real (i.e. each child would be run on a different thread), we have opted for a simpler solution. In this case, the execution will be similar to that of a sequence node, but all the branches will be run in the same step.

1

```
1  template< class AIInstance >
2  ENodeResult Parallel< AIInstance >::Step()
3  {
4        ASSERT_STR( !m_children.empty(),
5                    L"Trying to update an invalid subtask" );
6
7        // Update the nodes. Stop as soon as one ends
8        NodeVector::iterator it = m_children.begin();
9        NodeVector::iterator end = m_children.end();
10       for ( ; it != end; ++it )
11       {
12             AbstractNode< AIInstance >* pNode = *it;
13             ENodeResult eResult = pNode->Step();
14             if ( eResult != NR_IN_PROGRESS )
15             {
16                   return eResult;
17             }
18       }
19
20       return NR_IN_PROGRESS;
21 }
```

Examining this code, we can tell we are iterating through all the child nodes (line 10) and we bail out as soon as one of them is not in progress anymore (lines 14, 16).

We can use parallels to create assertions, that is, conditions that we want to check on every step and that are mandatory for a sub-tree to be executed (we will talk about them in 9.4.1.2). Following the example presented to explain how sequences work, we can extend it. Let us say that if we want to cover, we need to have an enemy we want to hide from. Figure 28 shows how this would work.

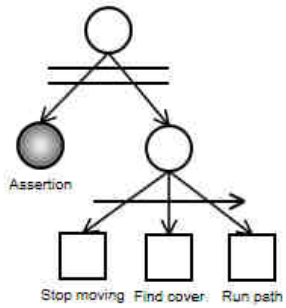


Figure 28. Example of a parallel node

We illustrate how this parallel works in Table 5.

Has enemy?	Right branch	Parallel result
TRUE	In progress	In progress
TRUE	Failed	Failed
TRUE	Succeeded	Succeeded
FALSE	-	Succeeded

Table 5. Execution of our sample parallel node

9.3.3. Selector

Selectors are the most powerful metanodes. They add a way for AIs to decide what to do, based on the state of the world.

Different types of selectors can be built. The easiest of them relies on the order of its sub-branches: it will start choosing the leftmost one, so effectively the priorities decrease as we move to the right. These are also called 'static selectors', as designers have to keep this in mind when they define the structure of the tree, and decide what the best configuration is, because the node will always execute subtasks in the given order (Champandard, 2009-1; Champandard, 2009-2; Champandard, 2009-3); it is also the one we have implemented in our library, as we show below:

```
template< class AIInstance >
ENodeResult Selector< AIInstance >::Step()
{
	ENodeResult eResult = NR_ERROR;

	ASSERT_STR( m_currentSubtask != m_children.end(),
			L"Trying to update an invalid subtask" );

	eResult = ( *m_currentSubtask )->Step();
	if ( eResult == NR_SUCCEEDED )
	{
		// Succeeded
		m_currentSubtask = m_children.end();
		eResult = NR_SUCCEEDED;
	}
	else if ( eResult == NR_FAILED )
	{
		// Stop current task
		( *m_currentSubtask )->Stop();

		// Jump to next task
		++m_currentSubtask;
		if ( m_currentSubtask != m_children.end() )
		{
			// Start new task and return that we're in progress
			if ( ( *m_currentSubtask )->Start() )
			{
```

```
28                              eResult = NR_IN_PROGRESS;
29                      }
30                      else
31                      {
32                              eResult = NR_ERROR;
33                      }
34              }
35              else
36              {
37                      // Failed, as no further task was found
38                      m_currentSubtask = m_children.end();
39                      eResult = NR_FAILED;
40              }
41      }
42
43      return eResult;
44  }
```

The way our selector works is pretty simple. We first update the current child (line 9) and, if it succeeds, then the selector bails out and succeeds. Otherwise, we will move to the next child (line 22), start it (line 26) and return we are still in progress, if everything goes fine (line 28). When we reach the end of our list of children, our selector fails.

A real-world example of a selector is shown in Figure 29. In this case, we want our AI to reload its weapon if it is necessary. The left branch executes a condition to check this. If the condition fails, then the selector will choose to attack.

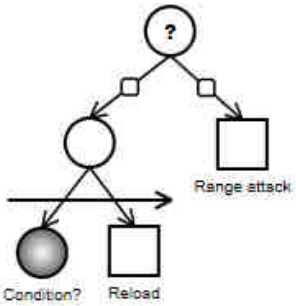


Figure 29. A simple selection allows the AI to choose whether it has to reload its weapon

Again, Table 6 shows how this selector would work.

Left branch	Right branch	Selector result
Succeeded	-	Succeeded
Failed	Succeeded	Succeeded
Failed	Failed	Failed

Table 6. Possible results of our selector example

The node presented in this section is just a first, simple version of a selector. Our Hinted-execution Behaviour Trees work thanks to an advanced selector, that is able to reorder its branches depending on the external hints it receives. It will be shown in detail in Chapter 10.

9.4. Leaves

Once we have defined our metanodes, we need to create some leaves for our tree. These atomic nodes are the ones that actually perform actions by themselves, rather than just be there to provide structure to the trees.

We have only defined two types of leaf nodes in the library: *conditions* and *actions*. A third type, *hints*, will be studied in Chapter 10.

9.4.1. Conditions

A condition is a piece of logic that checks the state of the world, returning a boolean result. Most of the decision making in BTs come from them.

It is easy to see that, because conditions are checking the environment of our agents, they are closely tied to the game itself, so we have only defined the abstract logic that makes conditions work in our library.

To make conditions more flexible, we have opted to use *condition trees* instead of plain, normal ones. This allows us to reuse our conditions in different situations, and treat their values in different ways. The editor is capable of building these trees visually, just as it does with our HeBTs, as we will see in Chapter 13.

9.4.1.1. Condition trees

The structure of our condition trees is similar to that of our BTs. They are also built using a composite pattern, so we have a basic interface for nodes, which can be atomic or complex. Figure 30 shows the classes that are part of condition trees.

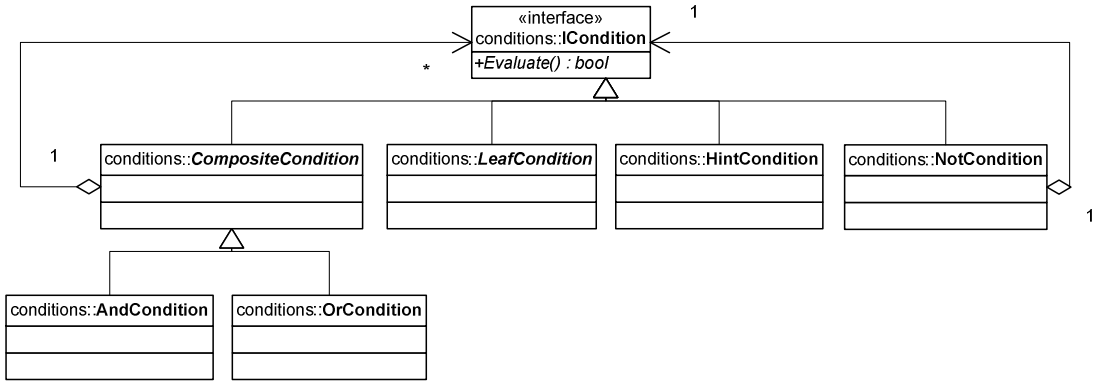


Figure 30. Design of a condition tree

"And" and "Or" conditions are simply a list of conditions which results are logically combined.

```
bool AndCondition::Evaluate() const
{
	ASSERT_STR( !m_children.empty(), L"Evaluating an empty AND condition" );
	bool bResult = true;

	ConditionVector::const_iterator it = m_children.begin();
	ConditionVector::const_iterator end = m_children.end();
	for ( ; it != end; ++it )
	{
		const ICondition* pCondition = *it;
		bResult &= pCondition->Evaluate();
		if ( !bResult )
		{
			//shortcut
			break;
		}
	}

	return bResult;
}

bool OrCondition::Evaluate() const
{
	ASSERT_STR( !m_children.empty(), L"Evaluating an empty OR condition" );
	bool bResult = false;

	ConditionVector::const_iterator it = m_children.begin();
	ConditionVector::const_iterator end = m_children.end();
	for ( ; it != end; ++it )
	{
		const ICondition* pCondition = *it;
		bResult |= pCondition->Evaluate();
		if ( bResult )
		{
			//shortcut
			break;
		}
	}

	return bResult;
}
```

"Not" conditions work is a pretty similar way, just negating the result of their underlying condition.

```
bool NotCondition::Evaluate() const
{
	ASSERT_STR( m_pDecoratedCondition != NULL, L"Evaluating a NULL condition (NOT)" );
	return !m_pDecoratedCondition->Evaluate();
}
```

On the other hand, "Leaf" conditions are those that are atomic and do not base their result in those of others. Our library only provides an abstract class as a base point for actual conditions to be defined in the game code. We will see some of these in Chapter 12.

Finally, "Hint" conditions are used by our novel extension of BTs, and will be studied in Chapter 10.

Figure 31 shows an example of a pretty complex condition, which can be read as: "We have an enemy and low ammunition, or our health is low".

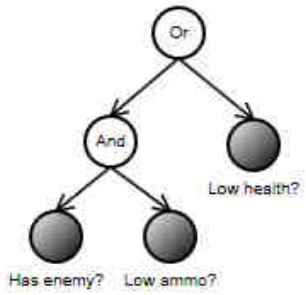


Figure 31. An example of a complex condition

9.4.1.2. Using conditions in a behaviour tree

Now that we have defined our condition trees, we have to find a way to use them in behaviour trees. BTs are just a collection of nodes, so we have to create a new node to hold these conditions. From our experience, we have noted that we need to evaluate conditions in two different ways:

- Instantaneous checks, so we can decide whether to run a branch. We call these **preconditions**. Figure 32 shows an example of a precondition. In this case, we want to check if we need to reload our weapon before we trigger the action.

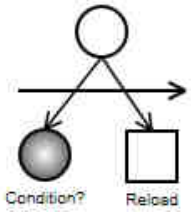


Figure 32. Example of a precondition

- Evaluation of a condition over a period of time, so we can get a failure if it is no longer met. We call these **assertions**. These nodes are more useful if they are combined with a *parallel*, as we show in Figure 33. Let us think attacking an enemy is not an instantaneous action; let us also say an enemy is required, if we want to run the action. Using this structure, if the condition fails at any time, the tree will bail out returning the failure.

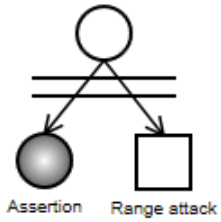


Figure 33. Example of an assertion

We have implemented this, as a single node that can function in two different modes.

```
template< class AIInstance >
ENodeResult Condition< AIInstance >::Step()
{
	bool bResult = m_pCondition->Evaluate();
	if ( m_bAssertion )
	{
		return ( bResult ) ? NR_IN_PROGRESS : NR_FAILED;
	}
	return ( bResult ) ? NR_SUCCEEDED : NR_FAILED;
}
```

We can translate this code into a tabular format to simplify it (Table 7).

Operation mode	Condition?	Node result
Precondition	TRUE	Succeeded
Precondition	FALSE	Failed
Assertion	TRUE	In progress
Assertion	FALSE	Failed

Table 7. Results obtained using the different operation modes of condition nodes

9.4.2. Actions

Actions are the nodes that modify the state of the world. They can perform different activities, such as modifying values, running an animation, creating new entities, etc. Because of this, they are very dependent on the game.

We considered two different design approaches to model actions. The first one was to think in advance of all the possible actions any game may need to perform and create an abstract layer games would have to implement. Using this solution, we could have created all of our action nodes in the library, but we would be limiting the set of actions videogames using the library could execute.

In contrast, the solution we chose is more flexible. We only define a basic, abstract action node, and games are in charge of creating their final action nodes.

We will show some actual implementations of actions in 12.2.1.1.

9.5. Filters

A filter is just a node that wraps a branch, so it alters its normal execution flow. They use a decorator pattern. Our library contains some useful decorators, which allow us to create things such as loops. Figure 34 show what filters have been implemented, as well as the relations existing among them.

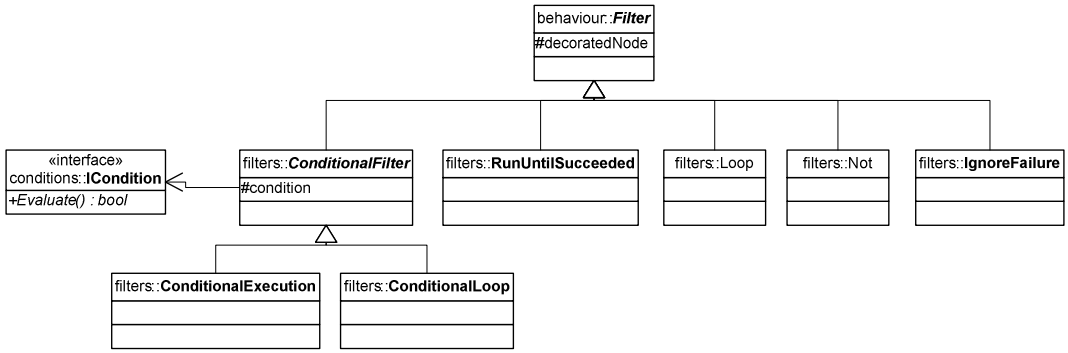


Figure 34. Filters that are available in the library

A simple example of the usage of a filter can be seen in Figure 35.

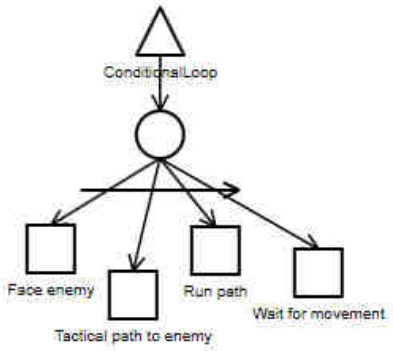


Figure 35. A filter working in a real-world example

In this case, we are modelling a simple zombie behaviour. We just want our zombie to chase its enemy while it is alive. Using a conditional loop that checks whether or not this is satisfied, we can build the final behaviour swiftly.

9.5.1. Loops

A loop is a node that runs a branch several times in a row. We have implemented three different types of loops, which are studied below.

9.5.1.1. Basic loop

The basic type of loop will just keep executing its underlying branch until it fails.

```
1  template< class AIInstance >
2  ENodeResult Loop< AIInstance >::Step()
3  {
4         ENodeResult eResult = m_pDecoratedNode->Step();
5         if ( eResult == NR_SUCCEEDED )
6         {
7               // Restart decorated node
8               m_pDecoratedNode->Stop();
9               m_pDecoratedNode->Start();
10
11              eResult = NR_IN_PROGRESS;
```

```
12          }
13
14          return eResult;
15    }
```

If the decorated node succeeds (line 5), we just restart it (lines 8-9) and return we are still in progress. This is explained in Table 8.

Child branch	Node result
Succeeded	In progress
Failed	Failed
Error	Error
In progress	In progress

Table 8. Possible results of a basic loop node

It is worth noting that a loop will never succeed, but just fail or keep running.

9.5.1.2. Conditional loop

This type of loop replicates the functionality of C++'s "while" loop: it will keep executing its child branch while a condition is met.

```
1   template< class AIInstance >
2   ENodeResult ConditionalLoop< AIInstance >::Step()
3   {
4         ASSERT_STR( m_pCondition != NULL, L"A condition wasn't set!" );
5         ENodeResult eResult = NR_SUCCEEDED;
6
7         if ( m_pCondition->Evaluate() )
8         {
9               eResult = m_pDecoratedNode->Step();
10              if ( eResult == NR_SUCCEEDED )
11              {
12                    // Restart decorated node
13                    m_pDecoratedNode->Stop();
14                    m_pDecoratedNode->Start();
15
16                    eResult = NR_IN_PROGRESS;
17              }
18        }
19
20        return eResult;
21  }
```

Let us express this as a table (Table 9).

Child branch	Condition?	Node result
Succeeded	TRUE	In progress
In progress	TRUE	In progress
Failed	TRUE	Failed
Error	TRUE	Error
-	FALSE	Succeeded

Table 9. Possible results of a conditional loop node

In this case, we have decided that the node will always succeed if the condition is not satisfied.

9.5.1.3. Run until succeeded

The last type of loop will restart the child branch indefinitely –ignoring all the possible failures that can happen in it– until it succeeds.

```
template< class AIInstance >
ENodeResult RunUntilSucceeded< AIInstance >::Step()
{
	ASSERT_STR( m_pDecoratedNode != NULL, L"NULL decorated node" );

	ENodeResult eResult = m_pDecoratedNode->Step();
	if ( eResult == NR_FAILED )
	{
		// Restart the node
		m_pDecoratedNode->Stop();
		m_pDecoratedNode->Start();

		eResult = NR_IN_PROGRESS;
	}

	return eResult;
}
```

Table 10 shows the possible results of this kind of node.

Child branch	Node result
Succeeded	Succeeded
Failed	In progress
Error	Error
In progress	In progress

Table 10. Possible results of a "Run until succeeded" node

From these results, we must note this node can never fail.

9.5.2. Conditional execution

This node will allow us to prune a branch quickly. It will bail out with a failure result, and without executing the child node, if a condition is not satisfied.

```
template< class AIInstance >
ENodeResult ConditionalExecution< AIInstance >::Step()
{
	ASSERT_STR( m_pCondition != NULL, L"A condition wasn't set!" );
	ENodeResult eResult = NR_FAILED;

	if ( m_pCondition->Evaluate() )
	{
		eResult = m_pDecoratedNode->Step();
	}

	return eResult;
}
```

The possible return values of this type of node are shown in Table 11.

Condition?	Node result
TRUE	Same as the child result
FALSE	Failed

Table 11. Possible results of a "conditional execution" node

9.5.3. Result modifiers

We call result modifiers those nodes that just run their decorated ones and convert their return value to produce a different output. Our library includes two of these nodes, which are presented next.

9.5.3.1. Not

This node will negate the result of the underlying branch, as shown in the following code snippet.

```
template< class AIInstance >
ENodeResult Not< AIInstance >::Step()
{
	ASSERT_STR( m_pDecoratedNode != NULL, L"NULL decorated node" );

	// Negate the result
	ENodeResult eResult = m_pDecoratedNode->Step();
	if ( eResult == NR_FAILED )
	{
		eResult = NR_SUCCEEDED;
	}
	else if ( eResult == NR_SUCCEEDED )
	{
		eResult = NR_FAILED;
	}

	return eResult;
```

```
}
```

It will, however, retain errors, as explained in Table 12.

Child node result	Node result
Succeeded	Failed
Failed	Succeeded
Error	Error

Table 12. Possible results of a "Not" filter

9.5.3.2. Ignore failure

This type of node converts node failures into successes.

```
template< class AIInstance >
ENodeResult IgnoreFailure< AIInstance >::Step()
{
	ASSERT_STR( m_pDecoratedNode != NULL, L"NULL decorated node" );

	// Ignore failures
	ENodeResult eResult = m_pDecoratedNode->Step();
	if ( eResult == NR_FAILED )
	{
		eResult = NR_SUCCEEDED;
	}

	return eResult;
}
```

As the code shows, it is a very basic filter, and its possible results are presented in Table 13.

Child node result	Node result
Failed	Succeeded
Any other result	Same as the child result

Table 13. Possible results of an "Ignore failure" filter

9.6. Running a behaviour

So far, we have defined all the basic elements in our library, but, how do they combine to produce a behaviour?

First, the game has to create an AI instance; it will do so through the AI manager. Once the instance is ready, the client will set a behaviour (tree), and the instance will run it until it is finished. Figure 36 illustrates this process.

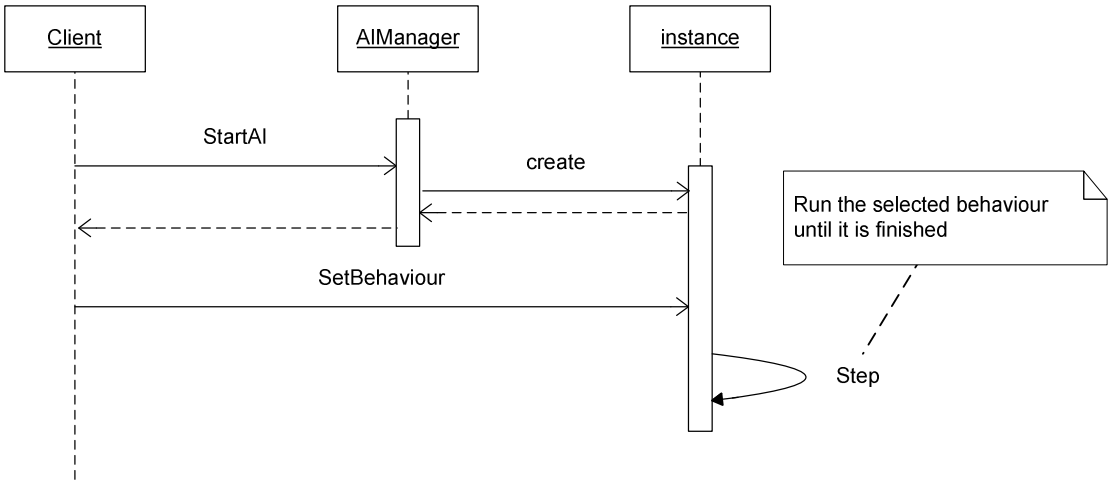


Figure 36. Process involved in the creation and initialisation of an AI

A BT is always run starting at its root. Let us show how a simple tree would be executed. To do that, we will use the zombie behaviour presented earlier (Figure 35), explaining the first four updates of the tree, which are outlined in Figure 37.

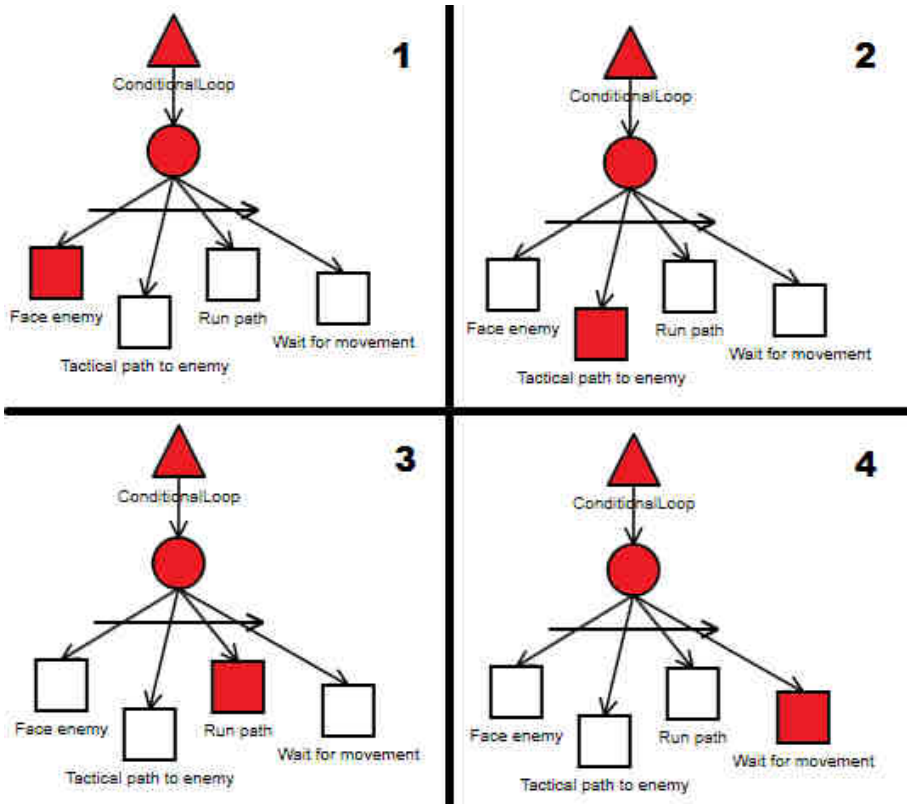


Figure 37. A BT is always executed starting from its root. The execution flow is determined by the type of nodes used to build the tree

In the first step, the root filter is executed for the first time. Let us say the condition is met (we are still alive), so the decorated node, in this case, a sequence, is run. The sequence starts running its first node, and it succeeds. The sequence will then update its current child and point to the second element in the sequence. The step ends here and the sequence returns it is in progress; the loop is in progress as well. The following steps are pretty similar, with leaf succeeding, and the sequence updating its current child accordingly.

Let us say "Wait for movement" takes some time to complete. In that case, step 4 would be different to the rest. Rather than having the leaf ending, it will return an "in progress" state, causing the sequence not to increment its iterator. This will continue for some updates, until the action is completed. When this happens, the sequence will know it has got to the end of its list of children, and thus will succeed. The main loop, upon receiving this success, will restart the sequence, and the following step would be the same as step 1, as depicted in the figure.

Finally, the AI's enemy decides to kill it. When the AI starts updating its tree, it will notice that on its root: the condition will no longer be satisfied, and the loop will bail out, succeeding, and ultimately, finishing the behaviour.

The execution flow in a behaviour tree is defined by the filters and metanodes we use in its creation. These nodes are the ones that make behaviour trees work as reactive planners, because they guide the execution towards different situations (sub-trees or sub-behaviours) depending on the state they detect.

In the following chapter we will introduce our novel technique, which will allow us to modify the execution flow without modifying the structure of the tree.

Chapter 10. EXPANDING OUR BT: HINTS

So far, we have described how our implementation of a standard behaviour tree system works. However, regular BTs do not allow us to prototype new behaviours swiftly and without additional programming work, so they do not meet our requirements.

In this chapter, we will examine our new approach, Hinted-execution Behaviour Trees, which try to overcome these problems.

10.1. The concept of hint

The main difference between a behaviour tree and a hinted-execution counterpart is that, while the execution flow for the former is defined by its own structure, HeBTs can reorder their branches dynamically to produce different results.

The system tries to imitate real-life command hierarchies, where lower-levels are told by higher-level ones what must be done, but, in the end, deciding how to do it is up to the individuals. In our system, the individual AI is a complex behaviour tree that controls the AI, so it behaves autonomously; this tree will probably be created and maintained by engineers.

