BT++
A Behavior Tree Library in C++
with a ROS wrapper in C++ and python

USER MANUAL

Michele Colledanchise

November 1, 2016

Contents

Chapter 1

Introduction

Welcome to the BT++ User Guide, a Behavior Tree library in C++.

Behavior Trees (BTs) were developed within the computer gaming community as a modular alternative to Finite State Machines (FSMs). Their recursive structure and usability have made them very popular in industry, which in turn has created a growing amount of attention in academia

The main advantage of BTs compared to FSMs is their modularity, as can be seen by the following programming language analogy. In FSMs, the state transitions are encoded in the states themselves, and switching from one state to the other leaves no memory of where the transition was made from, a so-called *one way control transfer*. This is very general and flexible, but actually very similar to the now obsolete *GOTO command*, that was an important part of many early programming languages, e.g., BASIC. In BTs the equivalents of state transitions are governed by calls and return values being passed up and down the tree structure, i.e. *two way control transfers*. This is also flexible, but more similar to the use of *function calls*, that has replaced GOTO in almost all modern programming languages. Using function calls when programming made it much easier to modularize the code, which in turn improved readability and reusability. Thus, BTs exhibit many of the advantages in terms of modularity (including readability and reusability) that was gained when going from GOTO to function calls in the 1980s. Note however, that there are no claims that BTs are superior to FSMs from a purely theoretical standpoint.

1.1 Behavior Trees

In this section, we will describe BTs in the classical way. To enable our analysis, we will then, in Section IV, provide a functional model of BTs. The classical description is included for comparison with the functional one. Let a BT be a directed tree, with nodes and edges, using the usual definition of parents and children for neighboring nodes. The node without parents is called the root node, and nodes without children are called leaf nodes. Now, each node of the BT is labeled as belonging to one of the six different type. Four control flow nodes: *Selector*, *Sequence*, *Parallel*, or *Decorator*. And two execution nodes: *Action* or *Condition*. Execution nodes the leaf nodes of the tree.

Upon execution of the BT, each time step of the control loop, the root of the

BT is *ticked*. This tick is then progressed down the tree according to the types of each node. Once a tick reaches a leaf node (Action or Condition), the node does some computation, possibly affecting some continuous or discrete states/-variables of the BT, and then returns either *Success*, *Failure* or *Running*. The return status is then progressed up the tree, back towards the root, according to the types of each node. If a running node does no longer receive a tick, it has to stop (preempted). The stopping of a preempted action is implemented in this library by the *halt* procedure. We will now describe how all the different node types handle the tick and processes the different return statuses.

Selector. Selector nodes are used to find and execute the first child that does not fail. A Selector node will return immediately with a status code of success or running when one of its children returns success or running. The children are ticked in order of importance, from left to right. The selector node is graphically represented by a box with a question mark, as in Fig. 1.1.

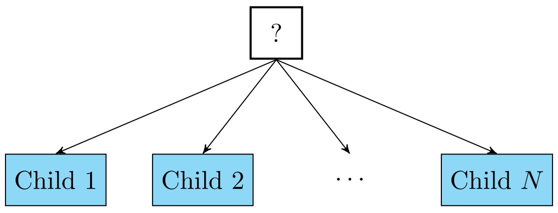

Figure 1.1: Graphical representation of a selector node with N children.

Sequence. Sequence nodes are used to find and execute the first child that has not yet succeeded. A sequence node will return immediately with a status code of failure or running when one of its children returns failure or running (see Figure II and the pseudocode below). The children are ticked in order, from left to right. The sequence node is graphically represented by a box with an arrow, as in Fig. 1.2.

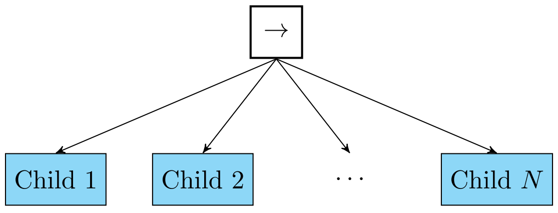

Figure 1.2: Graphical representation of a sequence node with N children.

