

BhTSL, Behavior Trees Specification and Processing

Miguel Oliveira

Centro ALGORITMI, DI, Universidade do Minho, Portugal

Pedro Mimoso Silva

Centro ALGORITMI, DI, Universidade do Minho, Portugal

Pedro Moura

Centro ALGORITMI, DI, Universidade do Minho, Portugal

José João Almeida

Centro ALGORITMI, DI, Universidade do Minho, Portugal

Pedro Rangel Henriques

Centro ALGORITMI, DI, Universidade do Minho, Portugal

Abstract

In the context of game development, there is always the need for describing behaviors for various entities, whether NPCs or even the world itself. That need requires a formalism to describe properly such behaviors.

As the gaming industry has been growing, many approaches were proposed. First, finite state machines were used and evolved to hierarchical state machines. As this wasn't enough, a more powerful concept appeared. Instead of using states for describing behaviors, people started to use tasks. This concept was incorporated in behavior trees.

This paper focuses in the specification and processing of these behavior trees. A DSL designed for that purpose will be introduced. It will also be discussed a generator that produces \LaTeX diagrams to document the trees, and a Python module to implement the behavior described. Additionally, a simulator will be presented. These achievements will be illustrated using a concrete game as a case study.

2012 ACM Subject Classification Replace `ccsdsc` macro with valid one

Keywords and phrases Game development, Behavior trees (BT), DSL, NPC, Code generation

Digital Object Identifier 10.4230/OASICS.CVIT.2016.23

Acknowledgements I want to thank ...

1 Introduction

At some point in the video-game history, NPCs (Non-Playable Characters) were introduced. With them came the need to describe behaviors. And with these behaviors came the need of the existence of a formalism so that they can be properly specified.

As time passed by, various approaches were proposed and used, like finite and hierarchical state machines. These are state-based behaviors, that is, the behaviors are described through states. Although this is a clear and simplistic way to represent and visualize small behaviors, it becomes unsustainable when dealing with bigger and more complex behaviors. Some time later, a new and more powerful concept was introduced: using tasks instead of states to describe behaviors. This concept is incorporated in what we call behavior trees.

Behavior trees (BT) were first used in the videogame industry in the development of the game *Halo 2*, released in 2004 [2, 1]. The idea is that people create a complex behavior by only programming actions (or tasks) and then design a tree structure whose leaf nodes are actions and the inner nodes determine the NPC's decision making. Not only these provide



© John Q. Public and Joan R. Public;
licensed under Creative Commons License CC-BY
42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:5

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an easy and intuitive way of visualizing and designing behaviors, they also provide a good way to work with scalability through modularity, solving the biggest issue from state-based design. Since then, multiple gaming companies adopted this concept and, in recent years, behavior trees are also being used in different areas like Artificial Intelligence and Robotics.

In this context, we felt that it could be useful to have a DSL to specify BTs independently of application area and the programming language chosen for the implementation. The language must be compact and easy to use but it should be expressive enough to be applied to real situations. In that sense a new kind of node was included, as will be described.

This paper will introduce the DSL designed and the compiler implemented to translate it to a programming language, in this case Python. Additionally, the compiler also generates \LaTeX diagrams to produce graphical documentation for each BT specified.

XXX (TODO) game will be described in our language as a case study to illustrate all the achievements attained.

The paper is organized as followed: Concepts 2 State of the Art 3 Architecture and Specification 4 Tools 5 Example 6 Conclusion 7 .

2 Concepts

Formally, a BT is a tree whose internal nodes are called control flow nodes and leafs are called execution nodes.

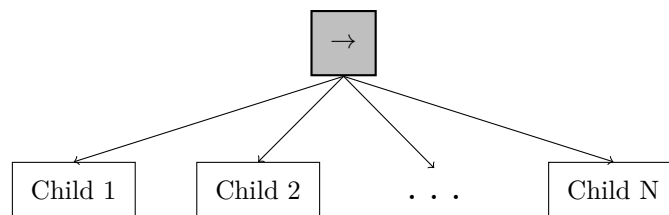
A behavior tree executes by periodically sending ticks to its children, in order to traverse the entire tree. Each node, upon a tick call, returns one of the following three states to its parent: **SUCCESS** if the node was executed with success; **FAILURE** if the execution failed; or **RUNNING** if it could not finish the execution by the end of the tick. In the last case, the next tick will traverse the tree until it reaches the running execution node, and will try again to run it.

2.1 Control Flow Nodes

Control flow nodes are structural nodes, that is, they don't have any impact in the state of the system. They only control the way the subsequent tree is traversed. In the classical formulation, there are 4 types of control flow nodes: **Sequence**, **Selector**, **Parallel** and **Decorator**.