We want other members of the team, not only programmers, to be able to test their ideas easily. A non-technical person will probably not be interested in how the AI works internally, but they only want to tell it to do things, i.e. they just want to be able to order the AI to "kill an enemy", rather than "find a path to your enemy, then get closer, draw your weapon and fire at your target, reloading your gun when you need to, etc.".

We call *"hint"* a piece of information an AI can receive from a higher-level source and use to produce an alternative behaviour, as a consequence of a priority reorder. This means an NPC, while maintaining its capability to respond properly to different situations, will take into account the requests coming higher in the command hierarchy to adapt its behaviour to these petitions.

10.2. Execution flow in behaviour trees

As we explained in 9.3, composite nodes are used to create complex structures in our trees. Among them, *selectors* provide trees with a way to choose, more or less intelligently, what to do when it comes to solving a problem. So, ultimately, selectors decide which branch should be executed at any given moment.

Recalling how the type of selector we have implemented in our prototype works, it will try each of its branches sequentially until one of them succeeds: its decisions are based on how its children nodes are ordered.

The AI will still be autonomous, as it will decide if a branch can be executed or not, but, basically, choosing which branch is the most important one, or what actions should be tried before others, is up to the engineers.

Hinted-execution behaviour trees provide a way for these priorities to be changed dynamically, just as we showed in Figure 21.

10.3. Implementing a hint system

Now that we have described the idea behind HeBTs, we have to modify our prototype to support hints. To do this, we are going to extend our selector nodes.

Selectors, as composite nodes, have a list of children sub-branches, each of which represents a possible action a higher-level will, potentially, want the node to choose. We will talk further about these higher-levels in 11.2.

Each of these branches is assigned a unique identifier. This identifier is assigned at creation time. This allows designers to name the branches, and therefore the hints that will favour their execution. Identifiers will be explained in 12.3.3. At any given time, a hint can be **positive**, **negative** or **neutral**; if a hint is *positive* the tree is being told to do something, if *negative* it is being told not to do something, and *neutral* if the selector is not receiving the hint at all. Identifier and state form our basic *HintInfo* structure.

Each selector keeps a list of hints it is able to accept, having one *HintInfo* per branch. This information will be used by the node to reorder its branches accordingly.

10.3.1. Updating hints

We use an *observer* pattern to manage the dispatching of hints. A new class, *BehaviourTree*, which wraps a root node and provides some basic functionality, will work as the *subscriber*, holding a list of observers (*IHintChangeReceiver*) that want to be notified when a hint is updated. Figure 38 shows how this is organised.

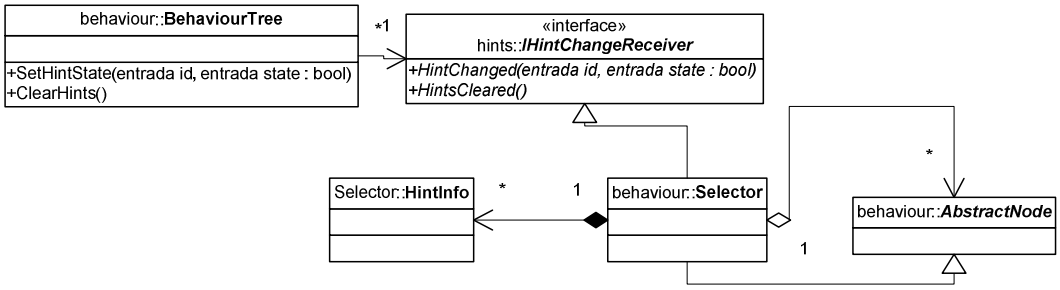


Figure 38. Structure of our improved selectors

When a new selector is added to a behaviour tree, it registers itself as a new observer. Also, the *BehaviourTree* keeps a map of *(hint_id, state)*, so it can correctly decide which nodes to notify.

The following code snippet shows how a tree dispatches a change of state in one of its hints to all the observers.

```
1  template< class AIInstance >
2  void BehaviourTree< AIInstance >::SetHintState( const hints::HintID& hintId,
3  const hints::EState eState )
4  {
5         HintMap::iterator it = m_hints.find( hintId );
6         ASSERT_STR( it != m_hints.end(), L"Invalid hint ID!" );
7
8         hints::EState ePreviousValue = it->second;
9         if ( ePreviousValue != eState )
10        {
11               it->second = eState;
12
13               // Notify all receivers
14               HintChangedReceiverVector::iterator receiverIt =
15                        m_hintReceivers.begin();
16               HintChangedReceiverVector::iterator receiverEnd =
17                        m_hintReceivers.end();
18               for ( ; receiverIt != receiverEnd; ++receiverIt )
19               {
20                     ( *receiverIt )->HintChanged( hintId, eState );
21               }
```

```
22          }
23  }
```

The BT starts looking for the previous state of the hint in its map of hints (line 5). If the state has changed (line 9), then it must notify all the receivers about this (lines 11-22). The observers can then ignore or process the message. Figure 39 illustrates the process.

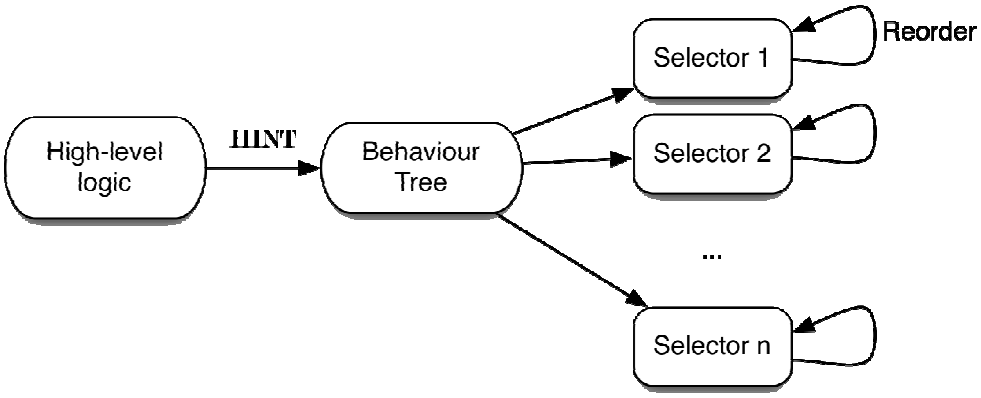


Figure 39. Hints are sent to BTs, which then notify all the selectors

It is important to note that when a tree receives a new hint, its normal execution is interrupted, and the tree gets restarted. This assures the tree is reordered properly and it also forces it to re-evaluate any condition that may have changed.

10.3.2. Reordering branches

Now that we have defined what a hint is (a state and a unique identifier), we need to modify our selectors, so they can use this new information.

Looking back at 9.3.3, normal selectors iterate through their children, looking for a branch that can be run. Rather than modifying this original list, our new selectors maintain three different lists: the first of them keeps the nodes that have been *positively hinted* (and thus, have more priority), the second one stores the nodes that have not been hinted (they are *neutral*), and the last one maintains the nodes that have been *negatively hinted* (they have reduced priority). These lists are still sorted using the original order, so if two or more nodes are hinted, AIs will know which action is more important according to their original behaviour. A couple of examples are shown in Table 14.

Original list	Hinted branches	High-priority	Normal-priority	Low-priority	New order of execution
1-2-3-4-5-6-7	1(+), 3(+), 4(-)	1-3	2-5-6-7	4	1-3-2-5-6-7-4
1-2-3-4-5-6-7	4(-), 5(-), 7(+)	7	1-2-3-6	4-5	7-1-2-3-6-4-5

Table 14. Some examples of how a list of branches is split into two lists, depending on priority

In code, this works as shown below.

```
1  template< class AIInstance >
2  void Selector< AIInstance >::UpdateIteratorVectors()
3  {
4         Clear();
5
6         NodeVector::iterator it = m_children.begin();
7         NodeVector::iterator end = m_children.end();
8         uint uiIndex = 0;
9         for ( ; it != end; ++it, ++uiIndex )
10        {
11              HintInfo& info = m_hints[ uiIndex ];
12              if ( info.id != INVALID_HINT_ID )
13              {
14                    m_hintedNodes[ info.eState ].push_back( it );
15              }
16              else
17              {
18                    // The hint wasn't named, so the sub-branch remains neutral
19                    m_hintedNodes[ hints::EState_Neutral ].push_back( it );
20              }
21        }
22
23        m_nextHintedNode = m_hintedNodes[ m_eCurrentState ].begin();
24        if ( m_nextHintedNode == m_hintedNodes[ m_eCurrentState ].end() )
25        {
26              IncrementNodeIterator();
27        }
28 }
```

The node iterates through its children (lines 6-9), and checks the state of the info assigned to that branch to decide which sub-list it will be assigned to (lines 12-20). After the branches have been reordered, the node must be restarted, so the correct child is selected, and any previous state is discarded.

The final result is a selector which is quite similar to the original one, but that can be changed dynamically to obtain different outputs. The *Step* method of the final node is explained below.

```
1  template< class AIInstance >
2  ENodeResult Selector< AIInstance >::Step()
3  {
4         ENodeResult eResult = NR_IN_PROGRESS;
5
6         // Get our current node
7         AbstractNode< AIInstance >* pNode = NULL;
8         if ( m_nextHintedNode != m_hintedNodes[ m_eCurrentState ].end() )
9         {
10                pNode = **m_nextHintedNode;
11        }
12        else
13        {
14                ERROR_STR( L"Don't have a valid node to update!" );
15        }
16
17        eResult = pNode->Step();
18        if ( eResult == NR_SUCCEEDED )
19        {
20                // Succeeded
21                Clear();
22                eResult = NR_SUCCEEDED;
23        }
24        else if ( eResult == NR_FAILED )
25        {
26                // Stop current task
27                pNode->Stop();
28
29                // Jump to next task
30                AbstractNode< AIInstance >* pNextNode = IncrementNodeIterator();
31                if ( pNextNode == NULL )
32                {
33                        // Failed, as no further task was found
34                        Clear();
35                        eResult = NR_FAILED;
36                }
37                else
38                {
39                        // Start new task and return that we're in progress
40                        if ( pNextNode->Start() )
41                        {
42                                eResult = NR_IN_PROGRESS;
43                        }
44                        else
45                        {
46                                eResult = NR_ERROR;
47                        }
48                }
49        }
50
51        return eResult;
52 }
```

First, the selector tries to find the node it has to run in this step (lines 7-15), from one of its lists. Then, it will execute it (line 17) and either bail out succeeding, if the node succeeded (lines 18-23) or move its iterators (line 30) in case the sub-branch fails, so in the next step the next node can be selected. If no node was found, the selector will bail out failing (lines 31-36), otherwise, it will start the new node and return it is still in progress (lines 39-48).

The method in charge of selecting the subsequent node has been implemented as follows:

```
1  template< class AIInstance >
2  AbstractNode< AIInstance >* Selector< AIInstance >::IncrementNodeIterator()
3  {
4         AbstractNode< AIInstance >* pNode = NULL;
5         while ( pNode == NULL )
6         {
7               if ( m_nextHintedNode != m_hintedNodes[ m_eCurrentState ].end() )
8               {
9                     // Increment the iterator
10                    ++m_nextHintedNode;
11              }
12              else
13              {
14                    // Move to the next vector of nodes
15                    m_eCurrentState =
16                          static_cast< hints::EState >( m_eCurrentState + 1 );
17                    if ( m_eCurrentState == hints::EState_Count )
18                    {
19                          // We have reached the last node
20                          break;
21                    }
22
23                    // Get our new iterator
24                    m_nextHintedNode =
25                           m_hintedNodes[ m_eCurrentState ].begin();
26              }
27
28              if ( m_nextHintedNode != m_hintedNodes[ m_eCurrentState ].end() )
29              {
30                    // Just use the current iterator...
31                    pNode = **m_nextHintedNode;
32              }
33        }
34        return pNode;
35 }
```

This method will return the next node the selector will have to use. Internally, we store three vectors of iterators to the node's list of children, one each per priority group. We store which our current state (priority group, either *positive*, *neutral* or *negative*) is, and we use it to select one of the vectors. We also keep an iterator to the last element we have used.

We try to look for a new node until we have found one (line 5) or we have run out of nodes (lines 17-21). On each step, we first try to increase the current iterator, and use it if we have not got to the end of the active vector (lines 7-11); otherwise, we increase our current state and re-initialise the iterator, so it uses the correct vector (lines 14-25). Finally, if we have found a valid one, we dereference it to get the node (lines 28-32). Note we have to dereference it twice, because

m_nextHintedNode is an iterator to a vector of iterators to the node's vector of children.

10.4. Hints and conditions

With the modifications presented so far, we have made our trees capable of accepting hints and reordering the branches controlled by their selector nodes. However, we can push this a little further if we add the possibility of designing the lower-level trees to make use of hints.

It is up to tree designers to expose whatever logic they feel is important to higher-levels, i.e. to choose what hints the tree will accept. The main objective of this thesis is to find a system in which high-level users do not have to worry about the internals of the system, but they can just use it easily. To achieve this, lower-level behaviour trees must have been designed correctly, having in mind how the system works, and making sure everything that is exposed works correctly.

Mostly, we can think of a behaviour tree as a collection of sequences that can be run by a selector, just as we show in Figure 40. In fact, if we repeat this structure recursively, we can get the basic shape of most BTs.

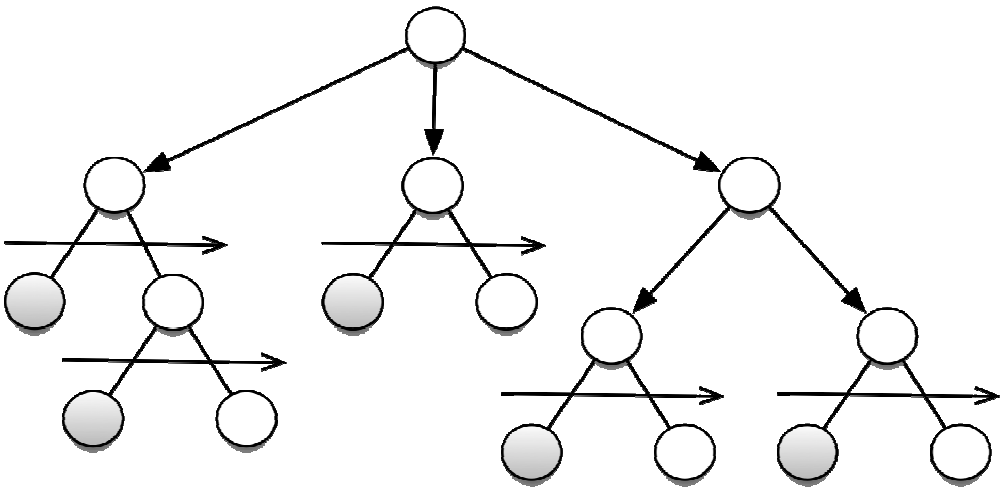


Figure 40. A behaviour tree can be seen as a structure made up of sequences which are run by selectors

Most of these sequences follow a basic pattern (Figure 41), where some condition nodes are placed as the first children, followed by actual actions. This way, the

actions will only get executed in these *preconditions* are met. If one of the conditions fails, the sequence will bail out, returning a failure, which will probably be caught by a selector that will then try to run a different branch.

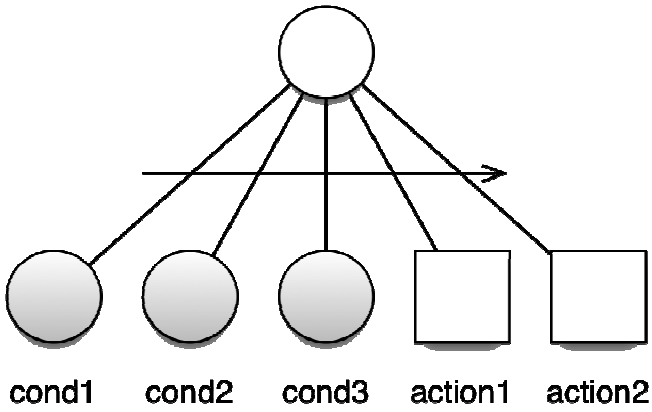


Figure 41. Basic sequence structure, where actions are preceded by a collection of preconditions

This is not good for our hints in some situations, as the conditions could be making a hinted branch fail and not be executed. We can expand the example presented in 7.4.2 to illustrate this. In that section, we did not expand the main branches, but just assumed everything would work. However, let us say that in our base tree, cover is only chosen if our health is low. As Figure 42 shows, in the event the condition was not met (which is not according to our current state), the branch we were hinting the tree to run was not chosen.

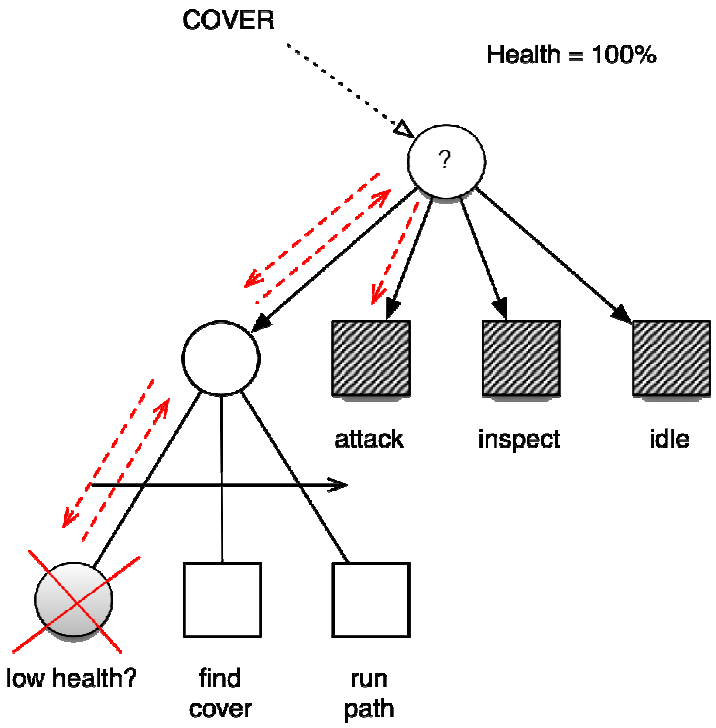


Figure 42. A bad design can lead to useless hints

So, we need a way to be able to ignore these preconditions **if** that makes sense, that is, if the condition is not really mandatory for the rest of the branch to be executed. In our example, we did not really need to be low on health to cover, but it was just something that was a design decision, trying to build a believable behaviour.

We have achieved this by:

- Allowing complex conditions to be created, so we can represent a complete list of preconditions in a single node. This was explained in detail in 9.4.1.
- Creating a new type of condition that is able to check the state of a hint, that is, whether it is active or not.

10.4.1. Hint condition

This type of condition keeps track of the state of a particular hint at any given moment. Internally, it implements *ICondition*, so it can be used as part of a complex condition.

These conditions store a reference to the behaviour tree they belong to; this will be used to query the state of the hint we are interested in, as shown below:

```
1  template< class AIInstance >
2  bool HintCondition< AIInstance >::Evaluate() const
3  {
4        return ( m_pTree->GetHintState( m_hintId ) == hints::EState_Positive );
5  }
```

It is important to note we have decided to only allow these nodes to check if a hint is being received with a positive state (line 4). We chose to implement hint conditions this way so we could keep condition trees as simple as possible.

Going back to our example, we can change our precondition and use a complex condition instead, where we check if we have been told to cover, as well as test our old condition. In that case, the precondition will allow the branch to be executed, and our hint would have worked as expected for higher-levels, just as shown in Figure 43.

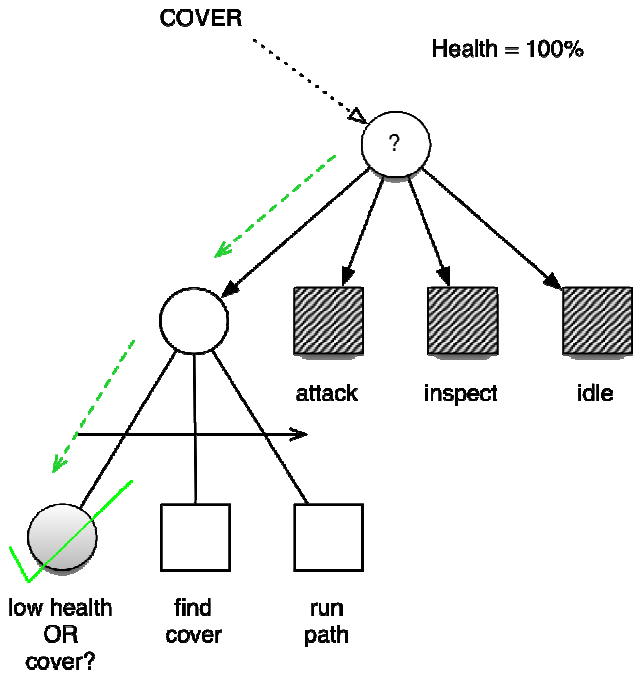


Figure 43. Using hint conditions we can overcome problems caused by preconditions

Chapter 11. MULTI-LEVEL ARCHITECTURE

In the previous chapter, we introduced the concept of hints, and how behaviour trees can be modified dynamically using them. These hints were sent to our trees by what we called "higher-levels of logic". Different approaches can be taken to implement these levels. For example, a quick and effective solution could be a layer of scripts that use the system to generate new behaviours. However, the usage of scripts makes things harder if we want everyone in the team to be able to build new behaviours, as they require some technical background.

A visual solution would be much more appropriate, as the ability to visualise things is much simpler than learning a new language and its rules. So, why not take advantage of the tools we have built to generate our base behaviour trees, and expand it, so it can be used by non-technical members of the team?

11.1. Behaviour controllers

A behaviour tree is a structure that reacts to the state of the world and executes actions depending on changes to its environment. So far, we have presented how regular behaviour trees work in Chapter 9, and expanded the model so the execution flow can be modified dynamically, in Chapter 10.

Behaviour trees are constructed using a set of building blocks, among which we have *actions*; they are the nodes in charge of modifying the environment or the state of the AI instance itself. Depending on the granularity of the system, these actions can be more or less complex, ranging from sub-behaviours, such as "cover", to atomic actions such as "find cover spot". For users not interested in how behaviours work, but just that they do work, the coarser the granularity, the simpler the system will be for them: they can work, test, and improve their ideas effortlessly if we can build a system that allows them to play with the AI, without worrying about details or needing external support, which is the core of this thesis.

Modifying a big BT can be complex, and could require taking into account quite a lot of variables. Also, small changes in a tree could lead to undesirable behaviours,

making AIs not work as expected. Because of this, we do not want new behaviours to be created from scratch, but we just want them to be flexible and malleable.

So let us keep a base tree, maintained by engineers, and provide the team with a tool to create new trees. We will use a different set of building blocks to create these new trees. Specifically, we will replace the *action* nodes with some new nodes called "*hint nodes*", which, as their name indicates, will send hints to our base tree. These new trees will work on top of our base behaviour, modifying it dynamically, and allowing designers to prototype new ideas easily and safely, as the main behaviour is not modified permanently, reducing risks. From now on, our AI instances will no longer be controlled by a single tree, but by a number of layers of behaviour trees.

In Figure 25, we showed each of our AIs owned a *BehaviourController*. These controllers are in charge of maintaining the multiple levels of trees an instance can use, and running all of them to produce the final results. Figure 44 shows the basic interface of one of these objects.

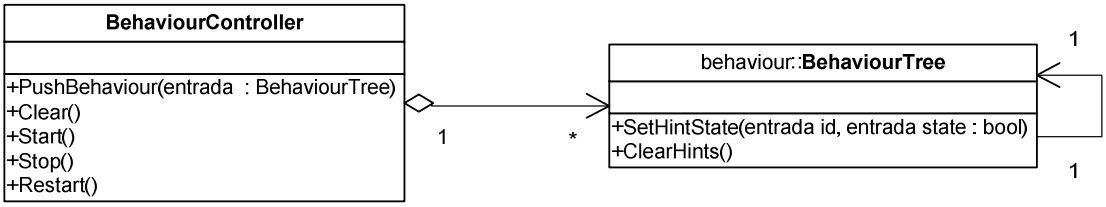


Figure 44. Behaviour controllers maintain a hierarchy of behaviour trees that will be run concurrently

A *BehaviourController* works as a stack, where we can push new trees. The top of the stack represents the highest level of logic, whereas the bottom contains the base behaviour tree. We have implemented this stack using an STL deque, as we want the top of the stack to be the first element (to simplify things when running HeBTs, as we will see later), as well as having a random access structure.

```
1  template< class AIInstance >
2  bool BehaviourController< AIInstance >::PushBehaviour( TreeType* pTree )
3  {
4         ASSERT_STR( pTree != NULL, L"Trying to add a NULL tree" );
5
6         // Get the immediate lower-level tree
7         TreeType* pLowerLevelTree = NULL;
```

```
8         if ( !m_trees.empty() )
9         {
10              pLowerLevelTree = m_trees.front();
11              pTree->SetLowLevelTree( pLowerLevelTree );
12        }
13
14        // Add the new tree
15        m_trees.push_front( pTree );
16
17        return true;
18  }
```

Every time we add a new tree, the controller informs the newly created high-level tree about what its immediate lower-level tree is (lines 10-11). This will allow hints to be sent to the correct BT. We can still create AIs controlled by old-plain behaviour trees, or modify them dynamically clearing the controller and pushing the correct trees back to the stack.

Behaviour controllers are the core components of hinted-execution behaviour trees, which we can now define as:

"A hierarchy of regular behaviour trees, where each level is able to hint its immediate lower level, and receive hints from the immediate higher one and the original behaviour remains at the lowest-level, and works as the interface with the AI's environment"

This is illustrated by Figure 45.