Parallel. The parallel node ticks its children in parallel and returns success if $M \leq N$ children return success, it returns failure if $N - M + 1$ children return failure, and it returns running otherwise. The parallel node is graphically represented by a box with two arrows, as in Fig. 1.3.

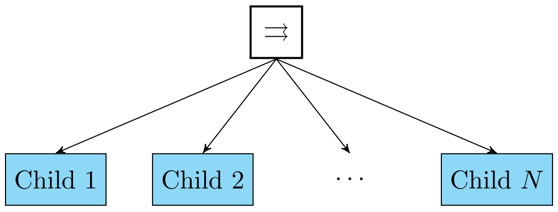

Figure 1.3: Graphical representation of a parallel node with N children.

Decorator. The decorator node manipulates the return status of its child according to the policy defined by the user (e.g. it inverts the success/failure status of the child). The decorator is graphically represented in Fig. **??**. In this library the decorators implemented are the two common ones: *Decorator Retry* which retries the execution of a node if this fails; and *Decorator Negation* That inverts the Success/Failure outcome.

Action. An Action node performs an action, and returns Success if the action is completed, Failure if it can not be completed and Running if completion is under way.

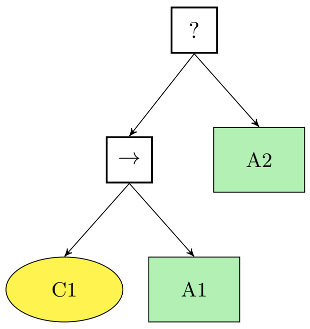

Figure 1.4: A Selector, a Sequence, a Condition (yellow) and two Actions (green). Action A1 is only performed when Condition C1 returns Success and Action A2 is only performed if C1 or A1 returns Failure.

Condition. A Condition node determines if a condition C has been met. Conditions are technically a subset of the Actions, but are given a separate category and graphical symbol to improve readability of the BT and emphasize the fact that they never return running and do not change any internal states/variables of the BT. Examples of Conditions can be found in Figure 1.4 below.

Non-Reactive Nodes There are cases in which a control flow node has to execute the feedforward execution of children (i.e. in a sequence node if Child i succeeded, the node executes Child $i + 1$ without checking if Child i is still in a success state). For these reason there exist the non-reactive counterparts of sequence and selector node. By convention we call them *SequenceStar* and *SelectorStar*. Figs. 1.5 and 1.6 shows an example of SequenceStar and SelectorStar.

However we do not advice the use of those non-reactive nodes as their execution can be encapsulated in a single action.

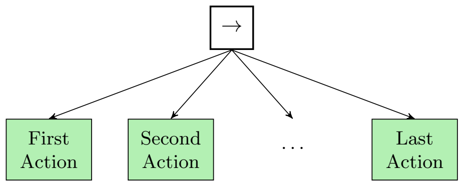

Figure 1.5: The SequenceStar execute its children in order. It returns failure when the current executing child returns failure. It return success if the last child returns success. It returns running otherwise.

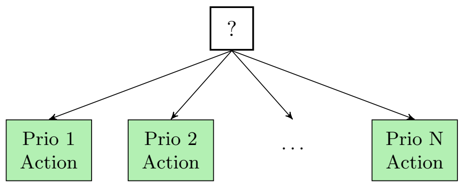

Figure 1.6: The SelectorStar its children in order. It returns success when the current executing child returns success. It return failure if the last child returns failure. It returns running otherwise.

