# 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. Aditionally, a simulator will be presented. These achievements will be illustrated using a concrete game as a case study.



**2012 ACM Subject Classification** Replace ccsdesc 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. Altough 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 [5]. 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 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 Inteligence 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.

A small example will be described in our language as a case study to illustrate all the achievements attained.

The paper is organized as follow: Concepts and State of the Art frameworks are presented in Section 2. Architecture and language specification are proposed in Section 3. Compiler development is discussed in Section 4. An illustrative example is presented in Section 5, before concluding the paper in Section 6. The paper also includes two appendices, one for tokens' table and another for the example specification.

## 2 Concepts

This section will be built based on references [1, 4, 3].

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 peridiocally 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 (figure 1a) 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`.

Like the sequence, the selector node (figure 1c) 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`.

A parallel node (figure 1e), 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.

A decorator (figure 1b) 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:

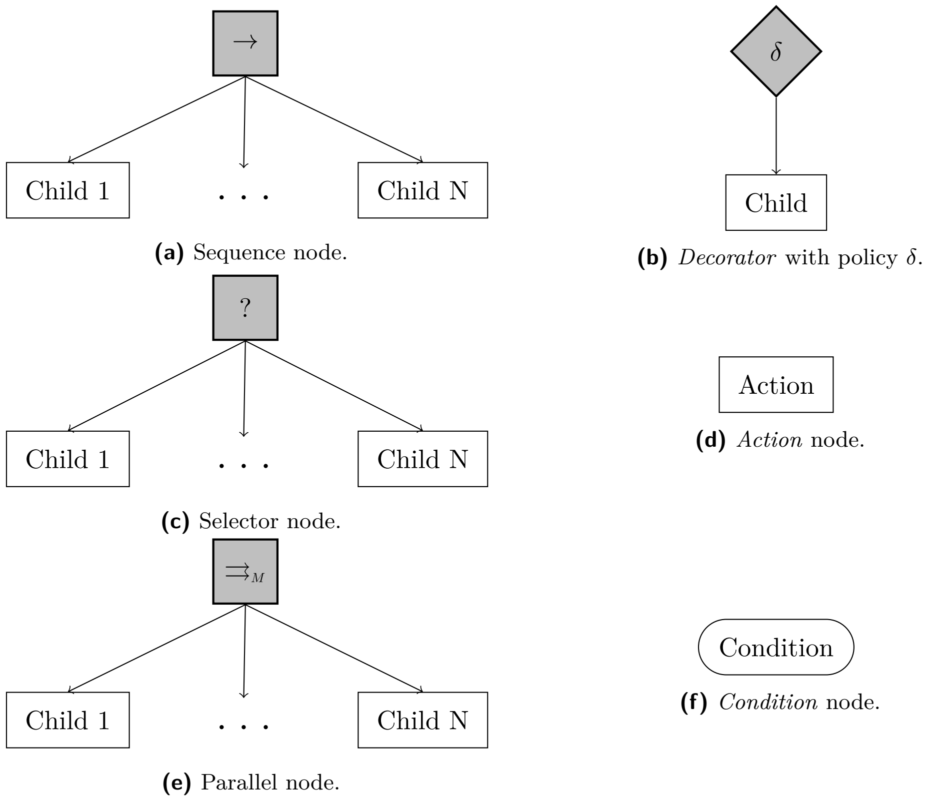
1. **Inverter** - inverts the `SUCCESS`/`FAILURE` return status of the child;
2. **Max-$N$-Times** - the child can only fail $N$ times. After that it only returns `FAILURE` without ticking the child.

## 2.2 Execution Nodes

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

Upon the execution of a tick, an action node (figure 1d) runs a chunk of code that can return either `SUCCESS`, `FAILURE` or `RUNNING`

The condition node (figure 1f) verifies a proposition, returning `SUCCESS`/`FAILURE` if the proposition is/is not valid. This node never returns `RUNNING`.




**(a)** Sequence node.

**(b)** *Decorator* with policy $\delta$.

**(c)** Selector node.

**(d)** *Action* node.

**(e)** Parallel node.

**(f)** *Condition* node.

**Figure 1** BT nodes' structure

## 2.3 Control Flow Nodes with memory

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

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

## 2.4 State of The Art

In the gaming industry there is some interesting projects that use tools based on Behavior trees as the main focus to describe NPCs behaviors. Unreal Engine [2] and Unity[1] are two examples of major game engines that use them. In their case, instead of a language, they offer a graphical user interface (GUI) to specify the BTs, through a drag and drop tactic. Upon the creation of an execution node, the programmer needs to specify the action or condition that will be executed. The nodes mentioned before are all implemented in these engines, along with some extensions. All the nodes that were mentioned before are implemented in both of these engines, along with some extensions.