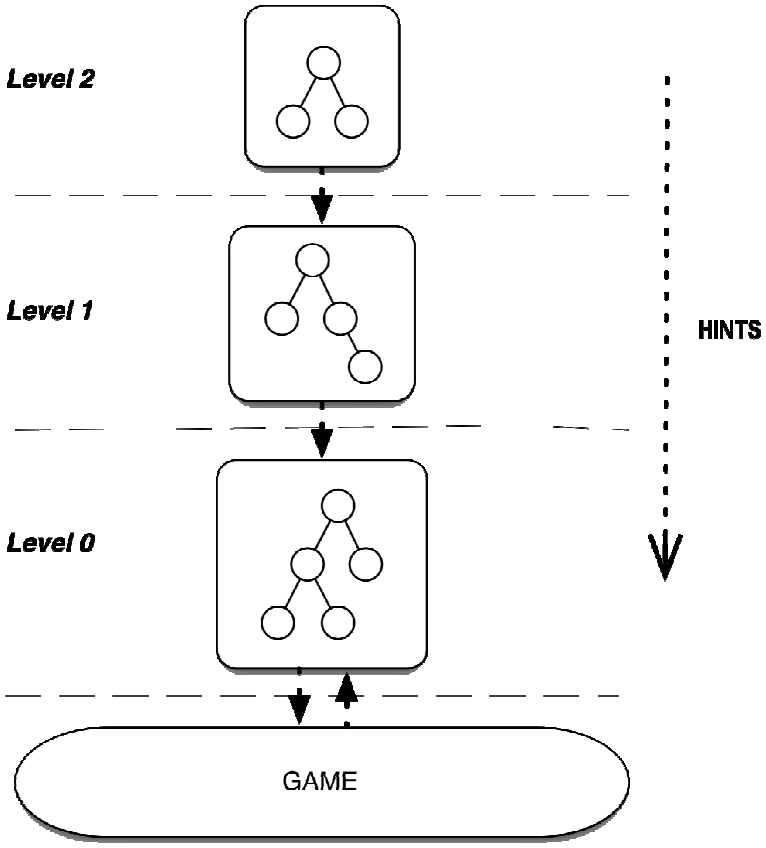


Figure 45. Structure of a Hinted-execution Behaviour Tree

The system must be totally transparent to high-level users, and they should know nothing about its internals. This has been addressed by enhancing our own BT editor, adding more complex functionality, such as allowing higher-level trees to be created, using any BT as a base, and allowing direct communication with the game, so new behaviours can be tested on the fly.

11.1.1. Running an HeBT

Once all the different trees have been created and registered with a *BehaviourController*, it can run the final behaviour.

Hinted-execution behaviour trees are run from the top down, so higher-levels are run first. This means that, by the time a tree is going to be executed, it would have already received all its hints, and their branches would be properly sorted, as shown in Figure 46.

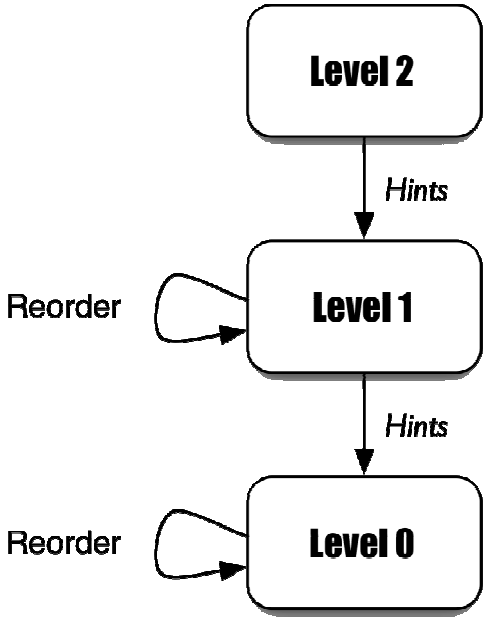


Figure 46. HeBTs are run from the top-down. This means low-level trees have been modified by the time they get executed

In code, the process look like this:

```
template< class AIInstance >
bool BehaviourController< AIInstance >::Step()
{
	TreeDeque::iterator it = m_trees.begin();
	while ( it != m_trees.end() )
	{
		TreeType* pTree = *it;
		if ( !pTree->IsStarted() )
		{
			// The tree was stopped, so remove it if the higher-levels
			// finished
			if ( it == m_trees.begin() )
			{
				SafeDelete< TreeType >( pTree );
				it = m_trees.erase( it );
			}
			else
			{
				// Move to next tree
				++it;
			}
		}
		else
		{
			thesis::behaviour::ENodeResult eResult = pTree->Step();
			if ( eResult == thesis::behaviour::NR_ERROR )
			{
				// The tree has returned an error, so we have to end
				// the behaviour now
				Clear();
				return false;
			}
			else if ( eResult != thesis::behaviour::NR_IN_PROGRESS )
			{
				// Stop this tree
```

```
36                              pTree->Stop();
37
38                              // Update the current iterator to the next valid
39                              // tree...
40                              ++it;
41
42                              // ... and clear its hints, if it exists
43                              if ( it != m_trees.end() )
44                              {
45                                    TreeType* pLowerLevelTree = *it;
46                                    pLowerLevelTree->ClearHints();
47                              }
48                        }
49                        else
50                        {
51                              // Move to next tree
52                              ++it;
53                        }
54                  }
55            }
56            return ( !m_trees.empty() );
57      }
```

We will iterate through our stack of trees (line 5), starting with the highest-level one, as we have already explained. The first thing we must note is that if a lower-level tree is stopped (either because it finishes or fails), we will not stop any other tree straight away; instead, we will maintain the tree in the stack, waiting for the higher-levels to finish their execution (lines 8-22). If our current tree is still running, we update it (line 25), and we can abort the execution if an error occurs (lines 26-32), and stop the current one, clearing all the hints it could have sent to the immediate lower-level tree (lines 33-48). If the tree is still running, we just increment our iterator (line 52) and keep executing the rest of the hierarchy.

By the end of the update, the AI will have run whatever action it has considered to have the higher priority, based on the information it has gathered from the environment **and** the hints it has received.

11.2. Exposing hints to higher-levels

High-level trees are built using a base BT to determine the hints that are available to it. When creating new trees, designers can name the branches of their selectors, and this would automatically expose that hint. In a similar way, if a condition hint is used anywhere in the tree, the hint will also be exposed.

As we showed in 10.3.1, behaviour trees maintain a map of hint ids and their states. When a tree is initialised, its nodes are processed, looking for hints that

have been named, and thus have to be exposed. The following code shows how our library handles this:

```
/**
 * Registers the hints a branch can handle
 */
template< class AIInstance >
void BehaviourTree< AIInstance >::RegisterHintsAndReceivers( NodeType* pNode )
{
	if ( pNode->GetType() == behaviour::NT_SELECTOR )
	{
		behaviour::Selector< AIInstance >* pSelector =
			static_cast< behaviour::Selector< AIInstance >* >( pNode );

		// Register node as a receiver
		m_hintReceivers.push_back( pSelector );

		// Register node's hints
		hints::HintIDVector hintIds;
		pSelector->FillHintIDs( hintIds );
		hints::HintIDVector::const_iterator it = hintIds.begin();
		hints::HintIDVector::const_iterator end = hintIds.end();
		for ( ; it != end; ++it )
		{
			const hints::HintID& hintId = *it;
			if ( m_hints.find( hintId ) == m_hints.end() )
			{
				// Add the new hint
				m_hints.insert(
				  std::pair< hints::HintID, hints::EState >(
					hintId, hints::EState_Neutral ) );
			}
		}
	}
	else if ( pNode->GetType() == behaviour::NT_CONDITION )
	{
		behaviour::Condition< AIInstance >* pCondition =
			static_cast< behaviour::Condition< AIInstance >* >(
					pNode );

		RegisterHintsInCondition( pCondition->GetCondition() );
	}
	else if ( pNode->GetType() == behaviour::NT_CONDITIONAL_FILTER )
	{
		behaviour::filters::ConditionalFilter< AIInstance >*
			pConditionalFilter =
			static_cast< behaviour::filters::ConditionalFilter<
				AIInstance >* >( pNode );

		RegisterHintsInCondition( pConditionalFilter->GetCondition() );
	}

	// Register children nodes
	unsigned int uiChildCount = pNode->GetChildCount();
	for ( unsigned int uiIndex = 0; uiIndex < uiChildCount; ++uiIndex )
	{
		RegisterHintsAndReceivers( pNode->GetChild( uiIndex ) );
	}

	// Register decorated nodes
	if ( ( pNode->GetType() == behaviour::NT_FILTER )
		|| ( pNode->GetType() == behaviour::NT_CONDITIONAL_FILTER ) )
	{
		behaviour::Filter< AIInstance >* pFilter =
			 static_cast< behaviour::Filter< AIInstance >* >( pNode );
		RegisterHintsAndReceivers( pFilter->GetDecoratedNode() );
```

```
64          }
65    }
66
67    /**
68     * Registers the hints present in a condition (or/and sub-conditions)
69     */
70    template< class AIInstance >
71    void BehaviourTree< AIInstance >::RegisterHintsInCondition(
72                                        conditions::ICondition* pCondition )
73    {
74          switch ( pCondition->GetType() )
75          {
76          case conditions::CT_AND:
77          case conditions::CT_OR:
78          {
79                conditions::CompositeCondition* pComposite =
80                      static_cast< conditions::CompositeCondition* >( pCondition
81    );
82                uint uiChildCount = pComposite->GetChildConditionCount();
83                for ( uint uiIndex = 0; uiIndex < uiChildCount; ++uiIndex )
84                {
85                      conditions::ICondition* pChild =
86                              pComposite->GetChildCondition( uiIndex );
87                      RegisterHintsInCondition( pChild );
88                }
89          }
90                break;
91          case conditions::CT_NOT:
92                {
93                      conditions::NotCondition* pNot =
94                         static_cast< conditions::NotCondition* >( pCondition );
95                      RegisterHintsInCondition( pNot->GetDecoratedCondition() );
96                }
97                break;
98          case conditions::CT_HINT:
99                {
100                     conditions::HintCondition< AIInstance >* pHintCondition =
101                         static_cast< conditions::HintCondition< AIInstance >*>(
102                                                pCondition );
103
104                     const hints::HintID& hintId = pHintCondition->GetHint();
105                     if ( m_hints.find( hintId ) == m_hints.end() )
106                     {
107                           // Add the new hint
108                           m_hints.insert(
109                            std::pair< hints::HintID, hints::EState >(
110                                        hintId, hints::EState_Neutral ) );
111                     }
112               }
113               break;
114         default:
115               break;
116         }
117   }
```

We have two methods that will be called recursively to add the new hints. The first of them (lines 1-64) processes behaviour tree nodes, adding the hints exposed in *selectors* (lines 7-31), while the second (lines 70-117) uses condition tree nodes, exposing hints used in *hint conditions* (lines 99-112). Hints are always added to the map in their *neutral* state (lines 26-28, 108-110).

To simplify things, our editor will allow new trees to be created based on existing ones, and it will automatically show which hints can be used, as we will see in Chapter 13.

11.3. Hint nodes

As demonstrated before in this chapter, high-level trees cannot use *actions*, but they can use *hint nodes* instead. These nodes allow trees to send any of the hints exposed by their lower-level trees, in order to produce new behaviours.

Internally, they are very simple nodes; their logic is only executed once, and they bail out succeeding right after the hint has been sent. Because of this, we have chosen to put this code in their *"Start"* method.

```
1  template< class AIInstance >
2  bool Hint< AIInstance >::Start()
3  {
4         if ( Parent::Start() && ( m_hintId != INVALID_HINT_ID ) )
5         {
6                BehaviourTree< AIInstance >* pLowLevelTree =
7                            m_pTree->GetLowLevelTree();
8                ASSERT_STR( pLowLevelTree != NULL,
9                   L"We must have a low-level tree if we want to use hints!" );
10               pLowLevelTree->SetHintState( m_hintId, m_eHintState );
11               pLowLevelTree->ReApplyHints();
12               return true;
13        }
14        return false;
15 }
```

The code enables a hint in the lower-level tree (line 10), and force hints to be re-applied (line 11), producing a shallow reorder of the branches of the tree (that is, their selectors re-build their internal structures).

It is important to note hint nodes can send different types of hints, allowing us to send *positive* or *negative* hints. They can also set a hint back to *neutral* if necessary.

11.3.1. Parallels and hint nodes

If we want to use a *parallel* node in our high-level logic, we can find ourselves trying to send two different hints at the same time. A bad design could lead us to a parallel sending both a negative and a positive version of the same hint, so, depending on the order the branches in the parallel are run, we could get different results (Figure 47).

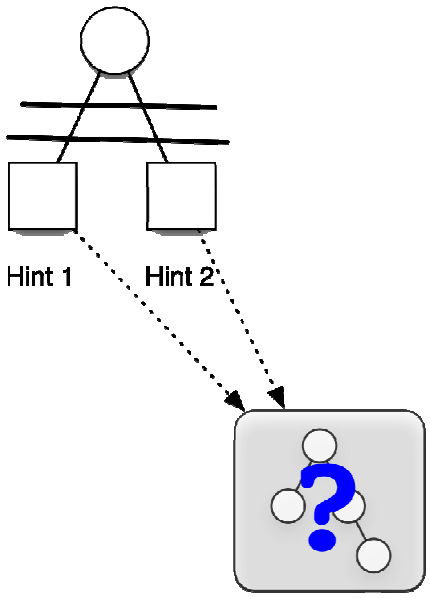


Figure 47. Depending on the order the children of a parallel get run, different hints would be sent to a lower-level tree, so we would get different results from the final behaviour

Parallels can cause odd behaviours, which sometimes require the usage of special types of nodes to control concurrency, such as resource allocators (Champandard, 2007), and they can definitely complicate things for non-technical people, so, depending on what we want to offer to final users, we could limit their use in our tools, as we will see later in Chapter 12.

11.4. Why allow more than two levels?

Most of this thesis is built around the idea of producing a system that can be used by designers to test and prototype new behaviours. From what has been presented so far, one can see that just using a level on top of our base trees would be enough for this purpose.

However, extensibility is one of the keys of Hinted-execution Behaviour Trees, and allowing an undefined number of levels opens the range of applications of these structures. None of these possibilities have been implemented as part of this thesis, as we decided this research would benefit from focusing on a particular problem, but some are pointed out later on in this document in 15.2.

Also, supporting multiple levels does not increase the complexity of the code or add any performance overhead.

Chapter 12. APPLYING HEBTS TO A GAME

In previous chapters we have presented our own implementation of a Hinted-execution Behaviour Tree system. We developed it as an abstract library, which can be adapted to, potentially, any type of game.

In order to exemplify how the library would be used in a real game environment, and also as a way to experiment with the new technique and obtain some results, we decided to build a game prototype, where the AI is controlled by our new approach.

This chapter studies how this prototype was built, and shows how hinted-execution behaviour trees can be part of a game and improve the way an AI system is created.

12.1. Choosing a game

The first step towards creating a good prototype was choosing the type of game we wanted to implement, and what tools we were going to use to do so. Developing a game is not a trivial task, and although it was an important part of our prototype, we decided not to spend unnecessary time trying to build a new framework, new art, sounds and all the different components that are part of a game. Instead, we decided that it was better to try and build our prototype on top of an existing game.

Many commercial games offer public SDKs so they can be extended or modified to create new experiences. Above all, FPSs[3] have traditionally been the type of game which released these tools. With this in mind, we considered three different games, and studied their suitability, according to our needs. These games were *Doom 3* (id Software, 2004), *Unreal*, which had recently released a public version of its engine, called *UDK* (Epic, 2009), and *Half-Life 2* (Valve, 2004).

[3] First Person Shooter, a genre started by some classic games such as Wolfenstein 3D, Doom or Quake, in which players control a character in first person, and move through complex environments, eliminating enemies using various different weapons.

UDK was an interesting option, as it basically offers one of the most widely used engines in the industry. However, we soon discarded it, because our library was written in C++. *Unreal Engine* is controlled by its own script language, *UnrealScript*, and only allows C++ code to be used using non-free licenses.

Doom was a second option. There is quite a lot of information on the internet about it, and we had some experience developing mods[4] for *Quake 2*, which used a previous iteration of id Software's Tech engines (*Doom 3* uses *id Tech 4*, while *Quake 2* used *id Tech 2*). A simple prototype was developed in a week, but, in contrast to *Quake*, which was written in C, *Doom's* AI was heavily scripted, using id's proprietary language. Although it was possible to move some key functionality back to code, so we could control game entities using HeBTs (a simple action to make an AI walk towards the player was implemented), it was going to take too long, and our research was not really going to benefit from it, so the prototype was dropped.

Finally, we opted for *Half-Life 2 (HL2)*. This game, and its predecessor, is extremely popular among mod communities; for example, the popular *Counter-Strike* (Valve, 2000) was first created as a mod of the original *Half-Life*.

12.1.1. Half-Life and the Source Engine

Half-Life 2 was built using Valve's Source Engine. This engine has been used in many games ever since, such as *Portal* (Valve, 2007) or *Left 4 Dead* (Valve, 2008). Source's shared AI (that is, the systems that are common to Source games) is implemented in C++, using a powerful technique, which is quite similar to a behaviour tree. Each AI is controlled by *schedules*. A *schedule* is a list of *tasks* the AI has to perform, and it is chosen based on a *state* and some *conditions*. There are some other concepts such as *behaviours*, but we can basically define an AI just by using these basic concepts.

The game defines a set of entities that implement their own behaviours, by defining new schedules and tasks (or using common ones). Then, the entity will run as follows (Valve, 2010):

[4] Game modifications are usually nicknamed "mods" in game communities.

1. The AI performs **sensing**, building lists of important entities or events (such as sounds).
2. A **list of conditions** is generated based on the lists built in point 1.
3. A ***state*** is chosen, based on the conditions.
4. A new ***schedule*** is selected if necessary, based on the current *state* and the list of conditions.
5. The current ***task*** in the *schedule* is run.

Entities inherit from various classes, including *CAI_BaseNPC*, which implements the sequence shown above.

12.2. Replacing Half-Life's AI system

The best way to make HL2 use our technique was to implement our own NPC class, as Valve similarly does in the SDK. In our case we decided to inherit from one of *CAI_BaseNPC's* children, *CAI_BaseActor*, which is the one that supports humanoid entities, like the ones we want to create; rather than defining tasks and schedules, we overwrote some of the entity's main virtual functions, particularly *RunAI*, which is the one in charge of stepping a normal AI's logic. We kept part of their logic (for example, we wanted to maintain the sensory system, so we did not have to implement our own), but removed all references to schedule updating. Our final method looks like this:

```
1  void CNPC_Thesis::RunAI()
2  {
3         // Gather conditions
4         AI_PROFILE_SCOPE_BEGIN(CAI_BaseNPC_RunAI_GatherConditions);
5         GatherConditions();
6         RemoveIgnoredConditions();
7         AI_PROFILE_SCOPE_END();
8
9         // Update the activity (animation)
10        MaintainActivity();
11
12        ClearTransientConditions();
13 }
```

Lines 4-7 and 12 are all related to performing sensing. Line 10 is in charge of maintaining a correct animation. This is all our agents need from Half-Life.

Recalling what we studied in 9.1, our library uses an AI Manager, which maintains and updates a list of AI instances. These instances have a pointer to a native entity, and receive its type as a template parameter.

In order to register our native instances with our manager, we had to make it a singleton, and made our NPC use it on construction/destruction, as shown below:

```
/**
 * Constructor
 */
CNPC_Thesis::CNPC_Thesis()
{
	// Register in the manager
	m_pAI = ThesisAIManager::Ref().StartAI( this );
}

/**
 * Destructor
 */
CNPC_Thesis::~CNPC_Thesis()
{
	m_pAI = NULL;

	// Unregister in the manager
	ThesisAIManager::Ref().StopAI( this );
}
```

Each AI is controlled by a behaviour controller, and, ultimately, shows its behaviour by executing different actions, or nodes that affect the game.

12.2.1. Creating custom nodes

Most of our system is game-independent; this means we can use it in different games without any modification. Because the system is so abstract, we cannot predict which actions or conditions will be needed in trees built for a particular game. The library delegates this responsibility to the game, which is in charge of providing its own actions and conditions. Let us build some examples.

12.2.1.1. Implementing a simple action

For instance, we will implement an action node to make an AI look at the player. We will call this node *FaceEnemy*. To do so, we will need to create a new class, which, in this case, will be included in the game's project, and make this new action inherit from our library's *Action* class (see 9.4.2). Once we have done this, we just have to overwrite its *Step* method as follows:

```
1  behaviour::ENodeResult FaceEnemy::Step()
2  {
3         // Get our native entity
4         CNPC_Thesis* pAI = m_pInstance->GetNativeAI();
5
6         // We need to have an enemy to run this action
7         if ( ! pAI->GetEnemy() )
8         {
9               return behaviour::NR_FAILED;
10        }
11
12        // Get our enemy's position
13        Vector vecEnemyLKP = pAI->GetEnemyLKP();
14        if (!pAI->FInAimCone( vecEnemyLKP ))
15        {
16              // Our enemy is inside our cone of vision
17              // CalcReasonableFacing() is based on previously set ideal yaw
18              pAI->GetMotor()->SetIdealYawToTarget( vecEnemyLKP );
19              pAI->GetMotor()->SetIdealYaw(
20                        pAI->CalcReasonableFacing( true ) );
21        }
22        else
23        {
24              // Not in our cone of vision... only set a new yaw if it's
25              // reasonable
26              float flReasonableFacing = pAI->CalcReasonableFacing( true );
27              if ( fabsf( flReasonableFacing - pAI->GetMotor()->GetIdealYaw() )
28                                                                    > 1 )
29              {
30                    pAI->GetMotor()->SetIdealYaw( flReasonableFacing );
31              }
32        }
33
34        // Update our entity's yaw
35        pAI->GetMotor()->UpdateYaw();
36
37        // Succeed if we're already facing our ideal yaw!
38        return ( pAI->FacingIdeal() ) ? behaviour::NR_SUCCEEDED
39                                      : behaviour::NR_IN_PROGRESS;
40 }
```

The action node acts through the native entity, which works as our interface with the game. In this case, the logic is quite simple. First, the AI checks whether it has spotted an enemy (line 7), and the action fails and bails out in case it has not (line 9). Otherwise, lines 13-31 calculate a new yaw for the entity, using the player's (enemy) last known position (LKP, line 13). Then, the AI updates its yaw (line 35) and finally succeeds if it is already facing the player; if it is not, the action will continue to be in progress.

So let us test this node. In order to do that, we will create a simple behaviour tree (we do not need to use an HeBT now, as a single tree will be enough) like the one shown in Figure 48.

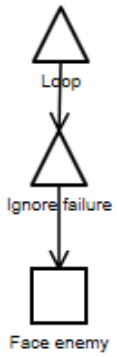


Figure 48. A simple BT that will make an AI face the player indefinitely

The tree uses a couple of filters (see 9.5). The first one is an "Ignore failure"; we need this one as our action will fail when the AI does not detect us as an enemy. Because the AI is static –as we have not added any movement action–, this will happen as soon as we walk away from it. The second filter is an unconditional loop, which will make sure our logic is executed constantly, unless an error occurs.

Running this tree in the game, we would have a static AI that rotates on itself, facing the player as he moves around. Figure 49 shows two screenshots, taken at different positions in our test level.

Figure 49. Our first BT will make entities face us, but they will maintain their positions

12.2.1.2. Implementing a simple condition

Similarly to how we created our action nodes, conditions are created by inheriting from a class in our library. In this case, we want to create a *LeafCondition* (9.4.1).

We will create a condition to check whether our AI is alive. The logic resides in the *Evaluate* method, which we show below:

```
1  /**
2   * Evaluates the condition
3   */
4  bool Alive::Evaluate() const
5  {
6        return ( m_pInstance->GetNativeAI()->GetHealth() > 0 );
7  }
```

Easy enough, we just have to check if our native AI's health is greater than zero (line 6). We could use this action in the tree presented in Figure 48, and replace our loop with a conditional one, so the logic is stopped when the agent dies.

12.3. Building and using HeBTs

Although possible, building our trees directly in code is a tedious process, and makes the whole idea of prototyping useless. That is why we need a way to load and set behaviours in the game: we want our AI to be data-driven.

We have chosen to integrate a Lua interpreter into our project, because it is free and easy to use. Our library contains a *LuaManager*, which exposes some basic functionality to run Lua commands or load and run script files. Our final goal is to be able to build new trees using Lua commands, so we need to implement a set of commands to do this.

Lua provides developers with a way to call C functions from script, so we can map our commands to methods in code that instantiate and create links among nodes. The process is pretty straightforward, but we have simplified it by introducing *LuaInterfaces*. These interfaces define all the functions we want to expose, along with the commands we want to map them to.

Basically, a command is a C function of this type:

```
typedef int ( *FunctionPtr ) ( lua_State* );
```

Our LuaManager allows us to register these functions using the following method:

```
void LuaManager::RegisterFunction( FunctionPtr f,
                                   const std::string& name )
{
      lua_register( m_pLuaState, name.c_str(), f );
}
```

As we have said before, we are trying to build an abstract system. Because our library is so abstract, we cannot instantiate any type of node –even the common ones, such as sequences– because they are templatised. This forces us to create a specific Lua interface for each new game.

For our prototype, we have created a *LuaBTInterface*. It is created when Half-Life is loaded, and it registers a set of commands we can then use in our scripts to generate new trees.

Internally, the interface uses a stack of nodes; this allows it to maintain the tree hierarchy while building it, pushing nodes onto it and adding links when the nodes are removed. Most of our commands are actually pairs, that is, a command to start the definition of a node, and another one to end it. Many times, we can think of "*start commands*" as commands that push nodes, and "*end commands*" as those which pop them.

Trees follow this structure, and we have two commands to start and end a new tree: **startBehaviour** and **endBehaviour**. The code for start a new behaviour is shown below:

```
int LuaBTInterface::StartBehaviour( lua_State* L )
{
      ASSERT_STR( s_nodeStack.empty(),
                  L"Some nodes were present in the stack of nodes!" );
      ASSERT_STR( s_pCurrentInstance == NULL,
                  L"A previous AI was already present!" );
      ASSERT_STR( s_pCurrentTree == NULL,
                  L"A previous tree was already present!" );

      // Clear previous stack. This SHOULDN'T HAPPEN! as it should be
      // empty
      while ( !s_nodeStack.empty() )
      {
            ThesisAbstractNode* pNode = s_nodeStack.top();
            delete pNode;
            s_nodeStack.pop();
      }

      // Get the AI we're creating the behaviour for. It's ID has to
      // be stored in a global variable
      lua_getglobal( L, CURRENT_AI_GLOBAL_VARIABLE_NAME );
      ThesisAIInstance::UIDType uid =
```

```
23                  static_cast< ThesisAIInstance::UIDType >(
24                                      lua_tonumber( L, 1 ) );
25        lua_pop( L, 1 );
26
27        s_pCurrentInstance = ThesisAIManager::Ref().GetAI( uid );
28        ASSERT_STR( s_pCurrentInstance != NULL,
29                    L"Can't find a valid instance!" );
30
31        // Create a new behaviour tree. It won't be initialised until
32        // EndBehaviour is called
33        s_pCurrentTree = new ThesisBehaviourTree();
34
35        return 0;
36  }
```

Lines 3-8 present some basic integrity checks. Our interface keeps a current instance and tree, along with the stack of nodes, and, at this point, everything should be empty or null. We need a way to define which AI we are creating the tree for. Instances have a unique numeric identifier, which is used to tell the manager what instance we want to operate in. In our case, we have chosen to pass this number as a global variable to the Lua interpreter. Lines 21-29 use this data to obtain a pointer to the AI instance. Finally, line 33 creates a new tree, and everything would be ready to create our new behaviour.