Chapter 2

C++ Version

2.1 Installation

The first step to use BT++ is to retrieve its source code. You can either download it here () or clone the repository:

```
$ cd /path/to/folder
$ clone git@bitbucket.org:miccol/bt.git
```

Once you have the repository. You compile the library

```
$ cd /path/to/folder
$ mkdir ./build
$ cd build
$ cmake ..
$ make
```

Check the installation by running a BT of test.

```
$ ./samples/BTpp_example
```

2.2 Create an Action Node

The file *templates/cpp/ActionNodeTemplate.cpp* is a template on how the class of your action should look like. Edit the *Exec()* procedure, adding your code to execute when the node is TICKED . If the action is finished it should send the return status (Success/Failure) calling setStatus(Success/Failure). Below is the portion of code to be edited:

```
while(ReadState() == Running)
{
 /*
  HERE THE CODE TO EXECUTE AS LONG AS
  THE BEHAVIOR TREE DOES NOT HALT THE ACTION
 */



  //If the action Succeeded
    setStatus(Success);
 //If the action Failed
   setStatus(Failure);
}
```

The file *src/example_nodes/ActionNodeExample.cpp* provides and example.

2.3 Create a Condition Node

The procedure is similar to the one above. The file *templates/cpp/ConditionNodeTemplate.cpp* is a template on how the your class should look like when it checks a condition IMPORTANT! A condition is supposed to reply very fast. If it takes too much time (w.r.t the tick frequency), the node has to be defined as an action (condition nodes are not preemptable).

```
// Condition checking and state update
if (/*your_condition satisfied*/)
{
    SetNodeState(Success);
}
else
{
    SetNodeState(Failure);
}
```

The file *example_nodes/ConditionNodeExample.cpp* provides and example.

2.4 Create a Behavior Tree

To create the BT, edit the file *src/tree.cpp* or create a new one. Declare and allocate the BT nodes of your BT. To add a node as a child of a control flow node (Parallel/Selector/Sequence) you must use the *AddChild(TreeNode* node)* function. To execute the BT you must call the *Execute(Treenode root, int Tick-Period_milliseconds)* function. Below there is an example of a BT construction.

```
#include "BehaviorTree.h"

using namespace BT;

int main(int argc, char **argv)
{
 try
 {
 int TickPeriod_milliseconds = 1000;
 ActionTestNode* test1 = new ActionTestNode("A1");
 ActionTestNode* test2 = new ActionTestNode("A2");
 ActionTestNode* test3 = new ActionTestNode("A3");
 ActionTestNode* test4 = new ActionTestNode("A4");
 SequenceStarNode* seq1 = new SequenceStarNode("sq1");
 SelectorStarNode* sel1 = new SelectorStarNode("sl1");
 SelectorStarNode* sel2 = new SelectorStarNode("sl2");

 sel1->AddChild(test1);
 sel1->AddChild(test2);

 sel2->AddChild(test3);
 sel2->AddChild(test4);

 seq1->AddChild(selector1);
 seq1->AddChild(selector2);


 Execute(sequence1, TickPeriod_milliseconds);
 catch (BehaviorTreeException& Exception)
 {
 std::cout << Exception.what() << std::endl;
 }

return 0;
}
```

2.5 Run a Behavior Tree

Compile and run tree.cpp (or your executable)

```
$ cd path/to/folder
$ cd build
$ cmake ..
$ make
$ ./tree
```

While running, the BT is graphically showed in a window. To move the BT inside the window use the arrow keys. To increase/decrease the space between node use the *PageUp/PageDown* keys.

Chapter 3

ROS Version

3.1 Installation

The first step to use BT++ is to retrieve its source code. You can either download it here () or clone the repository:

```
$ cd /your/catkin_ws/src
$ clone git@bitbucket.org:miccol/bt.git
```

Once you have the repository. You compile the library

```
$ cd ~/your/catkin_ws/
$ catkin_make
```

Check the installation by running a BT of test.

```
$ roslaunch behavior_tree_leaves test_behavior_tree.launch
```

You can create action and condition nodes as in the C++ version or you can create *external* ROS nodes for action and condition nodes in both C++ and python.