In addition to game engines, there are also frameworks like Java Behavior Trees[2] for Java and Owyl[3] for Python that implement BTs. In this case, they work as a normal library.

# 3 Architecture and Specification

In this section, it will be explained the general architecture of our system to process BTs, that is depicted in Figure 2. After introducing its modules, one subsection is devoted to the DSL design.

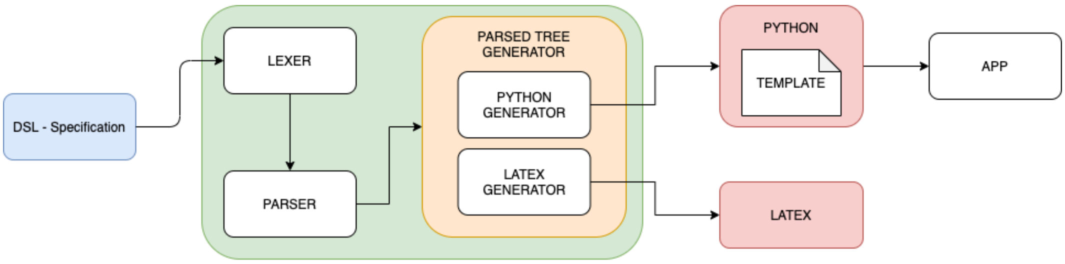



**Figure 2** System's architecture.

The input for our system, `DSL - Specification`, is a text file describing the behavior which should follow the language's standard. The compiler, which is represented as the green rounded rectangle in our architeture diagram, is composed by the following modules: a `Lexer`, a `Parser` and a `Parsed Tree Generator`. This generator has two sub-generators. The `Latex Generator`, that is responsible for the generation of the LaTeX representing the

[1] `https://unity.com`
[2] `https://github.com/gaia-ucm/jbt`
[3] `https://github.com/eykd/owyl`

behavior. And the other, `Python Generator`, generates the Python file representing the behavior according to a template, so that it can be later imported to an application.

## 3.1 BhTSL

Before we start describing the DSL, we will introduce a new node, called **Probability Selector** (figure 3), that will provide us with an extension to behavior specification.

A probability selector node is like a normal selector node, but instead of visiting its children from left to right, it visits them randomly, taking into account that each child has a probability, defined by the user, of being chosen first.

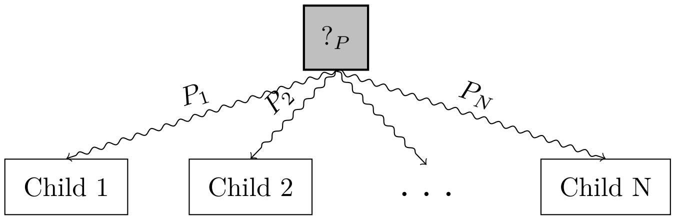



**Figure 3** Probability Selector node.

### 3.1.1 Syntax

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.

Up next, we have an example of a specification in our DSL:

```
behavior : [
    sequence : [
        condition : $cond1,
        condition : $cond2
        memory selector : [
            parallel : $par1,
            prob_selector : $prob1
        ]
    ]
]

parallel par1 : 10 [
    action : $action1,
    action : $action2
]

prob_selector prob1 : [
    $e1 -> decoraror : INVERTER [
        action : $action1
    ],
```

```
    $e2 -> action : $action2
]

%%

def action1(entity):
    pass

def action2(entity):
    pass

def cond1(entity):
    pass

def cond2(entity):
    pass

def e1(entity):
    pass

def e2(entity):
    pass
```

# 4 Tool development

## 4.1 Lexical analysis

The first step in the development of a compiler is the lexical analysis, that converts a char sequence in a token sequence. The tokens' table can be seen in the appendix.