We need to call **endBehaviour** once we have built our tree:

```
1   int LuaBTInterface::EndBehaviour( lua_State* L )
2   {
3         ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );
4         ASSERT_STR( s_nodeStack.size() == 1,
5                     L"No node (or more than one) was found in the stack!" );
6
7         // Initialises the tree, as the top element in the stack of nodes
8         s_pCurrentTree->Initialise( s_nodeStack.top() );
9         s_nodeStack.pop();
10
11        // Pushes the behaviour
12        s_pCurrentInstance->PushBehaviour( s_pCurrentTree );
13
14        // Clear the entity PTR
15        s_pCurrentInstance = NULL;
16
17        // Clear the pointer to the tree
18        s_pCurrentTree = NULL;
19
20        return 0;
21  }
```

First, we make a couple of integrity tests (lines 3-5) to check that we have called the right commands before ending our behaviour. After that, we initialise our new tree with the node at the top of the stack; at this point, all the links should have been created, and we should only have one node in the stack, which is the root of

our tree (lines 8-9). We will then push the newly created behaviour onto our instance's *behaviour controller* (line 12), and clear the interface object (15-18).

12.3.1. Exporting a simple BT

So, how do we build a tree? Let us build our "face enemy" tree using a script. It will look like this (note that we have added some indentation to the code, so it is easy to read):

```
1  startBehaviour()
2        startLoop()
3              startIgnoreFailure()
4                    addFaceEnemyAction()
5              endNode()
6        endNode()
7  endBehaviour()
```

Lines 1 and 7 are already known, and just define the start and the end of our tree. Line 2 starts a new *loop* filter. In code, this is mapped to the following method:

```
1  template< class T >
2  int LuaBTInterface::StartCompositeNode( lua_State* L )
3  {
4        ASSERT_STR( s_pCurrentInstance != NULL,
5                    L"No valid AI was specified!" );
6
7        // Create a new node and add it to the stack
8        T* pNode = new T();
9        pNode->Initialise( s_pCurrentInstance, s_pCurrentTree );
10       s_nodeStack.push( pNode );
11
12       return 0;
13 }
```

To simplify things, we have implemented a template method to create any composite (that is, any non-leaf node) node. This method creates a new instance of the given type (line 8), initialises it (line 9) and pushes it onto the stack (line 10). In our sample script, we have two of these commands: *startLoop* and *startIgnoreFailure*.

At this point, we would have two *filters* in the stack. We need to add our action now, so we call *addFaceEnemyAction*:

```
template< class T >
int LuaBTInterface::AddAction( lua_State* L )
{
	ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );

	T* pAction = new T();
	pAction->Initialise( s_pCurrentInstance, s_pCurrentTree );
	s_nodeStack.push( pAction );
	EndNode( L );

	return 0;
}
```

The code for this action is pretty similar to that used by filters. Note, however, that we are calling *EndNode* in line 10. This is the same command we are executing in our script to end our loops (lines 5 and 6 in the script listing), but we decided it was clearer to be able to add leaf nodes using a single command. Let us study what this code does:

```
int LuaBTInterface::EndNode( lua_State* L )
{
	ASSERT_STR( !s_nodeStack.empty(), L"No node was found in the stack!" );
	ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );

	if ( s_nodeStack.size() > 1 ) // If the node is the last one, just leave
	                              // it in the stack
	{
		// Get the top of the stack
		ThesisAbstractNode* pNode = s_nodeStack.top();
		s_nodeStack.pop();

		// Add to parent
		ASSERT( pNode );
		AddToParent( pNode );
	}

	return 0;
}
```

This method is in charge of adding links between nodes. To do so, it first gets the top of the stack (line 10-11) and calls *AddToParent* (line 15), which will add that node correctly to its parent, depending on its type. We only do this if we have more than one node in the stack, that is, if the node is not the root of the tree. So, let us inspect the remaining method:

```
void LuaBTInterface::AddToParent( ThesisAbstractNode* pChildNode )
{
	ASSERT_STR( !s_nodeStack.empty(),
	            L"No parent node was found in the stack!" );
	ASSERT_STR( pChildNode != NULL, L"No valid child node was specified!" );

	ThesisAbstractNode* pParent = s_nodeStack.top();
	ASSERT( pParent != NULL );

	// Add to parent, depending on its type
	switch ( pParent->GetType() )
	{
```

```
13          case behaviour::NT_SEQUENCE:
14          case behaviour::NT_SELECTOR:
15          case behaviour::NT_PARALLEL:
16                {
17                      behaviour::Composite< ThesisAIInstance >* pComplex =
18                            static_cast< behaviour::Composite<
19                                                ThesisAIInstance >* >( pParent );
20                      pComplex->AddChild( pChildNode );
21                }
22                break;
23          case behaviour::NT_FILTER:
24          case behaviour::NT_CONDITIONAL_FILTER:
25                {
26                      behaviour::Filter< ThesisAIInstance >* pFilter =
27                            static_cast< behaviour::Filter<
28                                                ThesisAIInstance >* >( pParent );
29                      pFilter->SetDecoratedNode( pChildNode );
30                }
31                break;
32          default:
33                ERROR_STR( L"Invalid parent type!" );
34                break;
35          }
36    }
```

The method receives the node we want to process. It will then get the parent node, which is now at the top of the stack (line 7). Finally, depending on the type of the node, it will add it either as a child (lines 13-21), in case we are dealing with a *metanode*, or as the decorated node (lines 23-31) in case it is a *filter*.

So, trees are built from the bottom up. Table 15 summarises what the tree will look like during the building process:

Lua command	Stack	State of the tree
startBehaviour()	-	-
startLoop()	Loop	-
startIgnoreFailure()	IgnoreFailure Loop	-
addFaceEnemyAction()	IgnoreFailure Loop	FaceEnemy
endNode()	Loop	IgnoreFailure | FaceEnemy
endNode()	Loop	Loop | IgnoreFailure | FaceEnemy
endBehaviour()	-	Tree is set to the AI

Table 15. Process followed to build a tree from a Lua script

12.3.2. Exporting condition trees

In the previous example, we did not use any condition at all; we will exchange our *loop* for a conditional one, and see how a complex condition tree would be declared using Lua commands.

Let us say we want our AI to face the player only if it can see him/her and its health level is low. This condition is represented by the condition tree shown in Figure 50.

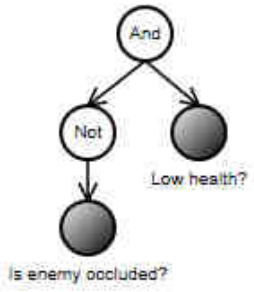


Figure 50. Condition tree we will export to Lua

Conditions follow the same structure we used to declare trees, so they are built using their own stack of condition tree nodes. Any node that requires a condition must define it right after its start command. For example, replacing the root of our previous tree, we would end up with a script like this:

```
1  startBehaviour()
2         startConditionalLoop()
3                startAndCondition()
4                      startNotCondition()
5                            addEnemyOccludedCondition()
6                      endCondition()
7                      addLowHealthCondition()
8                endCondition()
9                startIgnoreFailure()
10                     addFaceEnemyAction()
11               endNode()
12        endNode()
13 endBehaviour()
```

The condition tree is built in lines 3-8. Again, complex nodes are started and ended (for example, lines 4 and 6), while leaf nodes, that is, the actual conditions, are constructed using a single command.

It is worth noting that the last *endCondition* (line 8), which ends the condition tree, adds the newly created condition to the BT node that is at the top of the stack. In code, we implemented it like this:

```
1  int LuaBTInterface::EndCondition( lua_State* L )
2  {
3         ASSERT_STR( !s_nodeStack.empty(), L"No node was found in the stack!" );
4         ASSERT_STR( !s_conditionStack.empty(),
5                     L"No condition was found in the stack!" );
6
7         if ( s_conditionStack.size() > 1 )
8         {
9               // Get the top of the stack
10              behaviour::conditions::ICondition* pCondition =
11                           s_conditionStack.top();
12              s_conditionStack.pop();
13
14              // Add to parent
15              ASSERT( pCondition );
16              AddToParentCondition( pCondition );
17        }
18        else
19        {
20              //-- Set the condition to the node we're currently creating (the
21              //   one at the top of the stack of nodes)
22
23              // Get the condition
24              behaviour::conditions::ICondition* pCondition =
25                           s_conditionStack.top();
26              s_conditionStack.pop();
27
28              // Get node
29              behaviour::AbstractNode< ThesisAIInstance >* pNode =
30                           s_nodeStack.top();
31              if ( pNode->GetType() == behaviour::NT_CONDITION )
32              {
33                    behaviour::Condition< ThesisAIInstance >* pConditionNode =
34                                 static_cast< behaviour::Condition<
35                                                ThesisAIInstance >* >( pNode );
36                    pConditionNode->SetCondition( pCondition );
37              }
38              else if ( pNode->GetType() == behaviour::NT_CONDITIONAL_FILTER )
39              {
40                    behaviour::filters::ConditionalLoop< ThesisAIInstance >*
41                                 pFilter = static_cast<
42                                 behaviour::filters::ConditionalLoop<
43                                     ThesisAIInstance >* >( pNode );
44                    pFilter->SetCondition( pCondition );
45              }
46              else
47              {
48                    ERROR_STR( L"Not implemented yet..." );
49              }
50        }
51
52        return 0;
53 }
```

Lines 11-17 are in charge of adding links between nodes in the condition tree, similar to what we did in the previous section to build relations between BT nodes. As we said, in the event that the condition stack contains only one node, it

means we have to assign the complex condition we have just created to its parent BT node (which we get from the top of the BT stack in lines 29-30). Depending on the type of the parent node, we would have to various operations (lines 31-49).

12.3.3. Building an HeBT

So far, we have studied how regular behaviour trees are created, so we must describe how our hierarchy of trees are exported. Specifically, we need to define:

- How a base tree declares the list of hints that it will accept.
- How a high-level tree declares its hint nodes.

We will do so presenting a slightly more complex example.

12.3.3.1. Creating a base behaviour tree

First, we will create a complex base tree that will make our AI behave autonomously. Our agents will:

- Cover if their health is too low.
- Face their enemy and attack it (a range attack will be used). They will also have to reload their weapons if necessary.
- Chase their enemy if this tries to hide.

The final behaviour tree will look like the one shown in Figure 51.

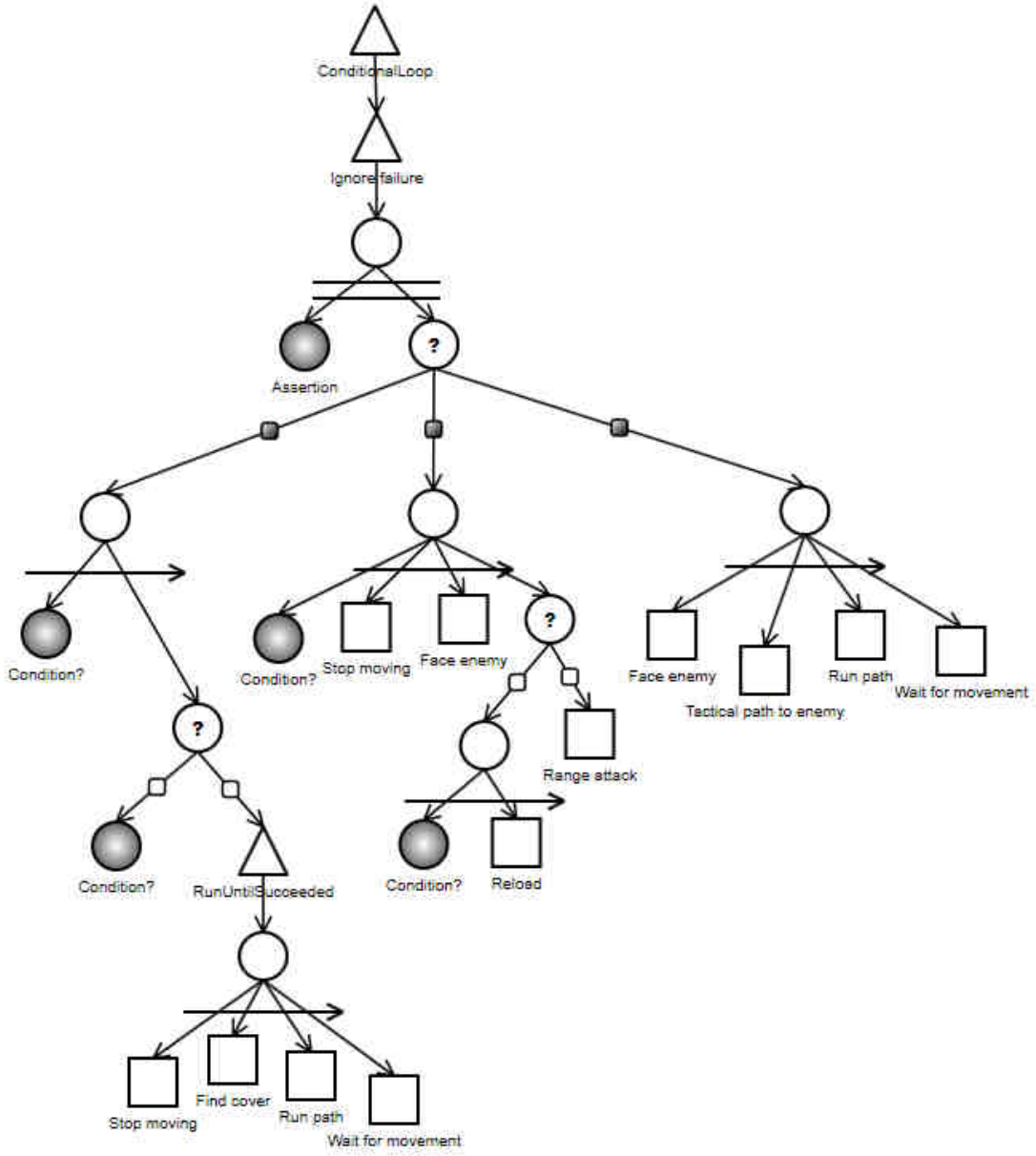


Figure 51. A complex behaviour tree that will make our agents behave autonomously

This behaviour presents a main selector that is run in parallel with an *assertion*, checking that the AI has an enemy. We want to ignore any possible failure, and keep executing this while the instance is alive, so we have added two *filters* at the root of the tree.

Our selector has three branches: COVER, ATTACK and CHASE. They look like good candidates to be exposed as hints to higher-levels, so we have done so[5].

[5] Most of the figures showing behaviour trees that are used in this dissertation have been captured directly from our editor. How the editor works, and thus how we can expose hints will be studied in 0

Our COVER branch starts with a precondition, which will test whether the instance should try to cover; particularly, it will test whether its health level is too low. Once we have decided this is the right branch to run, we will either stop if we are already in a cover position; otherwise, we will find one and run towards it until we are covered. We have modelled this as a selector that will first check if we are in a proper position, or will look for one as a secondary option, because we have sorted our branches in that order[6].

If for any reason our COVER branch fails –most likely because our precondition as failed–, our AI will try to attack its enemy. To do so, we use a sequence with a precondition, which is checking whether our enemy is visible, that will make the AI face its enemy and either attack it or reload its weapon. Again, we have used a selector to decide what is best to do.

Finally, if our AI has not been able to COVER or ATTACK us yet, we are their enemy, it means it has to try to find and chase us. We have built this as a simple sequence.

Exporting this tree to Lua, we end up with a quite long script (73 lines), so we will only highlight the most important sections. Undoubtedly, the key of a base tree, from an HeBT point of view, resides in exporting selectors correctly, and exposing their branches as *hints*, so they are available for higher-level trees. Exporting our main selector to Lua will produce something similar to this:

[6] Note our trees are read from the left to the right.

```
startSelector()
	startBranch( "COVER" )
	...
	endBranch()
	startBranch( "ATTACK" )
	...
	endBranch()
	startBranch( "CHASE" )
	...
	endBranch()
endNode()
```

Whereas, for instance, the process followed to build a *sequence* from Lua relies on BT Lua interface's node stack, constructing a selector requires the use of some additional commands, so we can give each branch a name. These names are the hints that we want to expose, and will be translated to an integer version to be used by *behaviour controllers*, as discussed in 11.2. We show the code in charge of generating these hints below:

```
/**
 * Starts a branch in a selector
 */
int LuaBTInterface::StartSelectorBranch( lua_State* L )
{
	ASSERT_STR( !s_nodeStack.empty(), L"No node was found in the stack!" );
	ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );

	// Get the top of the stack
	ASSERTS_ONLY( ThesisAbstractNode* pNode = s_nodeStack.top(); );
	ASSERT_STR( pNode->GetType() == behaviour::NT_SELECTOR,
			L"No selector was read before this branch!" );

	s_hintStack.push( std::string() );
	if ( lua_gettop( L ) == 1 )
	{
		ASSERT_STR( lua_isstring( L, 1 ), L"Invalid hint ID" );
		std::string& branchHint = s_hintStack.top();
		branchHint = lua_tostring( L, 1 );
	}

	return 0;
}

/**
 * Ends a branch in a selector
 */
int LuaBTInterface::EndSelectorBranch( lua_State* L )
{
	ASSERT_STR( !s_nodeStack.empty(), L"No node was found in the stack!" );
	ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );

	// Get the top of the stack
	ThesisAbstractNode* pNode = s_nodeStack.top();
	ASSERT_STR( pNode->GetType() == behaviour::NT_SELECTOR,
			L"No selector was read before this branch!" );

```

```
38          ASSERT_STR( !s_hintStack.empty(),
39                      L"No hint is available to close this branch!" );
40          const std::string& branchHint = s_hintStack.top();
41          if ( branchHint.size() > 0 )
42          {
43                behaviour::Selector< ThesisAIInstance >* pSelector =
44                        static_cast< behaviour::Selector<
45                                              ThesisAIInstance >* >( pNode );
46                pSelector->SetHintIDForChild( pSelector->GetChildCount() - 1,
47                                              hints::GetIDFromString( branchHint ) );
48          }
49          s_hintStack.pop();
50
51          return 0;
52    }
```

Branches are built in two steps (lines 4, 28). During the *start* command we push the name of our branch to a hint stack (lines 14-20). Naming a branch is not required (line 15); if we do not provide a name, we are simply not exposing that branch to higher levels. Names are used later in the *end* command. Note we always add a string to the hint stack (line 14); if the name was not an empty string, that is, if we supplied a valid name, we get the selector node from the top of the node stack, build a hint id from the name we provided (line 47) and assign this id to the appropriate child of the node (line 46).

The method we use to convert names to hint IDs is pretty simple, and is listed next:

```
1   HintID GetIDFromString( const std::string& hintStr )
2   {
3          int iLength = hintStr.size();
4          HintID id = 0;
5          for ( int i = 0; i < iLength; ++i )
6          {
7                HintID uiValue = ( ( HintID ) hintStr[ i ] ) * 1000;
8                id = ( id + uiValue ) / 2;
9          }
10
11         return id;
12  }
```

12.3.3.2. Creating a high-level tree

Once we have defined our high-level tree, we can start building a high-level one. To continue with our example, let us say we want to create a kamikaze AI, just as we did in 7.4.2.

High-level trees are always built on top of a base tree. Recalling what we studied in 11.3, high-level trees cannot use *actions* directly. Instead, they make use of *hint* nodes, which allow them to modify the normal execution flow of the underlying

tree. In our example, we exposed three hints in our base tree (COVER, ATTACK and CHASE), so these are the hints our new tree will be able to use.

So, let us build this new tree. We show the final result in Figure 52.

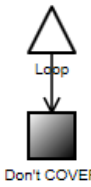


Figure 52. High-level tree that will make our AI be a kamikaze

The tree is very simple, which is what we were looking for. It simply has an unconditional loop as its root, which sends a 'DO NOT COVER' hint indefinitely.

Building a high-level tree from Lua is simple, as it uses exactly the same commands, but replaces our *action* commands with *hint* commands. This way, our tree would be defined using the following script:

```
startBehaviour()
	startLoop()
		addHint( "COVER", "Negative" )
	endNode()
endBehaviour()
```

The *addHint* command receives two parameters: the first one is the name of the hint we are going to send, which will, be converted internally to a hint id using the function previously shown; the second one indicate what type of hint we want to send, that is, which state the hint will be in. Three different states can be used as this second parameter; these are "Negative", "Positive" and "Neutral". The code handling hint commands is as follows:

```
int LuaBTInterface::AddHintNode( lua_State* L )
{
	ASSERT_STR( s_pCurrentInstance != NULL, L"No valid AI was specified!" );

	ASSERT_STR( lua_gettop( L ) == 2,
			L"Invalid number of parameters for addHint" );
	ASSERT_STR( lua_isstring( L, 1 ) && lua_isstring( L, 2 ),
			L"Invalid parameter type for addHint" );
	std::string hintName = lua_tostring( L, 1 );
	std::string hintStateStr = lua_tostring( L, 2 );
	thesis::hints::HintID hintId = thesis::hints::GetIDFromString( hintName
);
	thesis::hints::EState hintState =
			thesis::hints::GetStateFromString( hintStateStr );
```

```
15
16          thesis::behaviour::Hint< ThesisAIInstance >* pHint =
17                      new thesis::behaviour::Hint< ThesisAIInstance >();
18          pHint->Initialise( s_pCurrentInstance, s_pCurrentTree );
19          pHint->SetHintId( hintId );
20          pHint->SetHintState( hintState );
21          s_nodeStack.push( pHint );
22          EndNode( L );
23
24          return 0;
25  }
```

After performing some tests (lines 3-8), we get the parameters passed to the command, converting them to the appropriate types (lines 9-14). Once the parameters are read, we create a new instance of the node (lines 16-20), push the node onto the stack (line 21) and finalise it (line 22), as we are using a single command to create the node.

What our node, which is sending a 'do not cover' hint is doing, is effectively reordering the branches in our base tree as shown in Figure 53.

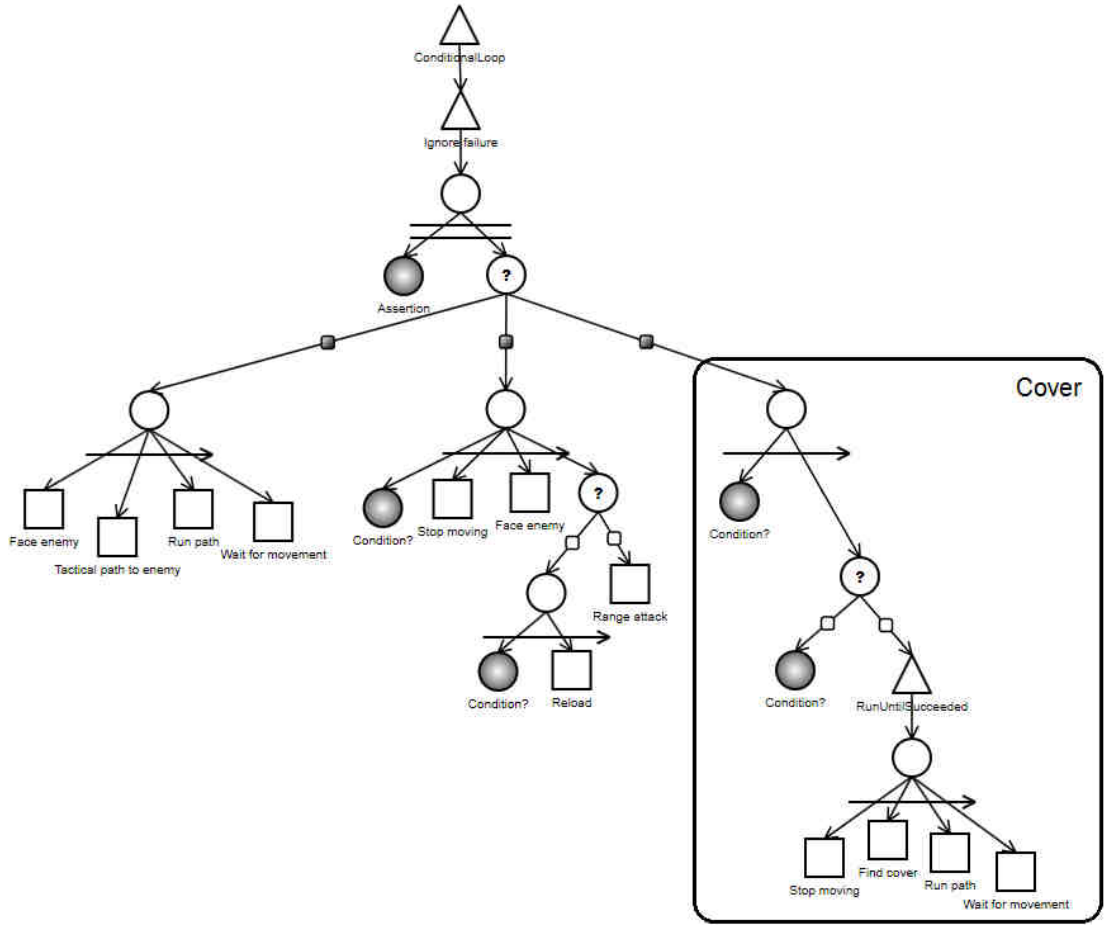


Figure 53. After receiving a 'do not cover' hint, our tree moves its cover branch to the last, and least priority, position

Let us examine a slightly more difficult example. In this case we want to model a coward AI that always tries to cover. The base tree remains the same and we only have to build a high-level tree such as the one shown in Figure 54, which is pretty similar to the one presented in the previous example, retaining its simplicity.