3.2 Create an external ROS Action Node in C++

The file *behavior_tree_leaves/templates/cpp/ActionTemplate.cpp* is a template on how the your ROS node should look like it it performs and action (it is an action in the Behavior Tree). Your action is the Server and does stuff. The Behavior Tree is the Client and tells to all the Server which ones have to start (TICK) and which have to stop (HALT). Edit in the executeCB procedure, adding your code to execute when the node is TICKED and when is HALTED. If the action is finished the node should send the return status (SUCCESS/-FAILURE) calling setStatus(SUCCESS/FAILURE).

```
void executeCB(const bt_actions::BTGoalConstPtr &goal)
  {

    // publish info to the console for the user
    ROS_INFO("Starting Action");

    // start executing the action
    while(/*YOUR CONDITION*/)
    {

      if (as_.isPreemptRequested() || !ros::ok())
      {
    ROS_INFO("Action Halted");



        /*
        HERE THE CODE TO EXECUTE WHEN THE
        BEHAVIOR TREE DOES HALT THE ACTION
        */


    // set the action state to preempted
    as_.setPreempted();
    success = false;
    break;
      }


      ROS_INFO("Executing Action");
        /*
        HERE THE CODE TO EXECUTE AS LONG AS
        THE BEHAVIOR TREE DOES NOT HALT THE ACTION
        */

  //If the action succeeded
      setStatus(SUCCESS);
  //If the action Failed
      setStatus(FAILURE);
}
```

Then set a name for your action. The name has the be unique. It will be used by the behavior tree to recognize it.

```
   ros::init(argc, argv, /*name as a std::string*/);
```

The file *behavior_tree_leaves/example_nodes/cpp/ActionExample.cpp* provides an example.

3.3 Create an external ROS Action Node in python

The file *behavior_tree_leaves/templates/python/action_template.py* is a template on how the your ROS node should look like it it performs and action (it is an action in the Behavior Tree). Your action is the Server and does stuff. The Behavior Tree is the Client and tells to all the Server which ones have to start (TICK) and which have to stop (HALT). Edit the execute_cb procedure, adding your code to execute when the node is TICKED and when is HALTED. If the action is finished the node should send the return status (SUCCESS/FAILURE) calling setStatus(SUCCESS/FAILURE).

```
def execute_cb(self, goal):
    # publish info to the console for the user
    rospy.loginfo('Starting Action')
    # start executing the action
    while #your condition:
      if self._as.is_preempt_requested():
      #HERE THE CODE TO EXECUTE WHEN THE
      BEHAVIOR TREE DOES HALT THE ACTION
        rospy.loginfo('Action Halted')
        self._as.set_preempted()
        success = False
        break

    rospy.loginfo('Executing Action')
    #HERE THE CODE TO EXECUTE AS LONG AS
    THE BEHAVIOR TREE DOES NOT HALT THE ACTION

    #IF THE ACTION HAS SUCCEEDED
    self.set_status('SUCCESS')
    #IF THE ACTION HAS FAILED
    self.set_status('FAILURE')
```

Then set a name for your action. The name has the be unique. It will be used by the behavior tree to recognize it.

```
   ros::init(argc, argv, /*name as a std::string*/);
```

The file *behavior_tree_leaves/example_nodes/python/action_example.py* provides an example.

3.4 Test your external ROS action node outside the Behavior Tree

You may want to test you external ROS node independently from the BT. This is possible using the executable *ActionClient*.

Run your external ROS action node and then run the ActionClient.

```
rosrun behavior_tree_leaves ActionClient
rosrun behavior_tree_leaves YOURACTIONNAME
```

then follow the instructions of *ActionClient*

3.5 Create an external ROS Condition Node in C++

The procedure is similar to the one above.