A sequence node visits its children in order, starting with the first, and advancing for the next one if the previous succeeded. Returns:

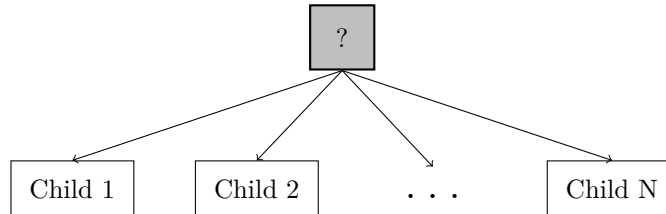
- **SUCCESS** - if all children succeed;
- **FAILURE** - if a child fails;
- **RUNNING** - if a child returns **RUNNING**.



■ **Figure 1** Sequence node.

Like the sequence, the selector node also visits its children in order, but it only advances if the child that is being executed returns **FAILURE**. Returns:

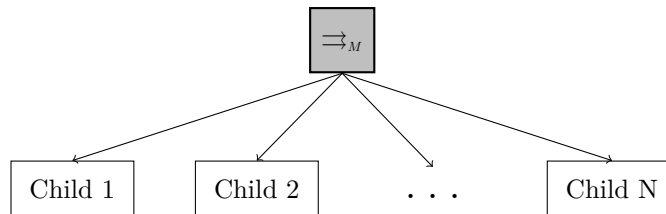
- **SUCCESS** - if a child succeeds;
- **FAILURE** - if all children fails;
- **RUNNING** - if a child returns **RUNNING**.



■ **Figure 2** Selector node.

A parallel node, as the name implies, visits its children in parallel. Additionally, it has a parameter M that acts as a success rate. For N children and $M \leq N$, it returns:

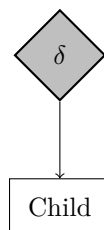
- **SUCCESS** - if M children succeed;
- **FAILURE** - if $N - M + 1$ children fail;
- **RUNNING** - otherwise.



■ **Figure 3** Parallel node.

A decorator is a special node that has an only one child, and uses a policy (set of rules) to manipulate the return status of its child, or the way it ticks it. Some examples of decorator nodes are:

1. **Inverter** - inverts the **SUCCESS/FAILURE** return status of the child;
2. **Max- N -Times** - the child can only fail N times. After that it only returns **FAILURE** without ticking the child.



■ **Figure 4** *Decorator* with policy δ .

94 2.2 Execution Nodes

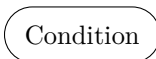
95 Execution nodes are the simplest, yet the more powerful. They are the ones that have access
 96 to the state of the system, and can update it. There are two types of execution nodes:
 97 **Action** and **Condition**.

98 Upon the execution of a tick, an action node runs a chunk of code that can return either
 99 SUCCESS, FAILURE or RUNNING



■ **Figure 5** *Action* node.

100 The condition node verifies a proposition, returning SUCCESS/FAILURE if the proposition
 101 is/is not valid. This node never returns RUNNING.



■ **Figure 6** *Condition* node.

102 2.3 Control Flow Nodes with memory

103 Sometimes, when a node returns RUNNING, we want it to remember which nodes he already
 104 executed, so that the next tick doesn't execute them again. We call this nodes with memory.
 105 And they are represented by adding a `_*` to the symbols mentioned previously. This is only
 106 sintatic sugar because we can also represent these nodes with a non-memory BT, but that
 107 will not be discussed here.

108 Please note that, while we avoid the re-execution of nodes with this type of node, we also
 109 lose the reactivity that this re-execution provides.

110 3 State of The Art

111 In the gaming industry there is some interesting projects that use tools based on Behavior
 112 trees as the main focus to describe NPCs behaviors. Unreal Engine and Unity are two
 113 examples of major game engines that use them. In their case, instead of a language, they offer
 114 a graphical user interface (GUI) to specify the BTs, through a drag and drop tactic. Upon
 115 the creation of an execution node, the programmer needs to specify the action or condition
 116 that will be executed. The nodes mentioned before are all implemented in these engines,
 117 along with some extensions. All the nodes that were mentioned before are implemented in
 118 both of these engines, along with some extensions.

119 In addition to game engines, there are also frameworks like Java Behavior Trees for Java
 120 and Owyl for Python that implement BTs. In this case, they work as a normal library.

121 4 Architecture and Specification

122 - DIAGRAMA DA ARQUITETURA

4.1 DSL

4.1.1 File structure

In our language, each file represents one and only one behavior. A file is divided in 3 components:

- *Behavior* - main behavior tree;
- *Definitions* (optional) - node definitions that can be referenced in other nodes or in the main BT;
- *Code* - Python code where are described the execution nodes, and other code that the programmer wishes to add.

4.1.2 Syntax

Meter exemplo

4.2 Compiler

4.2.1 Lexical analysis

4.2.2 Syntatic analysis

4.2.3 Semantic analysis

5 Tools

6 Example

7 Conclusion

– Ver com professores

References

- 1 Michele Colledanchise and Petter Ogren. *Behavior Trees in Robotics and AI: An Introduction*. Chapman & Hall/CRC Press, 07 2018. doi:10.1201/9780429489105.
- 2 Guillem Travila Cuadrado. Behavior tree library, 2018.