Figure 54. High-level tree that will make our AI be a coward

The tree is sending the same hint, "COVER", but in this case it is sending it with a positive value. The question here is: how does the hint work, if it is already the most important branch in the original tree? Inspecting our base tree, our "cover" branch has a precondition, which is checking whether the AI's health level is below a certain value; this makes the AI ignore our hint if it is healthy.

In order to solve this problem we must recall the *hint conditions*, which were introduced in 10.4. As we said then, base trees require a good design that allows higher-level logic to concentrate only on creating new behaviours, and not bother with low-level details. If we modify the precondition in the base tree, so the AI can decide to cover, **either** if its health is low **or** it is receiving a "COVER" hint, we will have solved the problem. The final precondition used by the branch is shown in Figure 55.

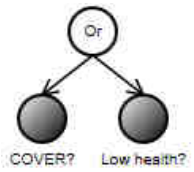


Figure 55. Precondition used by our "cover" branch

12.4. Setting behaviours

Once we have created our scripts to build our trees, we need a way to set them to the appropriate AI in the game. This can be done either during the creation of a level, or dynamically, while the game is running.

12.4.1. Level creation

Half-Life's SDK includes its own level editor, *Hammer*, which is the same one the developers used to create the game. The editor uses text files, with extension .fgd, to configure the entities that will be available during level edition.

In order to be able to add our own entities (*npc_thesis*) to a level, we have to create our own FGD, which looks like this:

```
1  @BaseClass base(Angles) studio("models/police.mdl")= npc_thesis :
2              "Thesis model entity."
3  [
4  BaseBehaviour(string) : "Base behaviour" : "" : "Base behaviour tree to use"
5  HighLevelBehaviour(string) : "High-level behaviour" : "" :
6              "High-level behaviour tree to use"
7
8  input PushBehaviour(string) : "Pushes a new behaviour."
9  input ClearBehaviours(void) : "Clears all the behaviours."
10 ]
```

With this file, we are letting Hammer know there is an entity called *npc_thesis* (line 1), which has two string properties that can be edited (lines 4-6), which are paths to the Lua files that define our base and high-level trees. We have decided to only use two levels as it will be enough for the examples presented in this thesis, but this could be easily modified later. The entity also exposes two methods, *PushBehaviour* and *ClearBehaviours*, which allow other entities to send messages to our agents and change their behaviours, as we will see later in Chapter 14.

Once our new FGD is loaded into *Hammer*, our agents will be ready to be used in new maps, as shown in Figure 56.

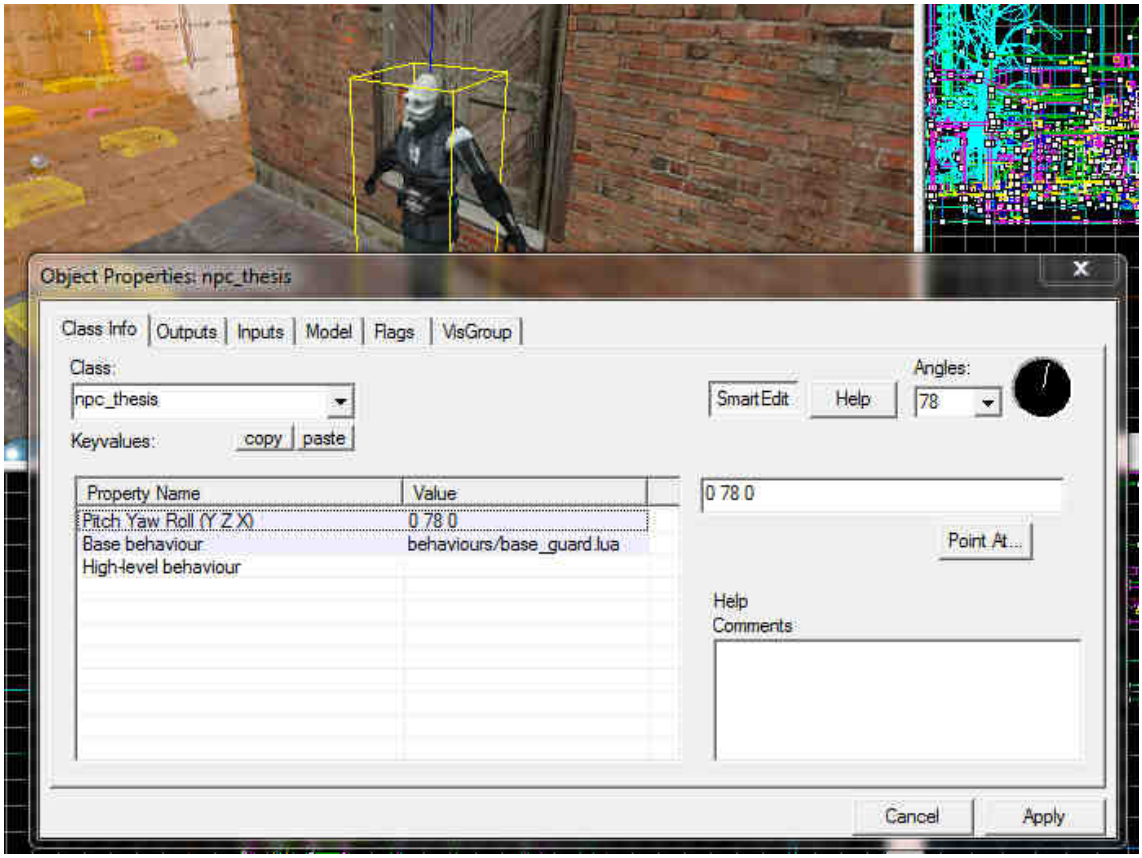


Figure 56. Adding an npc_thesis to a level in Hammer

We also have to map our class' member variables and methods to the properties and inputs we have defined. This is done in the entity's source file, using some specific macros. We show the code we use for our entities below:

```
BEGIN_DATADESC( CNPC_Thesis )

DEFINE_INPUTFUNC( FIELD_STRING, "PushBehaviour", PushBehaviourInput ),
DEFINE_INPUTFUNC( FIELD_VOID, "ClearBehaviours", \
                  ClearBehavioursInput ),
DEFINE_KEYFIELD( m_baseTreePath, FIELD_STRING, "BaseBehaviour" ),
DEFINE_KEYFIELD( m_highLevelTreePath, FIELD_STRING, \
                 "HighLevelBehaviour" ),

END_DATADESC()
```

Finally, we need to make our entities use these values. To do so, we have added a method that is called from the NPC's spawn function:

```
void CNPC_Thesis::SetInitialBehaviour()
{
       // Get the game dir
       char szGameDir[ 256 ];
       engine->GetGameDir( szGameDir, sizeof( szGameDir ) );

       // Set a base tree, if it's available
```

```
8         std::string treePath = m_baseTreePath.ToCStr();
9         if ( treePath.length() > 0 )
10        {
11                char szBuffer[ 512 ];
12                Q_snprintf( szBuffer, sizeof( szBuffer ), "%s/%s", szGameDir,
13                            treePath.c_str() );
14                Q_FixSlashes( szBuffer );
15                if ( !PushBehaviour( szBuffer ) )
16                {
17                        DevMsg( "Invalid base tree: [%s]", treePath.c_str() );
18                }
19                else
20                {
21                        // Set a high-level tree if it's available
22                        treePath = m_highLevelTreePath.ToCStr();
23                        if ( treePath.length() > 0 )
24                        {
25                                Q_snprintf( szBuffer, sizeof( szBuffer ), "%s/%s",
26                                            szGameDir, treePath.c_str() );
27                                Q_FixSlashes( szBuffer );
28                                PushBehaviour( szBuffer );
29                                if ( !PushBehaviour( szBuffer ) )
30                                {
31                                        DevMsg( "Invalid high-level tree: [%s]",
32                                                treePath.c_str() );
33                                }
34                        }
35                }
36        }
37  }
```

All our paths are relative to the game's initial path, so we get it first (lines 4-5). If a base tree was defined (lines 8-9), we simply generate a proper full path (lines 11-14) and push the behaviour to our AI's stack (line 15). We follow a pretty similar process to set a high-level tree if the base tree was set correctly (lines 19-36).

12.4.2. Dynamic changes

Half-Life has a powerful in-game console players can open at any moment to run commands, such as cheats. For example, typing "god" will make the player invincible.

We have added two commands, so we can control our AIs from the console:

- **pushBehaviour** receives the ID of the agent we want to use, as well as the full path to a behaviour tree we want to push to our entity's stack.
- **clearBehaviours** receives only an ID, and it will clear all the behaviours used by that AI.

We show the code used by *pushBehaviour* as an example:

```
1   static void PushBehaviour( const CCommand &args )
```

```
2  {
3         if ( args.ArgC() == 3 )
4         {
5               int id = atoi( args[ 1 ] );
6               ThesisAIInstance* pAI = ThesisAIManager::Ref().GetAI( id );
7               if ( pAI == NULL )
8               {
9                     DevMsg( "Can't find AI [%i]", id );
10              }
11              else
12              {
13                    DevMsg( "AI [%i]: pushing behaviour defined in [%s]",
14                           id, args[ 2 ] );
15                    ThesisAIManager::Ref().GetLuaManager().SetGlobalNumber(
16                lua::LuaBTInterface::CURRENT_AI_GLOBAL_VARIABLE_NAME, id );
17                    ThesisAIManager::Ref().GetLuaManager().RunFile(
18                                                              args[ 2 ] );
19              }
20        }
21        else
22        {
23              DevMsg( "Invalid arguments for pushBehaviour" );
24        }
25 }
26
27 ConCommand pushBehaviour_function( "pushBehaviour", PushBehaviour,
28                 "Pushes a behaviour for an AI (id, lua file)", FCVAR_CHEAT );
```

In order to define a command, we have to create a static function that receives a *CCommand* as its first and only argument (line 1). This object contains information about the parameters that were passed to the command in the console, so we check we are receiving two (line 3, args[0] contains the name of the command), and use it to set the behaviour to the appropriate instance (lines 5-19). We also need to register the command, which is done creating a global *ConCommand*, as shown in lines 27-28.

Chapter 13. VISUAL EDITING

In previous chapters, we introduced our new technique, Hinted-execution Behaviour Trees, and presented how the system works, and how we have implemented it. We also made some notes about how the system would be greatly improved, both in terms of productivity and usability.

Also, the use of an editor is crucial if we want to achieve our objective of allowing a multidisciplinary team share responsibilities in the development and enhancement of our AI.

Because of this, we have developed our own HeBT editor, as a fundamental part of this work. In this chapter we will present this tool and show how it is integrated with the rest of the system.

13.1. Functionality

HeBT Editor is a complete tool that manages Hinted-execution Behaviour Trees from their creation process to their deployment and testing. It has been developed using C#/WPF, so it requires .NET, and offers the following functionality:

- An XML-based node library, which allow new nodes to be added to the tool easily.
- A visual way to create and modify behaviour trees, which are shown in a graph-like representation. Nodes can be dragged-and-dropped from the library and can be connected with just a couple of clicks.
- Complex conditions can be created, in the form of condition trees, using the same graph-like, drag-and-drop enabled interface.
- High-level BTs can be created based on existing BTs. The node library is automatically updated, so actions are replaced with hint nodes, using those hints exposed by the base tree.

- Trees can be saved and loaded as XML files, which makes them easy to understand in a plain text editor, and can also be exported to a LUA format that is accepted by our game prototype (presented in Chapter 12).
- The editor is capable of establishing a connection with the game prototype, and changes the behaviours of the game's entities dynamically. This allows modifications to be tested quickly.
- Debugging capabilities are also offered, when the editor is connected to the prototype. The base tree is shown, and the nodes that are running are highlighted in real time, so the effect of the hints can be easily seen.
- The editor uses a command system, so actions can be undone and redone, improving usability.

13.2. Main window

The main window is simple (Figure 57), showing a toolbar with three icons that provide quick access to the game-communication functions (connect to game, run and debug tree), as well as a main menu.

Figure 57. Main window of HeBT Editor

The menu provides access to the basic functions of the editor (such as creating or saving trees), the undo system and also to the communication options shown in the toolbar. Figure 58 shows the menu items.

Figure 58. Options present in the main menu

13.3. Architecture

HeBT Editor uses a docking layout system similar to that present in Visual Studio, and uses *AvalonDock* (Marinucci, 2010) to do this. Taking advantage of the functionality provided by this library, we have been able to implement a MDI interface, where multiple documents, each of which represents a behaviour tree, can be opened simultaneously.

A document is the graphical representation of our logical model. We have separated both models and built a behaviour tree system based on *composite* and *visitor* design patterns. The latter is especially important for our editor, as we will see later in this chapter, we have implemented most of the functionality as different visitors that operate on our trees, simplifying the logic of the nodes and trees, and allowing us to have an extensible application.

13.3.1. Documents

Trees in our editor are represented as a non-connected graph, where each node is an instance of a class present in a *node library*. This basic structure is illustrated in Figure 59.

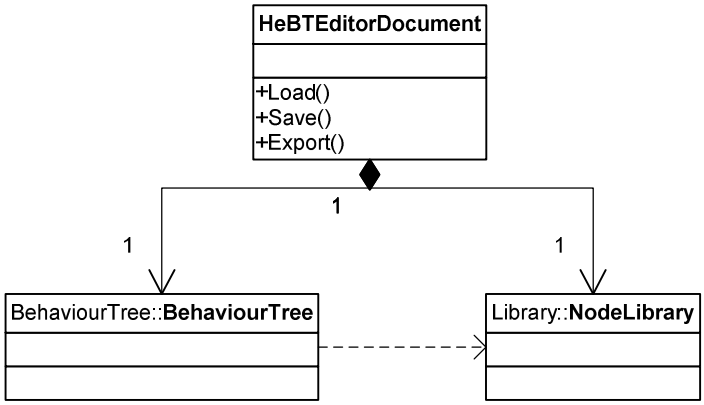


Figure 59. Structure of a document

Documents also expose some basic functionality, like loading, saving, and exporting documents, and are the interface the tool uses to communicate with our model.

For each document, the editor will show a new tab divided into two different areas. The main one shows the behaviour tree that is being created, while the second area shows the node library that is available for this particular tree. The library shows different elements depending on the type of tree. New nodes can be added to the tree just by drag-and-dropping them onto the main area, just as we illustrate in Figure 60.

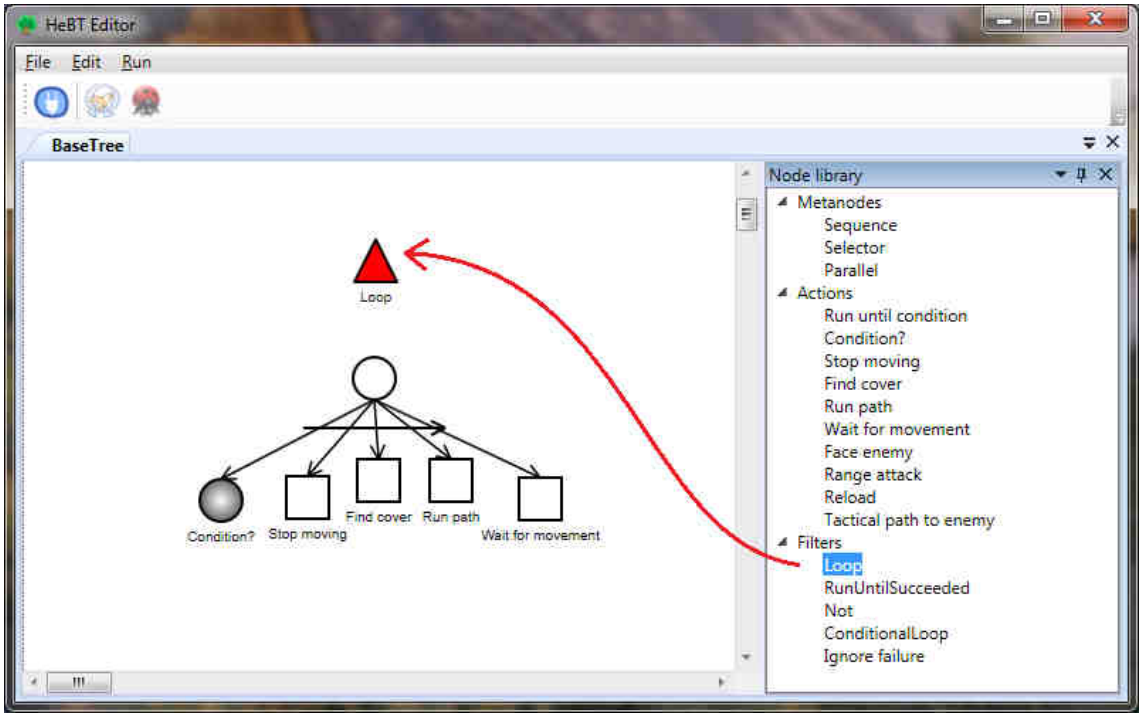


Figure 60. New nodes can be added to a BT just by drag-and-dropping them from the library

In a similar fashion, we can add new links between nodes just by clicking on the source node and on the destination one (in that order). A node or link can be removed by selecting them and pressing "Del" on the keyboard.

13.3.2. Behaviour trees

Behaviour trees are handled by the editor in a similar way to that used in the library presented in Chapter 9. In this case, the relations between the different classes are shown in Figure 61.

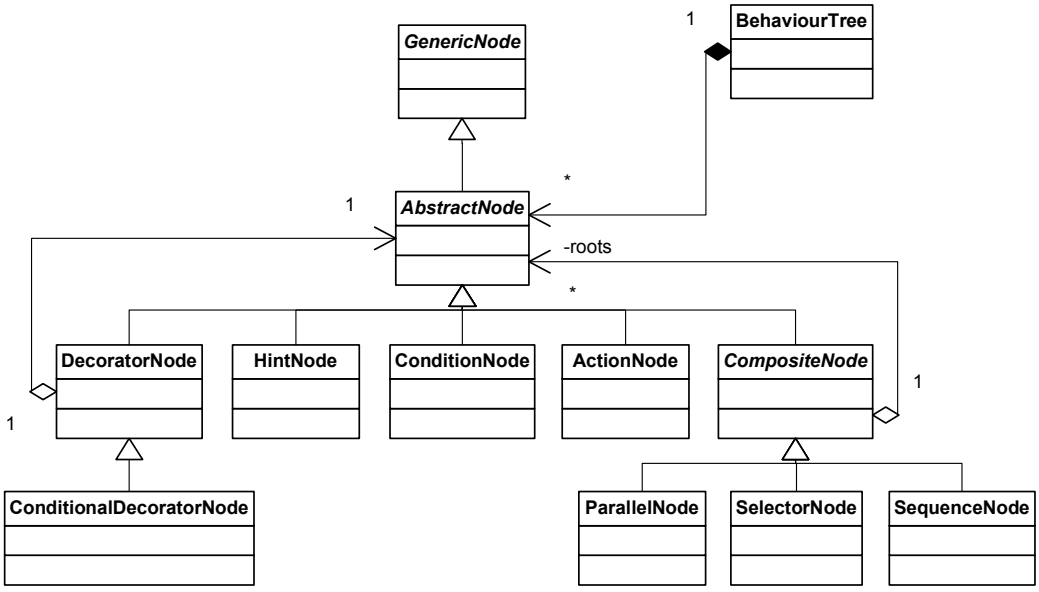


Figure 61. Structure of a behaviour tree in the editor

We must note that we use a *GenericNode* that contains the basic information needed for a node to be used in our graph editor. We have separated this from our *AbstractNodes* because we will reuse this component as an editor of *condition trees*, which will be introduced further on in this chapter. Each of these nodes stores some useful information such as which their parent nodes are, what their offsets are related to those parents, which are used for drawing, or whether they are selected or started, as well as the identifier of the node class they were created from.

A BT in the editor can have multiple root nodes. This is a consequence of the edition mode we have chosen, where we can add nodes by dragging them from the editor, but they are unconnected when they are dropped.

However, we decided not to bloat nodes with logic other than that which keeps the relations among nodes. Instead, we use various different visitors to perform operations on trees.

13.3.2.1. Visitors

When designing HeBT Editor, we found that almost certainly we were going to be adding new functionality to the tool as a result of our research. According to its definition, "a *visitor pattern* is a way of separating an algorithm from an object structure it operates on" (Wikipedia, 2010). Visitors can run some specific logic for each node on the tree, depending on their types, automatically, so we decided that was exactly what we needed.

The version of the editor presented in this document uses eight different visitors that operate on BTs, plus five additional ones that are used by condition trees (which will be presented later in this chapter). These visitors are:

- **CheckVisitor** is in charge of checking that a tree can be used by our BT library. A tree is valid only if it has a unique root, all its composite nodes have, at least, a child, and all the nodes that use conditions have been assigned a valid one.

- **DrawVisitor** is used to draw trees in the editor. It takes into account not only the type of node in order to produce a valid graphical representation of it, but also its state, so it is able to highlight nodes that are being run or are selected.

- **HintParserVisitor** is used to build a fill in a list with the identifiers of the hints used in the tree.

- **LinkSelectVisitor** and **SelectVisitor** receive a point on screen and are able to select a node or a link, based on that position.

- **LuaExporterVisitor** exports a tree to a LUA format that can be read by the game.

- **LuaNetExporterVisitor** and **StateUpdaterVisitor** use the communication capabilities of the editor, which will be studied in 13.5, to send trees through this communication channel and receive updates about the state of the nodes of a tree.

Once a visitor is implemented, using it is very simple. We just have to create a new instance of the editor and call *Accept* on the root of our tree, passing in the visitor. The following snippet shows how this works in code:

```
HintParserVisitor visitor = new HintParserVisitor();
foreach ( AbstractNode root in behaviourTree.Roots )
{
    root.Accept( visitor );
}
// At this point, visitor.ListOfHints contains all the hints used in
// behaviourTree
```

13.3.2.2. Saving a tree as XML

We use .NET's capabilities to serialise and deserialise objects to XML to save our trees and load them in the editor. However, we have chosen not to use this XML formats in the game, and use a LUA file instead, because the latter can be parsed and executed directly by a LUA interpreter, which we have included to our library.

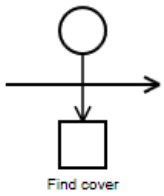


Figure 62. Simple tree we will serialise

An example of the output produced by our serialisation process, and using the simple tree shown in Figure 62, can be seen below:

```
<?xml version="1.0" encoding="utf-8"?>
<BehaviourTree xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Roots>
    <AbstractNode xsi:type="SequenceNode">
      <Offset>
        <X>178</X>
        <Y>53.043124237060539</Y>
      </Offset>
      <Size>
        <Width>30</Width>
        <Height>30</Height>
      </Size>
      <ItemId>1</ItemId>
      <Children>
```

```
16          <AbstractNode xsi:type="ActionNode">
17            <Offset>
18              <X>0</X>
19              <Y>74</Y>
20            </Offset>
21            <Size>
22              <Width>30</Width>
23              <Height>30</Height>
24            </Size>
25            <ItemId>102</ItemId>
26          </AbstractNode>
27        </Children>
28      </AbstractNode>
29    </Roots>
30  </BehaviourTree>
```

The process is quite simple: we serialise all the roots of the tree (line 4). In this instance we only have one root, which is our sequence (line 5). The sequence has one child, an action (line 16). We know which action we are serialising because of its itemId (line 25), which corresponds to one of items in the node library we were using.

13.3.3. Node libraries

A node library is a list of node classes that can be used in a behaviour tree. We needed the editor to be extensible, so new types of nodes could be used by it without any modifications. Above all, we needed this to add new actions, as they are the ones that change between different games.

As we said before, nodes in our BTs are just instances of the items in our library. These items contain information that is shared among those instances. The data that defines a library node is:

- A **unique numeric identifier**, which will be used by node instances to refer to their parent node class.
- A **class name**, which will be used to create an instance of the node in the editor. The class name must match one of those presented in 13.3.2, as we are using C#'s reflection capabilities to instantiate the new objects.
- A **label**, which will be used by the *draw visitor*.
- The **LUA commands** a node will be translated to, when we export the tree. We can add two strings to define these commands: one of them will be used

when we start exporting the node; the other, after we have finished the process. The exportation process was explained in detail in 12.3.

- A flag to indicate whether the node will be **available** in all levels, or just in the lowest one. This is useful if we want to avoid some nodes from being used in higher-levels of logic. For example, we could hide *parallel* nodes, as commented in 11.3.

- A list of **child nodes**. We have designed our library as a hierarchy, so it is easier to classify nodes in groups. It is important to note that only leaf nodes can be instantiated in an actual tree.

We use an XML file to define a library. We show an example library:

```
<?xml version="1.0" encoding="utf-8"?>
<NodeLibrary xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Items>
    <NodeLibraryItem>
              <Label>Metanodes</Label>
              <AvailableInAllLevels>true</AvailableInAllLevels>
              <Children>
                    <NodeLibraryItem>
                       <ClassName>
                        HeBTEditor.Logic.BehaviourTree.SequenceNode
                       </ClassName>
                       <Label>Sequence</Label>
                       <StartExportString>startSequence()</StartExportString>
                       <EndExportString>endNode()</EndExportString>
                       <Id>1</Id>
                       <AvailableInAllLevels>true</AvailableInAllLevels>
                       <Children />
                    </NodeLibraryItem>
              </Children>
    </NodeLibraryItem>
    <NodeLibraryItem>
              <Label>Actions</Label>
              <AvailableInAllLevels>true</AvailableInAllLevels>
              <Children>
                    <NodeLibraryItem>
                       <ClassName>
                       HeBTEditor.Logic.BehaviourTree.RunUntilConditionNode
                       </ClassName>
                       <Label>Run until condition</Label>
                       <StartExportString>
                         startRunUntilConditionNode()
                       </StartExportString>
                       <EndExportString>endNode()</EndExportString>
                       <Id>99</Id>
                       <AvailableInAllLevels>true</AvailableInAllLevels>
                       <Children />
                    </NodeLibraryItem>
              </Children>
    </NodeLibraryItem>
       <NodeLibraryItem>
              <Label>Filters</Label>
              <AvailableInAllLevels>true</AvailableInAllLevels>
```

```
44              <Children>
45                    <NodeLibraryItem>
46                      <ClassName>
47                       HeBTEditor.Logic.BehaviourTree.DecoratorNode
48                      </ClassName>
49                      <Label>Loop</Label>
50                      <StartExportString>startLoop()</StartExportString>
51                      <EndExportString>endNode()</EndExportString>
52                      <Id>200</Id>
53                      <AvailableInAllLevels>true</AvailableInAllLevels>
54                      <Children />
55                    </NodeLibraryItem>
56              </Children>
57    </NodeLibraryItem>
58  </Items>
59 </NodeLibrary>
```

This file is defining three categories (line 5 – metanodes, line 22 – actions and line 41 – filters), each containing a node. Figure 63 shows the library as it would be seen in the editor.