The file *behavior_tree_leaves/templates/cpp/ConditionTemplate.cpp* is a template on how the your ROS node should look like when it checks a condition. Your condition is the Server and checks stuff. Edit the executeCB procedure, adding your code to execute when the node is TICKED (it is never halted). If the condition is satisfied then the condition returns the status setStatus(SUCCESS). If the condition is not satisfied then the condition returns the status setStatus(SUCCESS). IMPORTANT! A condition is supposed to reply very fast. If it takes too long, the node has to be defined as an action.

```
void executeCB(const bt_actions::BTGoalConstPtr &goal)
      {
         if(/*condition  satisfied*/)
                  {
            setStatus(SUCCESS);
         }else{
            setStatus(FAILURE);
         }
      }
```

Then set a name for your condition. The name has the be unique. It will be used by the behavior tree to recognize it.

```
   ros::init(argc, argv, /*name as a std::string*/);
```

The file *behavior_tree_leaves/example_nodes/cpp/ConditionExample.cpp* provides an example.

3.6 Create an external ROS Condition Node in python

The procedure is similar to the one above. The file *behavior_tree_leaves/templates/python/condition_template.py* is a template on how the your ROS node should look like when it checks a condition. Your condition is the Server and checks stuff. Edit the execute_cb procedure, adding your code to execute when the node is TICKED (it is never halted). If the condition is satisfied then the condition returns the status setStatus(SUCCESS). If the condition is not satisfied then the condition returns the status setStatus(SUCCESS). IMPORTANT! A condition is supposed to reply very fast. If it takes too long, the node has to be defined as an action (condition nodes are not preemtable).

```
def execute_cb(self, goal):
  # publish info to the console for the user
  rospy.loginfo('Checking Condition')
   if #YOUR CONDITION
       #IF THE CONDITION IS TRUE
       self.set_status('SUCCESS')
  else
       #IF THE CONDITION IS FALSE
       self.set_status('FAILURE')
```

Then set a name for your condition. The name has the be unique. It will be used by the behavior tree to recognize it.

```
   rospy.init_node(#name as a string#);
```

The file *behavior_tree_leaves/example_nodes/python/condition_example.py* provides an example.

3.7 Test your external ROS condition node outside the Behavior Tree

You may want to test you external ROS node independently from the BT. This is possible using the executable *ConditionClient*.

Run your external ROS condition node and then run the ConditionClient.

```
rosrun behavior_tree_leaves ConditionClient
rosrun behavior_tree_leaves YOURCONDITIONNAME
```

then follow the instructions of *ConditionClient*.

3.8 Create a Behavior Tree

To create the BT, edit the file *src/main.cpp*. Declare and allocate the BT nodes of your BT. To add a node as a child of a control flow node (Parallel/Selector/Sequence) you must use the *AddChild(TreeNode* node)* function. To execute the BT you must call the *Execute(Treenode root, int TickPeriod_milliseconds)* function. Below there is an example of a BT construction.

```
#include "BehaviorTree.h"

using namespace BT;

int main(int argc, char **argv)
{
    ros::init(argc, argv, "BehaviorTree");
    try
    {
        int TickPeriod_milliseconds = 1000;

        ROSAction* act1 = new ROSAction("Act");
        ROSCondition* con1 = new ROSCondition("Cond");


         SequenceNode* seq1 = new SequenceNode("sq1");

        seq1->AddChild(con1);
        seq1->AddChild(act1);

        Execute(seq1, TickPeriod_milliseconds);

}
    catch (BehaviorTreeException& Exception)
    {
        std::cout << Exception.what() << std::endl;
    }

return 0;
}
```

3.9 Execute your ROS Behavior Tree

To run you behavior tree, you must run all the leaf ROS nodes and the tree

```
$ rosrun behavior_tree_leaves LEAF1NAME
$ rosrun behavior_tree_leaves LEAF2NAME
$ rosrun behavior_tree_core TREENAME
```

or alternatively you can create a launch file in the folder *behavior_tree_core/launch*. Below there is a launch file of example.

```
<launch>
  <node name="action" pkg="behavior_tree_leaves" type="ActionExample" />
  <node name="condition" pkg="behavior_tree_leaves" type="ConditionExample" />
  <node name="tree" pkg="behavior_tree_core" type="tree" />
</launch>
```

Acknowledgment

We thank Rocco Santomo for his valuable help in developing the early version of this library. We also thank Ewerton Lopes and Diogo Almeida for their feedback and suggestions.