## 4.2 Syntatic analysis

Syntatic analysis, or parsing, it the process of analyzing a string of symbols conforming the rules of a grammar. Here we have the grammar used in our DSL:

```
    root : behavior CODE
         | behavior definitions CODE
         | definition behavior CODE

    behavior : BEHAVIOR ':' '[' node ']'

    node : SEQUENCE ':' '[' nodes ']'
         | SEQUENCE ':' VAR
         | MEMORY SEQUENCE ':' '[' nodes ']'
         | MEMORY SEQUENCE ':' VAR
         | SELECTOR ':' '[' nodes ']'
         | SELECTOR ':' VAR
         | MEMORY SELECTOR ':' '[' nodes ']'
         | MEMORY SELECTOR ':' VAR
         | PROBSELECTOR ':' '[' prob_nodes ']'
         | PROBSELECTOR ':' VAR
         | MEMORY PROBSELECTOR ':' '[' prob_nodes ']'
         | MEMORY PROBSELECTOR ':' VAR
         | PARALLEL ':' INT '[' nodes ']'
```

```
          | PARALLEL ':' VAR
          | DECORATOR ':' INVERTER '[' node ']'
          | DECORATOR ':' VAR
          | CONDITION ':' VAR
          | ACTION ':' VAR

    nodes : nodes ',' node
          | node

    prob_nodes : prob_nodes ',' prob_node
               | prob_node

    prob_node : VAR RIGHTARROW node

    definitions : definitions definition
                | definition

    definition : SEQUENCE NODENAME ':' '[' nodes ']'
        | SELECTOR NODENAME ':' '[' nodes ']'
        | PROBSELECTOR NODENAME ':' '[' prob_nodes ']'
        | PARALLEL NODENAME ':' INT '[' nodes ']'
        | DECORATOR NODENAME ':' INVERTER '[' node ']'
```

## 4.3 Semantic analysis

Semantically, the compiler checks for non-declarable variables and variable redeclaration. A variable can only be accessed if it is declared, either in the *definitions* section (if it represents a control flow node), or in the *code* section (execution node). Additionally, a variable can only be declared one time, to avoid ambiguity in the memory access by the processor.

## 4.4 Code generator

The compiler can generate two different outputs: a LaTeX file, that provides a diagram for the BT specified; and a Python file, that contains the functions that implement the specified behavior.

## 4.5 Implementation

All the project was made using Python. For the compiler, we used the PLY (Python Lex-Yacc) library, which is an implementation of the lex and yacc parsing tools for Python. As for the generator, only some standard libraries were used.

Additionally, we created a LaTeX library, `behaviortrees.sty`, to draw the specified trees. This library is used in the LaTeX generator.

# 5 Example

In this section it will be presented an example to show how to specify a behavior in our language.

Suppose that, in our game, we want to have a guard that patrols a house. The guard has the following behavior: while he is patrolling, if he sees the player, activates an alarm and then, depending on the level of courage he has, decides (based on probabilities) whether he