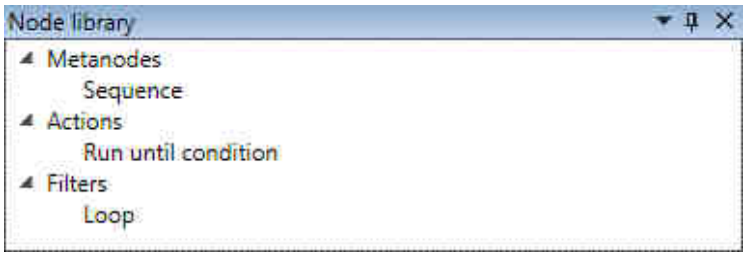


Figure 63. Example node library

We can also add properties to the nodes we define in our libraries; these will be used so users can specify certain parameters, and will allow us to create more complex nodes. An example would be a *timed loop filter*, where we can set the time the loop must be run for. We would define the property as follows:

```
1  <NodeLibraryItem>
2      <ClassName>HeBTEditor.Logic.BehaviourTree.DecoratorNode</ClassName
3      <Label>Timed loop</Label>
4      <StartExportString>startTimedLoop( {0} )</StartExportString>
5      <EndExportString>endNode()</EndExportString>
6      <Id>205</Id>
7      <AvailableInAllLevels>true</AvailableInAllLevels>
8      <Children />
9      <Properties>
10         <NodePropertyData>
11             <Key>Duration</Key>
12             <Value>1.0</Value>
13             <Type>System.Single</Type>
14         </NodePropertyData>
15     </Properties>
16 </NodeLibraryItem>
```

Each property we define must have a **key**, a default **value** and a **type** (C# types) that will be used to allow the editor to check if the values received for that

parameter are correct. Properties can be set for a node just by double-clicking it in the editor (see Figure 64).

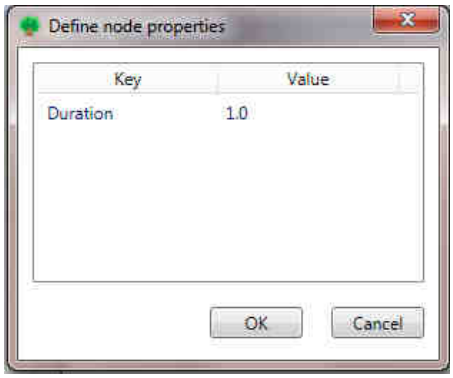


Figure 64. Some nodes expect some parameters (properties) that can be defined via the editor

13.3.4. Condition trees

Conditions trees were already introduced in 9.4.1.1., and, basically, they allow us to generate complex conditions. Our tool can create new condition trees using an editor that is very similar to the one used to build behaviour trees. The hierarchy of classes we use is shown in Figure 65.

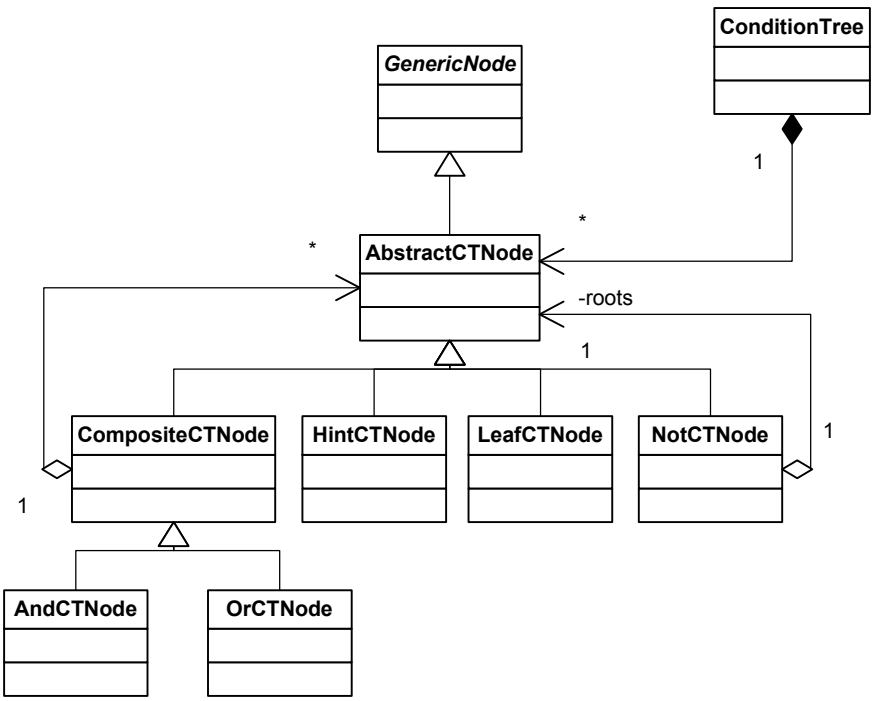


Figure 65. Structure of a condition tree in the editor

Condition trees are created in the same way as normal behaviour trees, just by drag-and-dropping new nodes from a library of them, and connecting our instances appropriately. Figure 66 shows our editor of conditions.

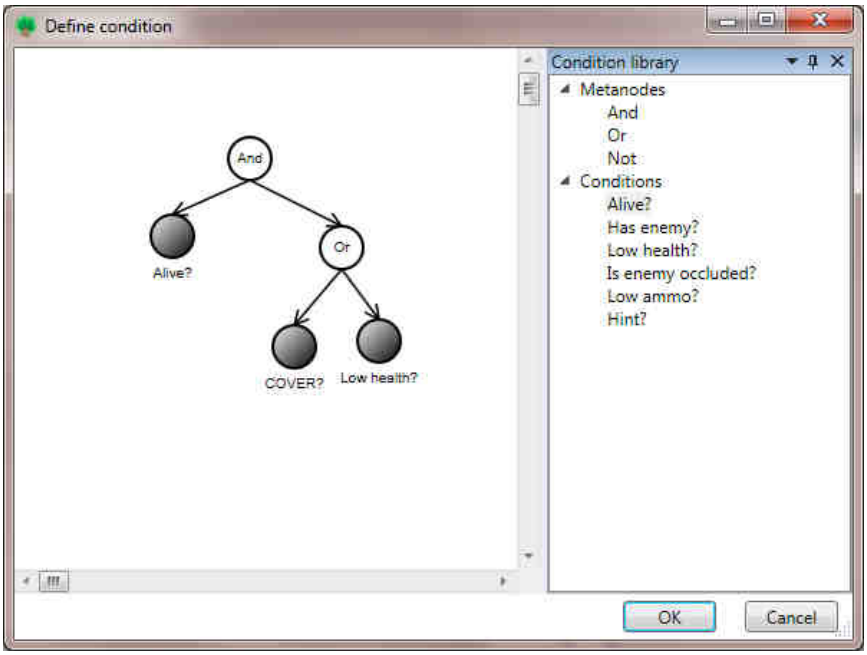


Figure 66. Interface of the condition tree editor

We use condition libraries that follow the same structure as our node libraries, which we defined in the previous section.

13.4. Creating a new tree

The editor allows users to create two different types of behaviour trees: base BTs and high-level ones; differences between these types were presented in Chapter 11. When we try to create a new tree, the tool will ask for the type of BT we want to use, as shown in Figure 67.

Figure 67. The editor allows us to create low-level or high-level trees

13.4.1. Low-level trees

Low-level trees are the ones AIs use to communicate with the game, using *action nodes*. They cannot send hints to other trees, but they can expose them, so higher-level ones can be built to improve them.

Recalling what we studied in 11.2, hints are exposed by naming *selectors*' branches or using *hint conditions.* In the editor, selectors are displayed graphically as shown in Figure 68.

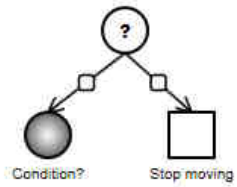


Figure 68. A simple selector, as shown by the editor

Note the two little squares in the middle of each link: the squares represent the hint the branch can receive. If no hint was defined, the square will be white. Double-clicking the square, we will access a new dialog where we can name the branch, thus exposing a new hint (Figure 69).

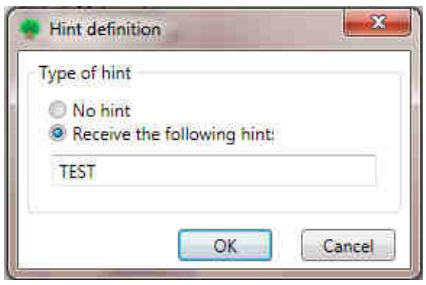


Figure 69. Naming a selector's branch will expose a new hint to higher-levels

After a hint is exposed, the little square will be coloured in black, as illustrated in Figure 70.

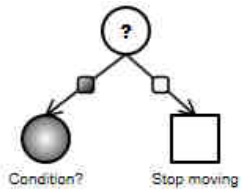


Figure 70. Branches that have exposed a new hint are marked in a different colour in the editor

As for conditions, we can expose new hints just by using a *hint condition* in a condition tree. As shown in Figure 71, the hint used by the condition is undefined by default.

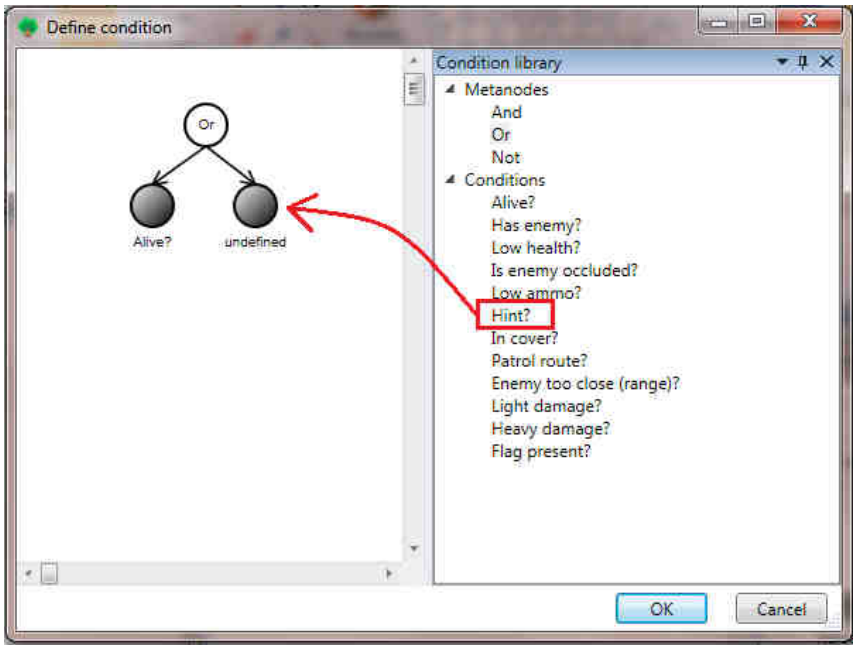


Figure 71. Exposing new hints using hint conditions is as simple as adding one of these nodes to a condition tree and defining the hint we want to check

Again, double-clicking on it will open a new window where we will be able to define which hint we want the condition to check (Figure 72).

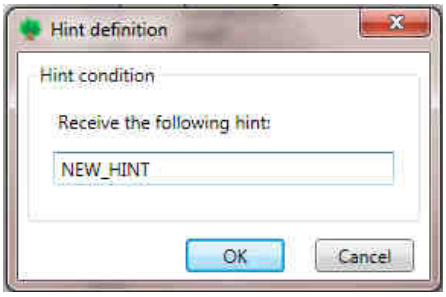


Figure 72. We must define which hint a condition hint will check

Once the hint has been chosen, the editor will show it correctly, as illustrated in Figure 73.

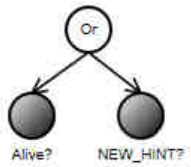


Figure 73. Once a hint is defined, the editor will reflect the change

13.4.2. High-level trees

High-level trees are built on top of a base tree, which will be used to generate the node library the new tree will use. This library will exclude the nodes that were marked as "not available in all levels" in the XML file, and will contain hint nodes for each of the hints exported in the low-level tree. For instance, if we create a high-level tree based on the base one shown in the previous section, we would get a library such as the one shown in Figure 74.

Figure 74. The node library our high-level tree would use. The hints exposed in the base tree are automatically added to the library

Hint nodes are treated in a special way in the editor, so we can define which hint state a node will be sending. Figure 75 shows the dialog the tool will show after double-clicking on one of these nodes.

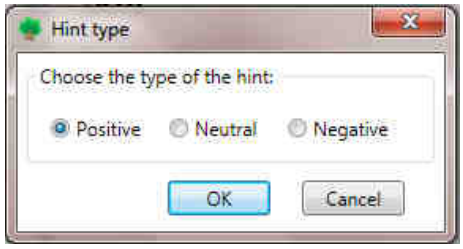


Figure 75. Hint nodes can send different states of a hint

13.5. Game communication

As stated earlier, HeBT editor is capable of establishing a communication with our Half-Life 2 prototype. This is done via *winsock*.

Our library defines a *Server* class which is always listening for connections at port 16777 (at the moment, it will only accept one, from the editor). Once a connection is made, it will notify a list of *receivers*, which must register with the server in order to get data. Receivers must implement an *IReceiver* interface. This structure is shown in Figure 76.

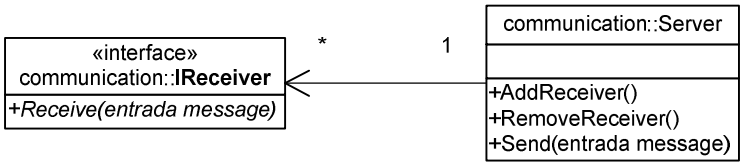


Figure 76. Structure of our communication server

We have also implemented a custom receiver, *ThesisCommunication*, which defines all the functionality we need to be able to set and debug behaviours from the editor. The object is registered with the server when the game is run.

ThesisCommunication works as a simple FSM with three states: *Idle*, *ReceivingTree* and *Debugging*. We show its *Receive* method below:

```
1  void ThesisCommunication::Receive( const std::string& message )
2  {
3         std::vector< std::string > tokens = Tokenize( message, DELIMITERS );
4         if ( m_eState == EState_Idle )
5         {
6               if ( tokens[ 0 ] == "SET_TREE" )
7               {
8                     ASSERT_STR( tokens.size() == 2,
9                                 L"Invalid number of parameters (id expected)" );
10                    int id = std::atoi( tokens[ 1 ].c_str() );
11                    ThesisAIManager::Ref().GetLuaManager().SetGlobalNumber(
12                lua::LuaBTInterface::CURRENT_AI_GLOBAL_VARIABLE_NAME, id );
```

```
13
14                      // Start receiving the tree
15                      m_eState = EState_Receiving_Tree;
16
17                      DevMsg( "Setting a new tree to AI [%i]\n", id );
18              }
19              else if ( tokens[ 0 ] == "CLEAR_BEHAVIOURS" )
20              {
21                      ASSERT_STR( tokens.size() == 2,
22                              L"Invalid number of parameters (id expected)" );
23                      int id = std::atoi( tokens[ 1 ].c_str() );
24
25                      //Get the AI and clear its behaviour
26                      ThesisAIInstance* pInstance =
27                          ThesisAIManager::Ref().GetAI(
28                             static_cast< ThesisAIInstance::UIDType >( id ) );
29                      if ( pInstance != NULL )
30                      {
31                              pInstance->ClearBehaviours();
32                      }
33
34                      DevMsg( "Clearing behaviours for AI [%i]\n", id );
35              }
36              else if ( tokens[ 0 ] == "DEBUG_TREE" )
37              {
38                      m_eState = EState_Debugging;
39              }
40              else
41              {
42                      // Just ignore any other message in this state
43              }
44      }
45      else if ( m_eState == EState_Debugging )
46      {
47              if ( tokens[ 0 ] == "STOP_DEBUGGING" )
48              {
49                      m_eState = EState_Idle;
50              }
51              else if ( tokens[ 0 ] == "REQUEST_TREE_STATE" )
52              {
53                      int id = std::atoi( tokens[ 1 ].c_str() );
54
55                      ThesisAIInstance* pInstance =
56                          ThesisAIManager::Ref().GetAI(
57                              static_cast< ThesisAIInstance::UIDType >( id ) );
58                      if ( ( pInstance != NULL )
59                           && ( pInstance->HasBehaviours() ) )
60                      {
61                              ASSERT_STR( tokens.size() == 2,
62                                  L"Invalid number of parameters (id expected)" );
63
64                              // First, send the hint states
65                              hints::HintAndStateVector info;
66                              pInstance->FillHintAndStateVector( info, -1 );
67
68                              hints::HintAndStateVector::const_iterator it =
69                                                              info.begin();
70                              hints::HintAndStateVector::const_iterator end =
71                                                              info.end();
72                              for ( ; it != end; ++it )
73                              {
74                                      const hints::HintAndState& hintAndState =
75                                                      *it;
76                                      char buffer[ 256 ];
77                                      sprintf_s( buffer, 256,
78                                                 "HINT_STATE %i %i",
79                                                  hintAndState.id,
80                                            static_cast< int >(
```

```
81                                              hintAndState.state ) );
82
83                                  ThesisAIManager::Ref().
84                                       GetCommunicationServer().Send( buffer );
85                              }
86
87                              // Now send the node states
88                              ThesisNetVisitor visitor;
89                              pInstance->Accept( &visitor, -1 );
90                              ThesisAIManager::Ref().
91                                   GetCommunicationServer().Send(
92                                                        "TREE_STATE_END" );
93                      }
94                      else
95                      {
96                            ThesisAIManager::Ref().
97                                       GetCommunicationServer().Send(
98                                              "TREE_STATE_STOPPED" );
99                            m_eState = EState_Idle;
100                     }
101             }
102             else
103             {
104                     DevMsg( "Error: invalid command received [%s]\n",
105                             message.c_str() );
106             }
107     }
108     else if ( m_eState == EState_Receiving_Tree )
109     {
110             // Receiving a tree
111             if ( tokens[ 0 ] == "END_TREE" )
112             {
113                     // End of the tree
114                     m_eState = EState_Idle;
115             }
116             else if ( !ThesisAIManager::Ref().GetLuaManager().
117                                     ExecuteCommand( message ) )
118             {
119                     DevMsg( "Error in command [%s] while setting a tree\n",
120                              message.c_str() );
121             }
122     }
123 }
```

We first tokenise the message we are receiving (line 3), so we can use commands properly. After that, we execute the logic of the active state:

- If we are **idle** (lines 4-44), we would could be receiving three different commands:
    - “SET_TREE instance_id” indicates we will start receiving lua commands to set an AI instance, switching to a new state (line 15).
    - “CLEAR_BEHAVIOURS instance_id” indicates we should clear the behaviour that is controlling the given instance.

- "DEBUG_TREE" indicates we want to debug a tree. This will put us in Debugging state.

- If we are **debugging a tree** (lines 45-107), we would be put in a state where we can receive update requests (REQUEST_TREE_STATE instance_id) (lines 51-100), which means we have to send the current state of the tree back. First, we will send the current states of the hints for this tree, as a list of "HINT_STATE hint_id hint_state" commands (lines 64-85), so the editor can show this information; then a list of nodes and whether they are started or not; the list is just a big string made of pairs (id state), where *id* is a numeric value indicating the order in which the node was read, and *state* is a boolean value, will follow (lines 87-92). The editor will send this command periodically, and will update the states of the nodes in the tree it is debugging accordingly, showing a tree where started nodes are highlighted in yellow (Figure 77). If for some reason our behaviour ends while it is being debugged, we would send a "TREE_STATE_STOPPED" command, and get back to the Idle state (lines 96-99). Finally, we can also receive a "STOP_DEBUGGING" command to indicate we do not want to debug the tree anymore, so we get back to Idle.

- If we are **receiving a tree** (lines 108-122), the editor is basically exporting a tree to Lua and sending the result through the socket; we will run the commands as we receive them (lines 116-117), mimicking the process shown in 12.4.2. When we receive an "END_TREE" command, we will get back to the Idle state (111-115).

This functionality is accessed from the main menu or the toolbar icons.

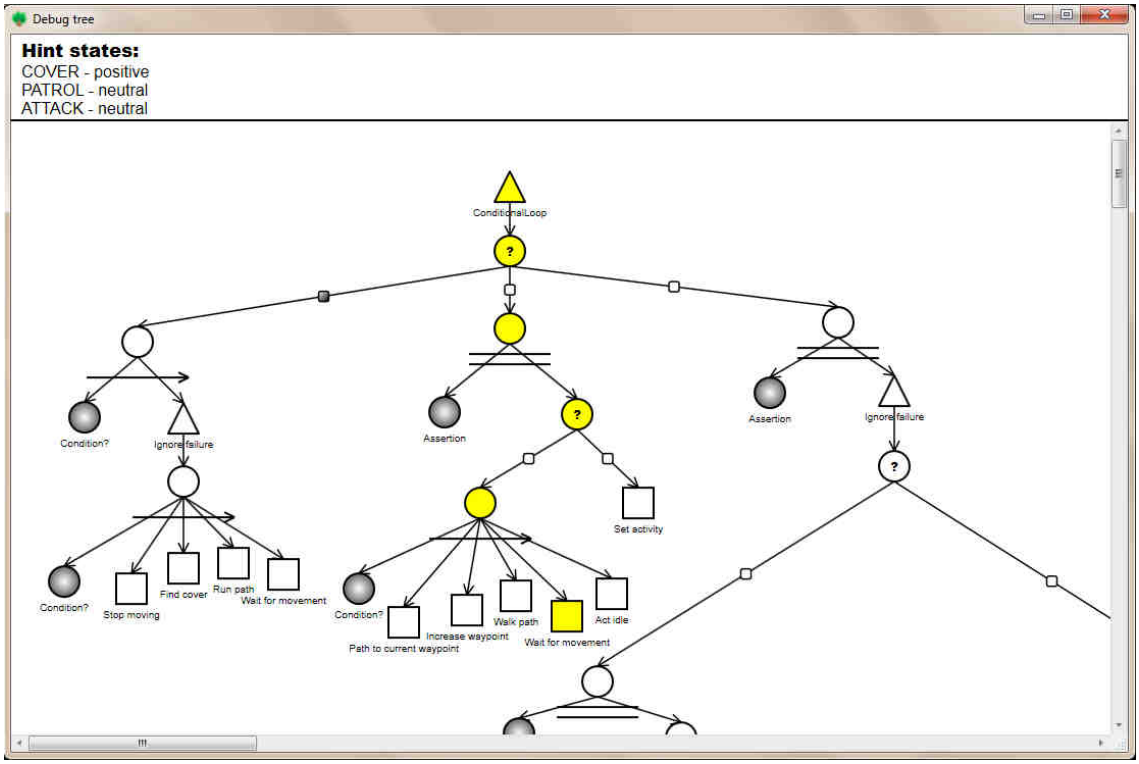


Figure 77. Debugging a tree in the editor

13.6. Future work

Although the editor developed in this thesis is fully functional, further work can be done to improve its usability and the functionality it provides. Among the possible extensions the tool would benefit from, we can suggest:

- A method to be able to check what conditions are being used by the corresponding nodes in our trees. At the moment, the only way to do this is by opening the condition tree editor (by double clicking a node), which makes it difficult to have an idea of what the tree is doing at a first glance.
- A way to store parts of the tree as sub-behaviours that can be reused in different trees. This would also allow trees to be easier to understand, as they could be edited modularly.
- Support for expanding and collapsing branches. A different type of view, explorer-like, could also be added.
- Support for copying and pasting or duplicating nodes or sub-trees.

- Extending the command system, so any action can be undone or redone, as, at the moment, condition or hint edition are not handled using commands.
- Enhanced debugging facilities. The current system only allows us to see which nodes are active, with a frequency of a second between updates. Also, if we are debugging a high-level tree, we would only be able to see how the underlying base tree is performing. To improve this, we could:
    - Allow the refresh rate to be modified dynamically.
    - Add a way to choose which tree of an HeBT we want to be looking at, or even being able to see more than one tree at a time.
    - Modify the debug drawing code, so we can actually see how branches are reordered when a selector is hinted.
    - Add a high-frequency mode, where all the state changes are stored in a file that could be examined later on. This would allow behaviours to be debugged off-line, and would offer more detailed information.
- Add the possibility of creating group behaviours, such as the ones studied in the previous section, using the editor.
- Find a better way to display all the trees that are part of the same HeBT at once. At the moment, BTs are created individually, without taking into account whether they belong to the same HeBT or not.

Chapter 14. WHY HEBTS ARE GOOD FOR AI PROGRAMMING: A PRACTICAL EXAMPLE

In Chapter 12 we studied how our new HeBT system can be applied to a real game, and developed a prototype using Valve's Half-Life 2 SDK. We also worked on some basic examples to show the internals of the system, and how to generate new behaviours creating simple high-level trees. However, these examples were too simple, and they did not really demonstrate the real power of our solution.

In this chapter we will present a new case, in which the game will require some complex additional logic; as we will show, it can be created as a high-level tree in an HeBT, avoiding the need to modify the base behaviour, which would add additional risks to the project.

14.1. Prototyping new ideas

In 7.4 we talked about different ways to modify a behaviour. Particularly, we studied the use of *personality traits* as a simple solution to obtain different responses from two AIs running the same logic. We also showed how adding a more complex logic was not trivial without changing the original tree. On the other hand, Hinted-execution Behaviour Trees allow us to generate this logic easily, just by using a high-level behaviour tree, which will run on top of our base one, guiding its normal execution towards what this new level is *suggesting* should be done.

In a real project, we are always subject to changes at any time, if these changes enhance the game experience. However, we cannot often work on new ideas as they would require too many resources, or may pose a big risk to the project. Although some examples were shown in Chapter 12, we have not built any complex high-level behaviour yet, so, in order to completely prove this point, we decided to implement a small gameplay feature in our prototype, completely controlled by our HeBTs.

14.2. Base behaviour

Working on a new behaviour requires that we have a base one working correctly; it will define the way AIs in our game respond to different situations and, in a real life project, it would have been thoroughly tested and optimised.

In our case, we will create a new Half-Life level and add some of our AIs to it. Let us say our design team have decided the game needs some guards that:

- Are able to patrol using a predefined route.
- Detect the player as an enemy when they enter their cone of vision.
- Attack the player once it is identified.
- Try to find a cover position to keep attacking from it, if the agent is damaged.
- Find a cover position to reload their weapons, it they are running low of ammunition.
- Run away if their health level is too low.

This behaviour would be represented by a complex BT, such as the one shown in Figure 78. As we will see, the root of our tree is a *conditional loop*, which is running the behaviour as long as the AI is alive. The loop is decorating our main *selector*, which is deciding whether to cover, patrol or attack.

Due to its size, we will study it by separating its different branches.

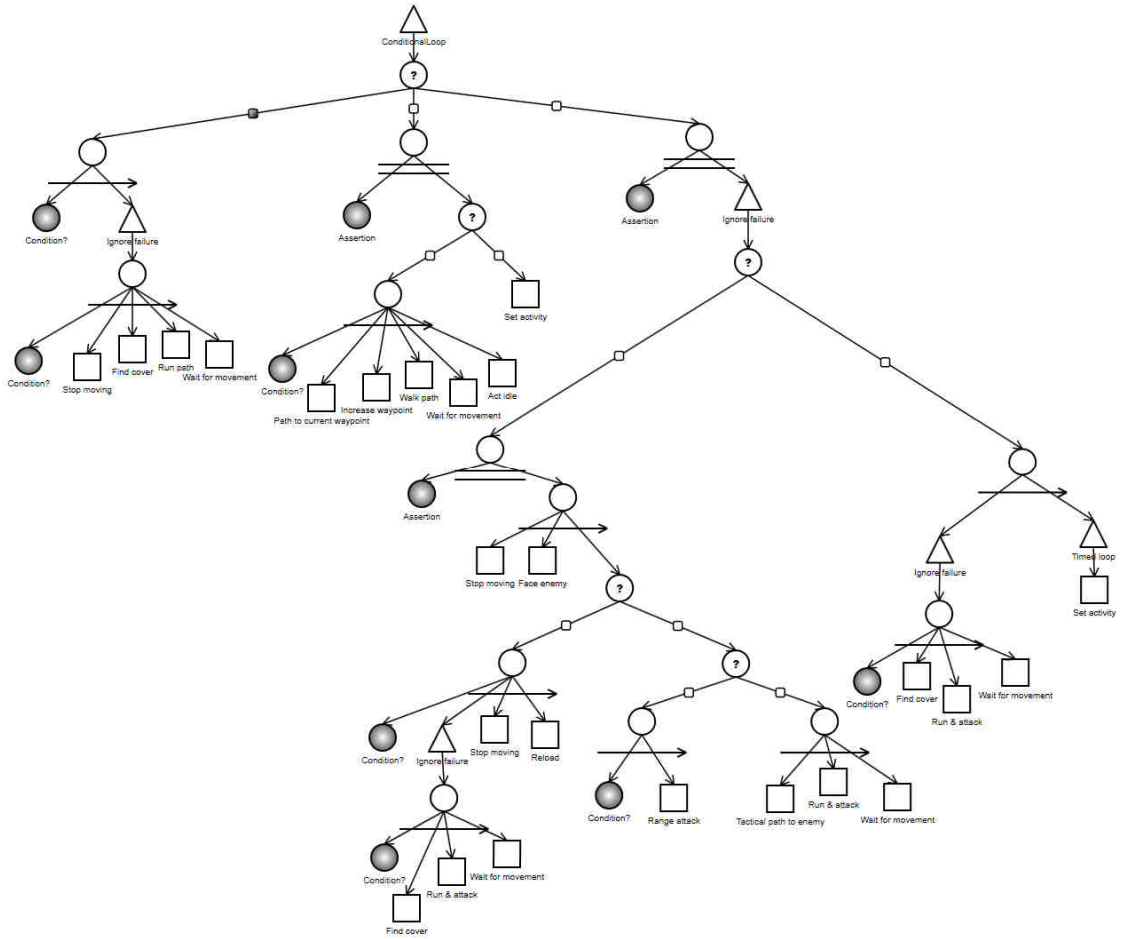


Figure 78. Base behaviour tree used in our example

14.2.1. Cover branch

The first branch our main selector tries to run is the *cover* one. Its root is a *sequence* with a *precondition* that will decide whether to execute the cover logic. Based on our original design, this precondition should check what our health level is. However, we find this is likely to be changed in the future, so we want to leave this open, so new logic can decide if we should run this branch. Because of that, we have decided to use a slightly more complex condition, so our logic will either be triggered by a low health level or if we are receiving a *hint* to cover; this is shown in Figure 79. The addition of this *hint condition* automatically exposes a *COVER* hint that can be used by higher-level trees.

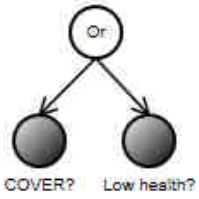


Figure 79. Precondition that will decide whether to make the AI run to a cover position

A second sequence defines the rest of the logic, again using a precondition, shown in Figure 80, which is preventing our agent from looking for a cover position if it is already hidden, and some actions to stop the previous movement, find a new cover position, run to it, and wait until we have got there.

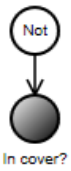


Figure 80. Precondition to prevent AIs from looking for cover positions when they are already hidden

When the branch is chosen by an agent, we will see how it tries to run away and find a suitable position (Figure 81).

Figure 81. AI running away while trying to cover

As we have said, the first thing an AI will try to do is check if it needs to cover. We might want to change this in the future, so we can build, for example, kamikaze AIs.

Because of that, we decided to let our main selector receive *COVER* hints to reorder its priorities (Figure 82).

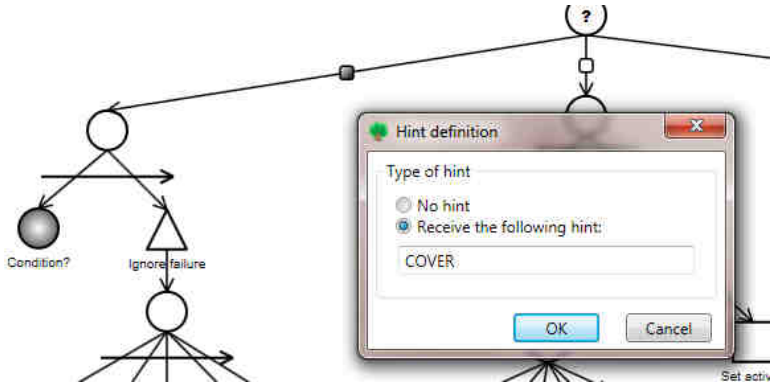


Figure 82. We have modified our main selector so it is able to receive COVER hints from higher-levels

14.2.2. Patrol branch

The second branch (see Figure 83) in the main selector makes agents follow a predefined set of waypoints, in what we can call a *patrol* behaviour. Waypoints are set from Hammer (Half-Life's map editor) in edition time, and parsed by the AI when it is spawned, generating a list of positions to patrol to. We make use of our blackboard (9.1.2) to maintain what our next waypoint is, and increment it every time we reach it.

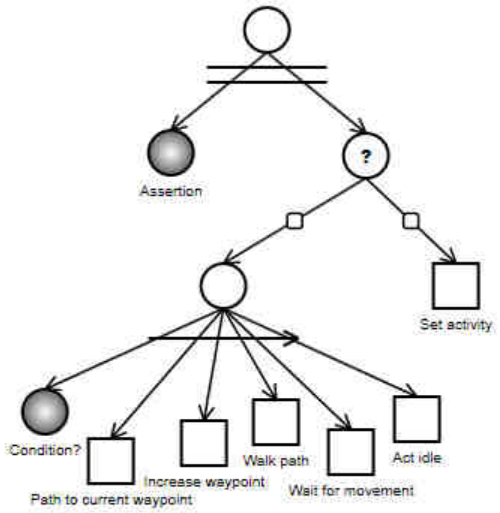


Figure 83. Patrol branch

Our *patrol* behaviour starts with a *parallel* node, which is running an assertion to make sure we do not have an enemy, as we want to abort this logic as soon as we

detect an enemy. Again, we have decided to expand this condition to allow extensions to this check, so the final condition used is shown in Figure 84.

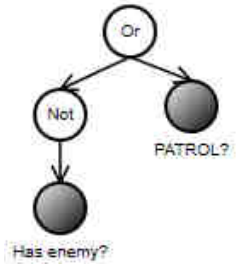


Figure 84. Assertion used by our patrol branch

This way, we have exposed our second hint, *PATROL*, so higher-level trees can control when an AI should be patrolling. Figure 85 shows an AI following its patrol route.

Figure 85. Still unaware of our presence, this agent is just following its patrol route

14.2.3. Attack branch

The last branch defines the aggressive behaviour of our AIs. We show the branch separated from the rest of the tree in Figure 86.

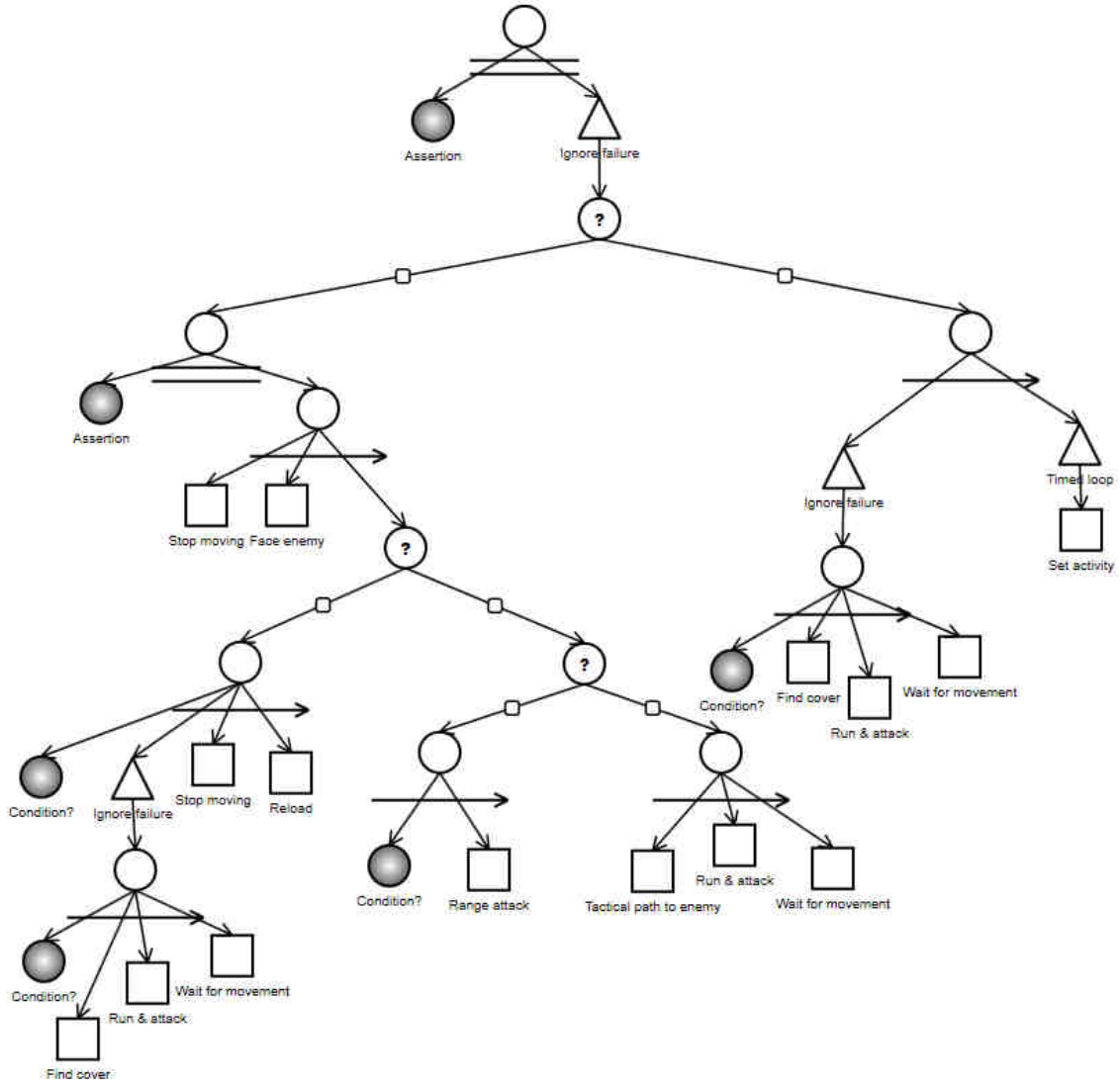


Figure 86. Our attack branch

Again, we have an *assertion*. In this case, it is making sure we have an enemy, as we would not be able to attack if we did not have one. In parallel to this assertion, we are running a *selector*, decorated with an *ignore failure* filter. We have chosen to do this as some of the actions could fail (such as those trying to find a cover position, if there is none), and we do not want the branch to fail.

The selector will first try to attack, or make the agent look for a cover position if it is under attack. We model this using an assertion in the leftmost branch to make it bail out as soon as we detect we have received some damage. Again, we have decided to expose a new hint so we can allow higher-levels to decide when or whether to look for cover. We have named this hint *ATTACK*, as shown in Figure 87.

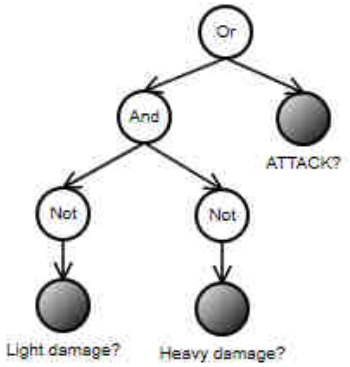


Figure 87. Assertion controlling the execution of the attack branch

If the assertion is met, then we always stop the previous action, face the enemy, and then we use a new selector to choose between running a *reload* sub-behaviour or attacking our enemy; actually, we use yet another selector to check if our enemy is visible before we decide to fire, as we would have to find a path to get back to the action if it was not. Figure 88 shows two AIs using this branch.

Figure 88. An AI trying to find a cover position, but still facing us. We can also see another AI at the back, ready to start its attack behaviour, as it has just spotted us

14.3. High-level trees

Once our AIs are able to run autonomously, meeting the game design, in an ideal situation, most of the work for AI engineers consists of debugging and polishing the behaviours. However, we could find ourselves in a situation when substantial changes to these behaviours are required... or at least needed to test new ideas. This is where the power of HeBTs comes into play.

Again, we should note some examples were already shown in 12.3.3.2, but they were just some simple ones that could have been imitated using a different approach, such as the use of personality traits. In that moment, we also talked about a "disguise system", which we will use as our scenario to demonstrate the capabilities of our new system.

As we have said, in a real project engineers would be in charge of modifying the standard behaviours to take into account the new design request. Let us say the new system should add the following features to the AI and gameplay systems:

- Players can wear the clothes of the enemies they kill, going unnoticed to other AIs if they do so.
- AIs should not recognise "disguised" players as enemies. However, they should react if the player damages them.

Basically, these changes would require gameplay and AI code modifications, and this new feature could not make it through to the final game.

As our game would have been using HeBTs, we could delegate the prototyping of new ideas to designers themselves. We only need to provide them with the right tools to do it. We have already built our systems taking this into account, and so we have developed the tool presented in Chapter 13.

We must bear in mind that if we want to have a system that requires virtually no programming work to be extended, we must start from designing our base behaviours correctly, just as we have done in the previous section. Also, building a complete set of tree nodes and conditions can facilitate things further down the line.

14.3.1. Modifying AIs knowledge

To let high-level trees add new gameplay features, such as the one we are trying to model, we have designed our prototype to allow trees to set and access flags in our AIs blackboards. Just as a reminder, our blackboards are just simple maps storing key/value pairs that can represent any useful information agents might need.

Setting flags is, actually, the communication system we have implemented, so an AI can send some information to other peers. The information we allow to be sent is a simple string identifier for the flag, and a boolean state. This can be done using a *FlagBroadcastAction* node, which works as follows:

```
1  template< class AIInstance >
2  ENodeResult FlagBroadcastAction< AIInstance >::Step()
3  {
4         ASSERT_STR( m_key.size() > 0, L"A valid key wasn't set" );
5         m_pInstance->BroadcastFlag( m_key, m_bValue, m_bSendToMyself );
6         return NR_SUCCEEDED;
7  }
```

This node is using the BroadcastFlag functionality inside AI instances (line 5), which receives the key and state (value) of the flag, as well as an extra flag to indicate whether this information has to be used by the AI broadcasting it:

```
1  template< class NativeAI >
2  void AIInstance< NativeAI >::BroadcastFlag( const std::string& key,
3                                              bool bValue, bool bSendToMyself )
4  {
5         AIManager< NativeAI >::AIVector instances;
6         AIManager< NativeAI >::Ref().FillAIVector( instances );
7
8         AIManager< NativeAI >::AIVector::iterator it = instances.begin();
9         AIManager< NativeAI >::AIVector::iterator end = instances.end();
10        for ( ; it != end; ++it )
11        {
12               AIInstance< NativeAI >* pInstance = *it;
13               if ( bSendToMyself || ( pInstance != this ) )
14               {
15                      pInstance->ReceiveFlag( key, bValue );
16               }
17        }
18 }
```

To broadcast a flag, an instance asks the AIManager for all the instances registered in it (line 6), and uses this to call ReceiveFlag (line 15) on each instance, even the one that originated the call, depending on what behaviour we have chosen to use (line 13).

```
1  template< class NativeAI >
2  void AIInstance< NativeAI >::ReceiveFlag( const std::string& key,
3                                            bool bValue )
4  {
5         if ( bValue )
6         {
7                m_blackboard.Update( ks::Blackboard::GetKeyFromString( key ),
8                                     bValue );
9         }
10        else
11        {
12               // Let's just remove it from the blackboard
13               m_blackboard.Remove( ks::Blackboard::GetKeyFromString( key ) );
14        }
```

```
15  }
```

When an AI receives a flag, it basically keeps it in its blackboard, in case the flag was enabled (lines 5-9) or removes it, if the state of the flag is false (lines 10-14).

We also need to create a new *condition* that checks an AI's blackboard for a flag, and returns true if the flag is present. We did this as shown below:

```
1  bool CheckFlag::Evaluate() const
2  {
3         const ks::Blackboard::Element& element =
4                   m_pInstance->GetBlackboard().GetElement(
5                                 ks::Blackboard::GetKeyFromString( m_key ) );
6         return ( ( element.eType == ks::Blackboard::Element::ET_BOOL )
7                   && ( element.uValue.boolValue ) );
8  }
```

Using these nodes, both action and condition ones, we can make our AIs use additional information about their environment which was not originally planned.

14.3.2. Building our high-level logic

If we do not want gameplay engineers to add new code to support our disguise feature, which we do not as we are only trying to prototype a new idea, then we need a high-level tree that can simulate this.

For designers, high-level trees should be the only thing they care about; they do not even need to know there is a base tree running underneath their logic, so what they really want to do is to build a pretty similar behaviour tree, yet much simpler.

We will start the tree using a "while AI is alive" loop, just as we did with our base behaviour. Want we want to do, is have AIs just patrolling if the player is disguised. So, we want something like this shown in Figure 89.

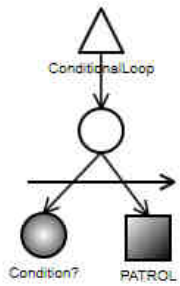


Figure 89. Basic idea behind the disguise system

This is, in a nutshell, what our high-level tree should look like. However, we still have to define our condition. We want to check if the player is disguised, but we do not have a *condition* that does that. So, we will use our new *CheckFlag* condition to try and gather this information. But, if we want to check a flag, we should be setting it somewhere.

If we have a look at our tree so far, we could see it uses a conditional loop as its root. To meet our new design, we need to notify other AIs the player is disguised if they kill us, so we need to use a *FlagBroadcastAction* to send a PLAYER_DISGUISED flag; the question here is... where should we use this node?

To model this, we could use a selector as our tree's root, and choose between our normal behaviour and broadcasting the flag if we are killed. However, recalling 9.5.1.2, a conditional loop will succeed as soon as its condition fails, so if we use this type of node as one of the children of our selector, the selector will succeed as soon as the condition is not met, so the flag would never be sent round. To overcome this, we can use a *Not* filter, which basically will negate our loop's result, causing the selector to choose its second branch, and broadcasting our message to the rest of the AIs. This tree is shown in Figure 90.

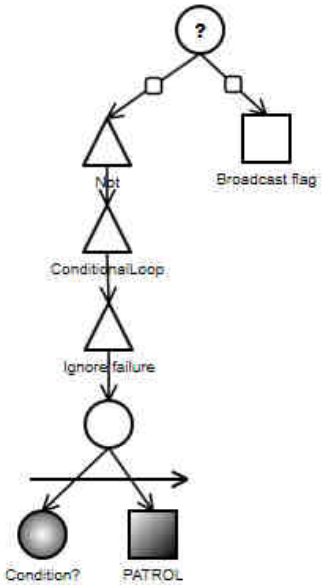


Figure 90. Our first complex high-level tree

The remaining AIs will then have their blackboards updated, and they will start sending *PATROL* hints to their base BTs, causing agents to ignore the player as intended (Figure 91).

Figure 91. An AI ignoring us after being hinted to PATROL

This tree is not completely correct yet, as AIs will not react to damage anymore if the player is disguised. To implement this last feature, we need to clear the PLAYER_DISGUISED flag if the agent is under attack. The final tree is shown in Figure 92.

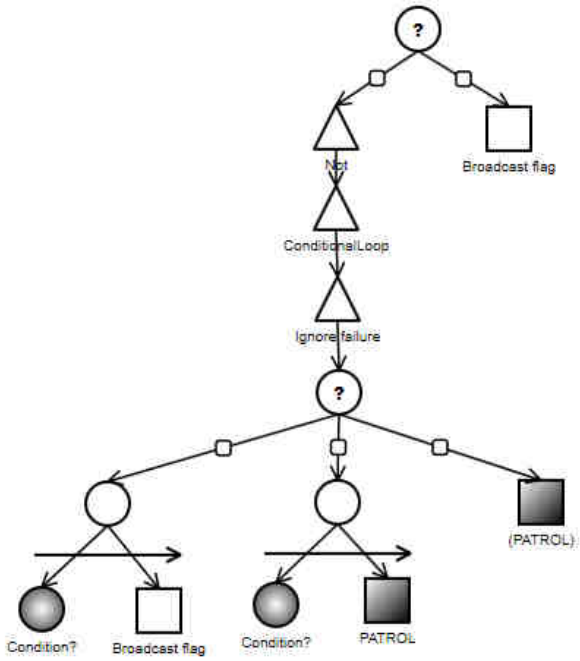


Figure 92. Final high-level tree that models the new feature

In this last tree, we have replaced our sequence with a selector with three branches. First, we need to check if we are under attack, as that means the player has to become our enemy again; we will do so broadcasting the flag with a false status. The condition we are checking is shown in Figure 93.

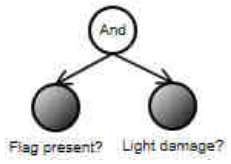


Figure 93. Condition that will decide whether to clear the PLAYER_DISGUISED flag

The second branch is just the one we used in the previous, simpler tree, while the last one is making sure we are not telling the base behaviour to PATROL if the previous branches failed (so, we are sending a *neutral* hint, as described in 10.3).

We can add or remove this tree dynamically, and we have not had to change a line of code, or move a single node in our base tree, removing any new risk. Eventually, if we decide we want this logic to be part of the final behaviour, we could get engineers to integrate the changes into the base tree.

Chapter 15. CONCLUSIONS AND FUTURE LINES OF RESEARCH

In this work, we have presented a novel approach, Hinted-execution Behaviour Trees, to model AI behaviours that can be modified dynamically using a stack of behaviour trees, where each level can hint to the immediate one below what it should try to be doing. This can be applied to commercial games, as we described in Chapter 14.

15.1. Results

Our new technique provides development teams with the required tools to build an AI system that meets the objectives set in Chapter 2. Among its features, we can enumerate the following key characteristics:

- Hinted-execution Behaviour Trees are built based on a methodology where everyone in the team can contribute directly to the quality of the AI. They are also designed to solve real problems game studios face during production time: Our solution allows rapid prototyping, where behaviours are built using a graphical solution, so even non-technical staff can understand and use them. This enhances the use of resources and reduces risks.

- The core of an HeBT system can be built as an abstract library, which will reduce costs. This way, a big part of the AI code can be shared among different projects; only game-specific actions and conditions have to be rewritten for different games.

- Behaviours can still be built in a traditional way if a single tree is used, allowing for autonomous AIs that can run without any additional higher-level logic, or they can be extended as necessary. The number of trees that can be run simultaneously is not limited, which can allow developers to decide how to arrange the different levels, and create very complex

behaviours in a layered fashion, where layers can be added or removed dynamically to obtain different results.

- HeBTs are a data driven solution, so any changes to behaviours can be tested and integrated into games without the need to rebuild the game. This also allows teams to build tools to generate them, improving productivity, and easing the process.

If we study these results using the same method we presented in 7.1, we would find HeBTs obtain the scores shown in Table 16.

Technique	Simplicity (programming)	Modifiable by designers? (no coding involved)	Scalability	Reusability	Behaviour unpredictability	Behaviour extensibility	Overall result (total)
HeBT	2	5	4	4	4	5	24

Table 16. HeBTs get the highest score in our study, mainly for the very good extensibility they offer

Analysing this data we can conclude:

- Our new technique is quite complex to implement; in fact, its complexity is comparable to an automated planner.

- Designers have a greater degree of control over the system; no programming work should be needed for them to be able to control and modify behaviours.