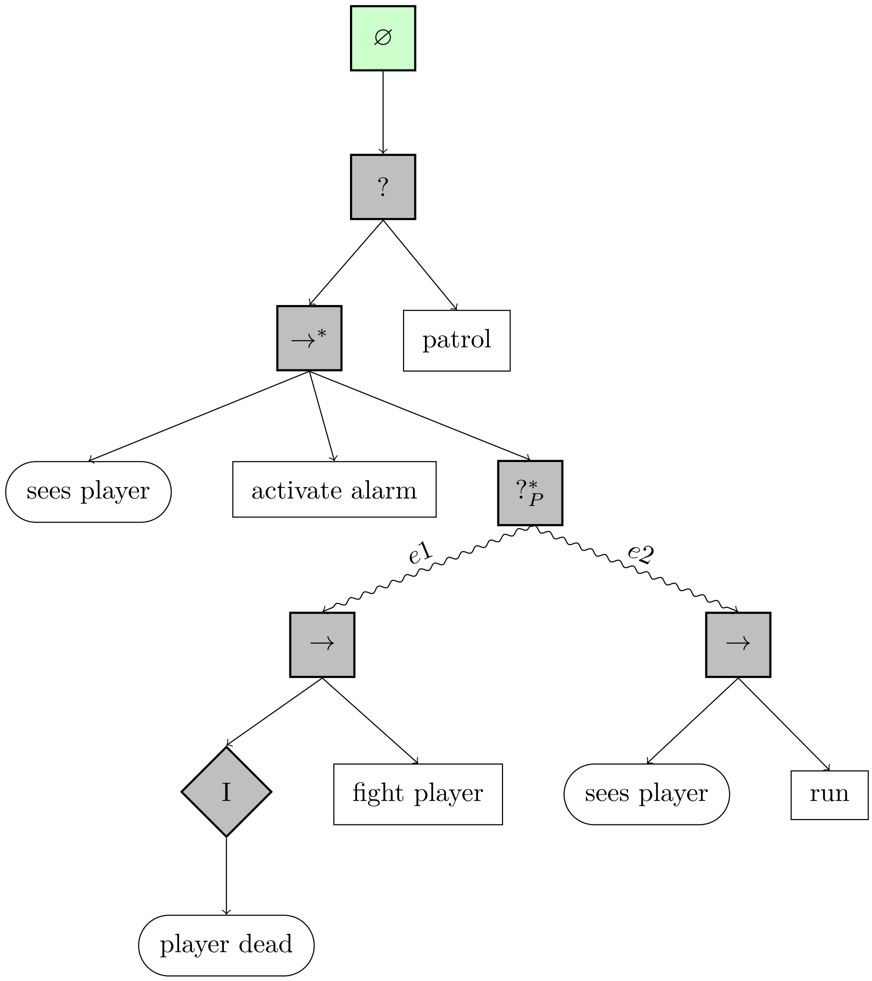



**Figure 4** Example.

runs away or fights the player. In case of running away, he constantly checks if he still sees the player, returning to patrolling in case he doesn't. If he still sees it, he keeps running. The same thing happens when he choses to fight the player, only this time he checks if the player is already dead or not.

An example specification for this behavior can be seen in the appendix B. Figure 4 shows the diagram of this specification after compiling it to LaTeX.

## 6 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** Epic-Games. Behavior trees, 2020. Accessed: 2020-05-21.

**3** Ian Millington and John Funge. *Artificial Intelligence for Games*. 01 2009. doi:10.1201/9781315375229.

4 Chris Simpson. Behavior trees for ai: How they work, 2014. Accessed: 2020-05-21.
5 Guillem Travila Cuadrado. Behavior tree library, 2018.

# A Tokens' table

The following table shows which tokens are utilized in our compiler:

| *Tokens* | |
|---|---|
| **Name** | **Value** |
| literals | `([]),:%` |
| RIGHTARROW | `->` |
| BEHAVIOR | `\bbehavior\b` |
| SEQUENCE | `\bsequence\b` |
| SELECTOR | `\bselector\b` |
| PROBSELECTOR | `\bprobselector\b` |
| PARALLEL | `\bparallel\b` |
| DECORATOR | `\bdecorator\b` |
| CONDITION | `\bcondition\b` |
| ACTION | `\baction\b` |
| INVERTER | `\bINVERTER\b` |
| MEMORY | `\bmemory\b` |
| INT | `\d+` |
| VAR | `$\w+` |
| NODENAME | `\b\w+\b` |
| CODE | `%%(.\|\n)+` |

**Table 1** Lexical analysis' tokens.

# B Example Specification

In this section it is shown the functions that are used in our example.

```
behavior : [
    selector : [
        memory sequence : [
            condition : $sees_player ,
            action : $activate_alarm ,
            memory prob_selector : [
                $e1 -> sequence : [
                    decorator : INVERTER [
                        condition : $player_dead
                    ] ,
                    action : $fight_player
                ] ,
                $e2 -> sequence : [
                    condition : $sees_player ,
                    action : $run
                ]
            ]
        ] ,
        action : $patrol
    ]
]
```

```
%%

def sees_player(patroller):
    from math import sqrt
    player_x = patroller['player']['x']
    player_y = patroller['player']['y']

    patroller_x = patroller['x']
    patroller_y = patroller['y']

    if sqrt(patroller_x ** 2 - player_x** 2 +
     patroller_y**2 - player_y**2) <= patroller['vision_radius']:
        return SUCCESS

    return FAILURE


def activate_alarm(patroller):
    print("ALARM ACTIVATED!!")
    patroller['alarm_activated'] = True
    return SUCCESS


def player_dead(patroller):
    if patroller['player']['hp'] == 0:
        return SUCCESS
    return FAILURE


def fight_player(patroller):
    patroller['player']['hp'] -= 10
    return RUNNING


def run(patroller):
    patroller['x'] = patroller['x'] - 10
    patroller['y'] = patroller['y'] - 10

    print("Running away from player!")
    return RUNNING


def patrol(patroller):
    patroller['x'] = patroller['x'] + 10
    patroller['y'] = patroller['y'] + 10
    return RUNNING


def e1(patroller):
    return patroller['guts'] / 10

def e2(patroller):
    return 1 - patroller['guts'] / 10
```