- The system is very scalable; we can add new layers of functionality by adding additional levels in our stacks. An example of this could be the implementation of group behaviours, as discussed in 15.2.2.

- In terms of reusability, HeBTs are quite similar to regular BTs; actions, metanodes, and even whole sub-trees can be reused by different trees.

- Behaviours are more unpredictable with HeBTs, comparable to those produced by automated planners, but, at the same time, they are more controllable.

- Using HeBTs, extending a behaviour is as easy as creating a new high-level tree, based on another existing one; no additional risks are added doing this, as higher-level trees work as plug-ins, and they can be added or removed without changing the original behaviour.

Finally, we must also note this thesis has also produced a fully functional HeBT editor, as well as a complete abstract library, so the system could be quickly applied to any commercial game.

15.2. Future research

As we noted in Chapter 2, this thesis was initially aimed at building a dynamic behaviour selection and modelling system that could allow AI to be adapted to players. Although this is still our long-term objective, we had to divide the work in various stages, the first of which was the design and implementation of a Hinted-execution Behaviour Tree system. Among the future research lines we can mention the following.

15.2.1. Adaptation to players

HeBTs were created as we found there was not any technique that would allow us to modify behaviours depending on players' preferences (Ocio & López Brugos, 2009).

Our technique allows us to build a base behaviour which will be controlled by a higher-level logic. This control will not be complete, as the base behaviour will maintain its autonomy, but will generate new behaviours that are coherent to the original design.

In order to be able to adapt our behaviours to different types of players, we first need to model those users, and we should be able to classify them depending on what we determine will optimise their experience with the game. In fact, we could approach this in two different ways, depending on whether we are modelling player's characteristics, such as accuracy, favourite weapons, etc., or we are describing the game itself.

15.2.1.1. Static classification

If we tried to model the game, rather than individual players, we would choose a set of elements to define a model of enjoyment (Sweetser & Wyeth, 2005). Each element would be given a score, where the higher the score, the more fun the game is.

We also need to classify players in groups, and build some score tables, based on what each group would think about our base behaviours. In this case, adaptation would come from creating high-level trees that are specific to each of the groups, trying to maximise the scores the game was given for each of the categories chosen. A similar approach has also been taken in other fields, such as e-learning (Paule et al., 2003).

A basic example would give a better idea of what this means; let us say we are developing an infiltration, Splinter Cell (Ubisoft, 2010)-like game. We will define two elements to model the enjoyment of players in this case: *action* and *toughness*, where the former describes the amount of clashes with the AI, and the latter evaluates how easy it is to take enemies down. We will also define two classes of players: those who like action, and those who like stealth. Providing we build a base behaviour that makes AI instances that are very good fighters, but weak, we would get scores (ranging from 1 to 10) as shown in Table 17.

	Action	**Toughness**
Prefers action	10	0
Prefers stealth	0	10

Table 17. Scores obtained by our base behaviours for the different classes of players

In order to improve our scores, we can create different high-level trees to hint to our base behaviour what to do. For example, we can create a BT that makes AIs very good at covering, co-operate to avoid showing weak points and prevent getting themselves into situations where they would be an easy target for players; on the other hand, we would build a tree for stealth players, so our agents cannot detect players as easily, prefer to be alone, or try to call for reinforcements before attacking. Using such trees, we would modify our scores, trying to find the optimal solution.

15.2.1.2. Dynamic modelling

Although trying to give our behaviours a score based on some predefined classes of players can work, we might find it is not enough in an environment as dynamic as those presented in videogames. Players can evolve as they play, so many times we will find it is not possible to classify them beforehand and maintain such classification over time.

Instead, we can build systems that can gather information dynamically (Thurau et al., 2003; Laird & Duchi, 2000), or use some classification methods that can be run concurrently or periodically, we could try to adapt the game to what players are doing at each moment.

We could also go a step further and work on machine learning or other types of algorithms that allow our high-level trees to evolve over time. Rather than maintaining some static logic, our trees can change to produce better results. This type of solution could allow us to have a much richer range of behaviours.

15.2.2. Group behaviours

In Chapter 14 we presented an example that used a special node to broadcast messages to other AIs in the system. This was used as a simple implementation of a communication system, which can be used by AIs to share important information; it was just scratching the surface of a complex implementation of group behaviours.

Our hinted-execution model could allow us to create complex group behaviours, based on command hierarchies (Reynolds, 2002; Pittman, 2008), where hints would flow down the chain, allowing some AIs to have a better control over what others should do.

In hint-based system, we would be able to create new links in our chain as high-level trees that are built on top of several base behaviours, rather than just one; in this case, each base tree would expose the orders that a particular class of AI can accept. We should also note these would not be regular BTs, but full HeBTs, so individual behaviours can be hinted as well. Our higher-level tree would be able to broadcast hints to groups of AIs that are using the behaviours this level was based on.

An example of this would be a small army that has warriors, archers and medics. A simplified version of their behaviours can be seen in Figure 94.

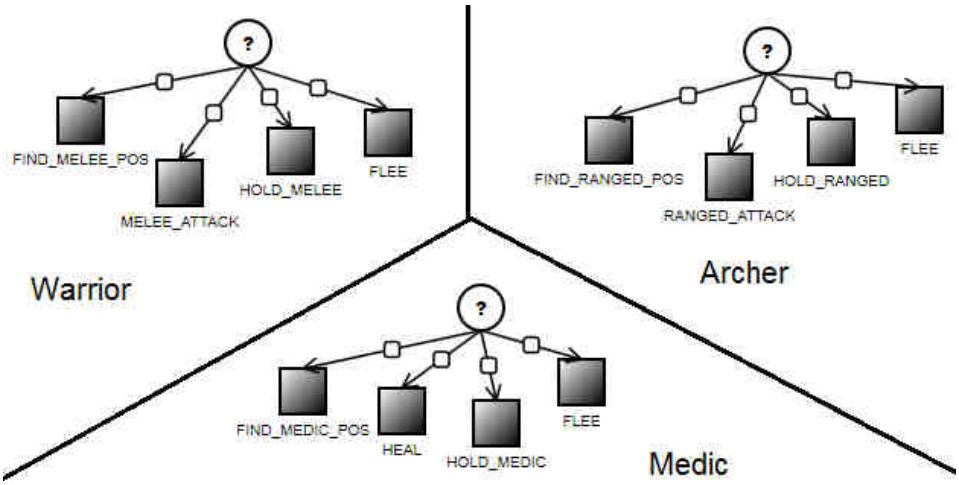


Figure 94. Base BTs controlling the different classes of AI in our army

We could use different generals, defining different high-level trees to create an intelligent army obeying the orders we want to send. For instance, we could build a high-level AI that wants to attack with its archers first, holding warriors and medics, to continue with a melee attack, sending the medics along with our warriors to try and heal wounded units... or a variation of this, hinting our units to never retreat, such as the one shown in Figure 95.

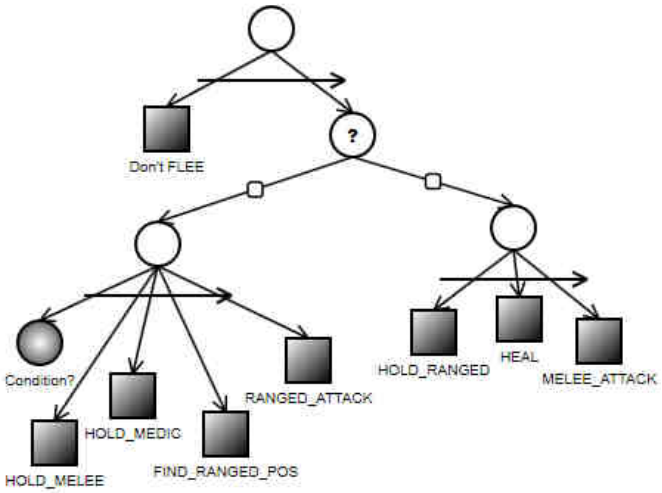


Figure 95. Tyrant high-level AI that never allows individuals to retreat

Groups, that is, high-level entities, would also be controlled by HeBTs, so they can receive hints too, and we could end up with a complex chain of command that can help us create more credible group behaviours.

15.2.3. Moving to a production environment

Although our system is fully functional, and can be applied to any game, there are some aspects of it that should be improved before it can be used efficiently in a production environment.

Among these, we can find some extra features the editor would benefit from, as we showed in 13.6, and optimising the existing library to reduce memory usage or enhance performance. Further than that, there is some essential functionality that would be required to make the system usable in a big team so we can:

- Integrate the editor with a source control solution, such as Perforce, so we keep track of changes in behaviours.
- Allow different people to work on the same behaviour concurrently, merging different changes and offering a way to resolve conflicts.
- Implement a strong validity checker that assures the data exported by the editor is correct and it cannot corrupt or crash the game.

REFERENCES

Alur, R., Kannan, S., & Yannakakis, M. (1999). *Communicating hierarchical state machines*. In Proceedings of the 26th International Colloquium on Automata, Languages and Programming (pp. 169-178). Springer.

Ambroszkiewicz, S., & Komar, J. (2006). *A Model of BDI-Agent in Game-Theoretic Framework*. In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06) (pp. 8-19). Models of Agents, ESPRIT Project Modelage Final Workshop.

Atari Inc. (1972). *Pong*. http://en.wikipedia.org/wiki/Pong.

Avalanche Studios (2010). *Just Cause 2*. http://www.justcause.com/.

Baekkelund, C. (2006). Academic AI Research and Relations with the Games Industry. *AI Game Programming Wisdom 3*, (pp. 77-88). Higham, Massachusetts: Charles River Media, Inc.

Bangeman, E. (2008). *Growth of gaming in 2007 far outpaces movies, music*. Retrieved February 1, 2010, from http://arstechnica.com/gaming/news/2008/01/growth-of-gaming-in-2007-far-outpaces-movies-music.ars

Beaudry, E., Brosseau, Y., Côté, C., Räievsky, C., Létourneau, D., Kabanza, F., & Michaud, F. (2005). *Reactive planning in a motivated behavioural architecture*. In Proceedings of the American Association for Artificial Intelligence Conference (pp. 1242-1247).

Bethesda Game Studios (2008). *Fallout 3*. http://fallout.bethsoft.com/.

Bratman, M. E. (1987). Intention, Plans, and a Practical Reasoning. Harvard University Press. Cambridge MA., USA.

Bratman, M. E., Israel, D. J., & Pollack, M. E. (1988). Plans and resource bounded practical reasoning. *Computational Intelligence, 4. (pp. 349-355).*

Bryant, B. D. (2006). *Evolving Visibly Intelligent Behavior for Embedded Game Agents*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

Buckland, M. (2005). Programming Game AI by example. Texas: Wordware Publishing, Inc.

Bungie (2001). *Halo*. http://www.bungie.net/.

Castelfranchi, C., Falcone, R., & Piunti, M. (2006). *Agents with anticipatory behaviours: To be cautious in a risky environment*. In Proceedings of the European Conference on Artificial Intelligence. Trento, Italy.

Champandard, A. J. (2007). Using resource allocators to synchronize behaviors. Retrieved February 10, 2010, from http://aigamedev.com/open/articles/allocator/

Champandard, A. J. (2008). Getting started with decision making and control systems. *AI Game Programming Wisdom 4*, (pp. 257-263). Boston, Massachusetts: Course Technology.

Champandard, A. J. (2009-1), *AI Blueprints for Action & Combat Behavior Trees*. Retrieved October 15, 2009, from http://aigamedev.com/premium/masterclass/blueprint-combat-ai/.

Champandard, A. J. (2009-2), *Behavior Tree Design Patterns: Priorization*. Retrieved October 15, 2009, from http://aigamedev.com/premium/masterclass/bt-prioritization/.

Champandard, A. J. (2009-3), *Dynamic Decisions: Building AI that can change its mind*. Retrieved October 15, 2009, from http://aigamedev.com/premium/masterclass/dynamic-decisions/.

Champandard, A. J. (2010-1), *Why State Machines Struggle with Concurrency*. Retrieved September 29, 2010, from http://aigamedev.com/open/articles/fsm-struggle/

Champandard, A. J. (2010-2), *10 Reasons the Age of Finite State Machines is Over*. Retrieved September 29, 2010, from http://aigamedev.com/open/articles/fsm-age-is-over/

Champandard, A. J., Iassenev, D., Merrill, B., Orkin, J., & Pfeifer, B. (2009). *Special Report: Goal-Oriented Action Planning*. Retrieved October 15, 2009, from http://aigamedev.com/premium/reports/goal-oriented-action-planning/.

Charles, D., & Black, M. (2004). *Dynamic player modelling: A framework for player-centric digital games*. Computer Games: Artifical Intelligence, Design and Education (CGAIDE 2004). (pp. 29-35). University of Wolverhampton.

Cremer, J., Kearney, J., & Papelis, Y. (1995). *HCSM: A framework for behaviour and scenario control in virtual environments*. ACM Transactions on Modelling and Computer Simulation, 5, 3, (pp. 242-267).

Cutumisu, M., & Szafron. D. (2009). An Architecture for Game Behavior AI: Behavior Multi-Queues. In Proceedings if the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009). AAAI Press.

D'Angelo, G. J. (1983). *Tutorial on petri nets*. ACM SIGSIM Simulation Digest. Volume 14, Issue 1-4, pp. 10-25.

Dennett, D.C. (1987). The Intentional Stance. MIT Press. Cambridge MA., USA.

Diller, D., Ferguson, W., Leung, A., Benyo, B., & Foley, D. (2004). *Behavior modelling in commercial games*. In Proceedings of the Thirteenth Conference on Behaviour Representation in Modeling and Simulation. Orland, FL. University of Central Florida.

Downey, A. B. (2005). *The Little Book on Semaphores*. Green Tea Press. Available at http://greenteapress.com/semaphores/.

Drachen, A., Canossa, A., & Yannakakis, G. N. (2009). *Player Modelling using Self-Organization in Tomb Raider: Underworld.* In Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG2009) (Milano, Italy). IEEE Computational Intelligence Society.

Ellinger, B. (2008). Artificial personality: A personal approach to AI. *AI Game Programming Wisdom 4*, (pp. 17-26). Boston, Massachusetts: Course Technology.

Epic Games. (2009). *Unreal Development Kit (UDK)*. http://www.udk.com/.

Erol, K., Nau, D., & Hendler, J. (1994). *HTN planning: Complexity and expressivity*. In Proceedings of the Twelfth National Conference on Artificial Intelligence (pp. 1123-1128). Menlo Park, California: American Association for Artificial Intelligence.

Evans, R. (2002). Varieties of Learning. *AI Game Programming Wisdom*, (pp. 567-578). Higham, Massachusetts: Charles River Media, Inc.

Firby, R. J. (1987). *An investigation into reactive planning in complex domains*. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87) (pp. 202–206). Milan, Italy.

Fu, D., & Houlette, R. (2004). The ultimate guide to FSMs in games. *AI Game Programming Wisdom 2*, (pp. 283-301). Higham, Massachusetts: Charles River Media, Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* . Addison-Wesley Professional, USA.

Garces, S., & Champandard, A. J. (2009). *AI For Dynamic Large-Scale Open Worlds in [Prototype] with Sergio Garces*. Retrieved February 7, 2010, from http://aigamedev.com/premium/interviews/prototype-large-scale/.

Georgeff, M. P., & Lansky, A. L. (1987). *Reactive Reasoning and Planning*. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87) (pp. 677-682). American Association for Artificial Intelligence. Menlo Park, California.

Georgeff, M. P., Pell, B., Pollack, M., Tambe, M., & Wooldridge, M. (1999). *The Belief-Desire-Intention Model of Agency*. In Proceedings of the Fifth International Workshop on Intelligent Agents: Agent Theories, Architectures and Languages (ATAL-98). Lecture Notes in Artificial Intelligence, vol. 1555 (pp. 1-10). Springer Verlag. Hedelgerg, Germany.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). PDDL – The Planning Domain Definition Language V. 2. Technical Report, report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

GSC Game World (2007). S.T.A.L.K.E.R. *Shadow of Chernobyl*. http:// www.stalker-game.com/.

Guerra-Hernández, A., El Fallah-Seghrouchni, A., & Soldano, H. (2004). *Learning in BDI Multi-agent Systems*. In Proceedings of Computational Logic in Multi-Agent systems (CLIMA IV) (pp 218-233). Springer-Verlag. Fort Lauderdale, FL. USA.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, (pp. 231-274). North-Holland.

Harland, J., Thangarajah, J., & Padgham, L. (2002). *Representation and reasoning for goals in BDI agents*. In Proceedings of the 25th Australian Computer Science Conference (ACS2002).

Hart, P. E., Nilsson, N. J., Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics (SSC4) 4 (2): 100-107.

Haugeland, J. (1985). *Artificial Intelligence: The very idea*. MIT Press, Cambridge, MA.

Hoang M., Lee-Urban, S., & Muñoz-Avila, H. (2005). *Hierarchical plan representations for encoding strategic game AI*. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05). AAAI Press.

Hopcroft, J., & Ullman, J. (1979). Introduction to Automata Theory, Languages, and Computation. Reading, MA: Addison-Wesley.

Houlette, R. (2004). Player modelling for adaptive games. *AI Game Programming Wisdom 2*, (pp. 557-565). Higham, Massachusetts: Charles River Media, Inc.

Houlette, R., Fu D., & Ross, D. (2001). *Towards an AI Behaviour Toolkit for Games*. AAAI 2001 Spring Symposium on AI and Interactive Entertainment.

id Software (2001). *Doom 3*. http://www.idsoftware.com/games/doom/doom3/.

Isla, D. (2005). Handling complexity in the Halo 2 AI. In Proceedings of the Game Developers Conference (GDC) 2005.

Isla, D. (2008). Building a better battle: the Halo 3 AI Objectives System. In Proceedings of the Game Developers Conference (GDC) 2008.

Kelly, J. P, Botea, A., & Koenig, S. (2008). Offline planning with hierarchical task networks in video games. In 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008) (pp. 60-65), Stanford, USA.

Laird, J. E. (2002). *Research in human-level AI using computer games*. Communications of the ACM. Volume 5, Issue 1. Special Issue: Game engines in scientific research, pp. 32-35.

Laird, J. E., & Duchi, J. C. (2000). *Creating human-like synthetic characters with multiple skill levels: A case study using the Soar Quakebot*. In Papers from the AAAI 2000 Fall Symposium on Simulating Human Agents (Tech. Rep. FS-0A-03; pp. 75–79). Menlo Park, CA: AAAI Press.

Laming, B. 2009. *From the ground Up: AI architecture and design patterns*. In Proceedings of Game Developers Conference,San Francisco.

LaMothe, A. (2002). Finite State Machines. Tricks of the Windows Game Programming Gurus Second Edition, (pp. 737-742). Sams Publishing (Macmillian).

Lekavý, M., & Návrat., P. (2007). *Expressivity of STRIPS-like and HTN-like planning*. In Proceedings of Agent and Multi-Agent Systems: Technologies and Applications, First KES International Symposium (KES-AMSTA) (pp. 121-130).

Lionhead Studios (2001). Black & White. http://www.lionhead.com/bw/Default.aspx.

Marinucci, A. (2010). AvalonDock. http://avalondock.codeplex.com/.

Maxis Software (2008). *Spore*. http://www.spore.com/.

Monolith Productions (2005). *F.E.A.R. First Encounter Assault Recon*. http://www.lith.com/.

Nareyek, A. (2004). *AI in Computer Games*. Queue. Feature: Q Focus: Game Development, 1(10), 58-65.

Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). *SHOP: Simple Hierarchical Ordered Planner*. In Proceedings of IJCAI'99 (pp. 968-973).

Nilsson, N., & Fikes, R. (1971). *STRIPS: A new approach to the application of theorem proving to problem solving*. Artificial Intelligence, 2(1971), 189-208.

Norling, E. (2004). *Folk Psychology for Human Modelling: Extending the BDI Paradigm*. In Proceedings of Autonomous and Multiagent Systems (AAMAS'04) – Volume 1 (pp. 202-209). New York, NY. USA.

O'Brien, J. (2002). A flexible goal-based planning architecture. *AI Game Programming Wisdom*, (pp. 375-383). Higham, Massachusetts: Charles River Media, Inc.

Ocio, S., & López Brugos, J. A. (2009). *Multi-agent Systems and Sandbox Games*. In Proceedings of AISB09. AI & Games (pp. 70-74).

Orkin, J. (2002). 12 Tips from the trenches. *AI Game Programming Wisdom*, (pp. 29-35). Higham, Massachusetts: Charles River Media, Inc.

Orkin, J. (2004-1). Applying goal-oriented action planning to games. *AI Game Programming Wisdom 2*, (pp. 217-227). Higham, Massachusetts: Charles River Media, Inc.

Orkin, J. (2004-2). Symbolic representation of game World state: Toward Real-Time planning in games. In AAAI Challenges in Game AI Technical Report, 26-30. Menlo Park, California: AAAI Press.

Orkin, J. (2006). Three states and a plan: The A.I. of F.E.A.R.. In Proceedings of the Game Developers Conference (GDC) 2006.

Paule, M. P., Ocio, S., Pérez, J. R., & González, M. (2003). *Feijoo.net: An approach to personalized e-learning using learning styles*. In Proceedings of ICWE 2003. Lecture notes in Computer Science, volume 2722/2003. Berlin, Heidelberg: Springer-Verlag, (pp. 151-154).

Peterson, J. L. (1977). *Petri Nets*. ACM Computing Surveys (CSUR). Volume 9, Issue 3., (pp. 223-252).

Pittman, D. (2008). Command Hierarchies using Goal-Oriented Action Planning. *AI Game Programming Wisdom 4*, (pp. 383-391). Boston, Massachusetts: Course Technology.

Rabin, S. (2002). Enhancing a State Machine Language through Messaging. *AI Game Programming Wisdom*, (pp.). Higham, Massachusetts: Charles River Media, Inc.

Rabin, S. (2004). Common Game AI Techniques. *AI Game Programming Wisdom 2*, (pp. 3-14). Higham, Massachusetts: Charles River Media, Inc.

Radical Entertainment (2009). *Prototype*. http://www.prototypegame.com/.

Rao, A. S., & Georgeff, M. P. (1991). *Modelling Rational Agents within a BDI architecture*. In Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91) (pp. 473-484). Morgan Kaufmann.

Rao, A. S., & Georgeff, M. P. (1995). *BDI agents: From theory to practice*. In Proceedings of the First Conference on Multi-agent Systems (ICMAS95).

Reynolds, J. (2002). Tactical Team AI using a Command Hierarchy. *AI Game Programming Wisdom*, (pp. 567-578). Higham, Massachusetts: Charles River Media, Inc.

Rockstar Games (2010). *Red Dead Redemption*. http://www.rockstargames.com/reddeadredemption/.

Russell, S., Graetz, M., Witaenem, W. (1962). SpaceWar!. http://en.wikipedia.org/wiki/Spacewar!

Sardina, S., & Padgham, L. (2007). *Goals in the context of BDI plan failure and planning*. In Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems. Honolulu, Hawaii. Article No.: 7.

Snavely, P. J. (2006). Custom Tool Design for Game AI. *AI Game Programming Wisdom 3*, (pp. 3-12). Higham, Massachusetts: Charles River Media, Inc.

Spronck, P. (2005). *Adaptive Game AI*. Ph.D. thesis. Universitaire Pers Maastricht.

Sweetser, P., & Wyeth, P. (2005). *GameFlow: a model for evaluating player enjoyment in games*. Computers in Entertainment (CIE). SECTION: Games. (pp. 3-3). New York, USA: ACM.

The Creative Assembly (2009). *Empire: Total War*. http://www.totalwar.com/.

Thurau, C., Bauckhage, C., & Sagerer, G. (2003). Combining self organizing maps and multilayer perceptrons to learn bot-behaviour for a commercial game". In GAME-ON, 2003. (pp. 119-123).

Tozour, P. (2002). The Perils of AI Scripting. *AI Game Programming Wisdom*, (pp. 541-547). Higham, Massachusetts: Charles River Media, Inc.

Tozour, P. (2004). Stack-Based Finite-State Machines. *AI Game Programming Wisdom 2*, (pp. 303-306). Higham, Massachusetts: Charles River Media, Inc.

Ubisoft (2009). *Assassin's Creed 2*. http://assassinscreed.uk.ubi.com/assassins-creed-2/.

Ubisoft (2010). *Splinter Cell Conviction*. http://splintercell.uk.ubi.com/conviction/.

Ubisoft (2011). *Driver: San Francisco*. http://driver.uk.ubi.com/san-francisco/.

Valve Corporation (2000). *Counter-Strike*. http://store.steampowered.com/css.

Valve Corporation (2004). *Half-Life 2*. http://orange.half-life2.com/.

Valve Corporation (2007). *Portal*. http://www.whatistheorangebox.com/.

Valve Corporation (2008). *Left 4 Dead*. http://www.l4d.com/.

Valve Corporation (2010). *Valve Developer Community*. http://developer.valvesoftware.com/wiki/Main_Page

Wallace, N. (2004). Hierarchical planning in dynamic worlds. *AI Game Programming Wisdom 2*, (pp. 229-235). Higham, Massachusetts: Charles River Media, Inc.

Wallace, N. (2006). Designing for Emergence. *AI Game Programming Wisdom 3*, (pp. 45-53). Higham, Massachusetts: Charles River Media, Inc.

Wallop, H. (2009). *Video games bigger than film*. Retrieved February 1, 2010, from http://www.telegraph.co.uk/technology/video-games/6852383/Video-games-bigger-than-film.html.

Wikipedia (2010). *Visitor pattern*. Retrieved June 17, 2010, from http://en.wikipedia.org/wiki/Visitor_pattern

Wilkins, D. E., Myers, K. L., Lowrance, J. D., & Wesley, L. P. (1995). Planning and reacting in uncertain and dynamic environments. Journal of Experimental and Theoretical AI 7(1), (pp. 197-227)

Young, R. M., Riedl, M., Branly, M., Jhala, A., Martin, R. J., & Saretto, C.J. (2004). *An architecture for integrating plan-based behaviour generation with Interactive game environments*. The Journal of Game Development, Volume 1 (1).

Yue, B., & de-Byl P. (2006). The state of the art in game AI standardisation. In Proceedings of the 2006 international conference on Game research and development (pp. 41-46), Perth, Australia.