at_mission_planner: Development Documentation

The goal of this document is to explain the basic tools that we use in the mission planning node to create behavior trees (BT). It will explain the most commonly used pieces of the py-trees and py-trees-ros libraries, giving examples of their use cases. This document does not intend to explain the design patterns behind BTs, and will not impose any exact method of designing trees. It will simply explain the pieces you should be working with so that you can plan and design trees yourself.

This document assumes you have a basic understanding of what a BT is; you should understand that a BT is made up of nodes and structures to control those nodes. You should also understand the concept of “ticking” a BT.

I also will clarify here: when discussing BTs, I use the term **node** to refer to child behaviors that implement the actual functionality of the tree. This means I am referring to the “leaves” of the tree. Do not confuse this with a ROS node that runs publishers, subscribers, services, etc.

A Brief Introduction to Py-Trees

The mission planner node makes use of the py-trees (here) library, and the ros extensions to this library. The documentation linked here is a very important reference to use when building trees, and I recommend reading it if you want more detail. I will give a brief overview of the most important structures here.

Sequences

Sequences in py-trees allow you to sequentially execute nodes. A sequence will execute its children in order until either:

1. One of the children fails, at which point the sequence will fail
2. All of its children succeed, at which point the sequence will succeed

Sequences can be created with or without memory. If one of the nodes in the sequenced returns RUNNING,

- A sequence with **memory enabled** will remember what child was RUNNING during the previous tree tick, and will continue to tick that child on the next tick.
- A sequence with **memory disabled** will re-tick all of its children each tick, and thus start from the first child again.

Both of these can be useful in the right situation.

Selectors

Selectors in py-trees allow you to manage priorities, and create fallback behaviors for nodes. A selector will execute its children in order until either:

1. One of its children succeeds, at which point the selector will succeed
2. All of its children fail, at which point the selector will fail.

Similar to sequences, selectors also can be created with or without memory. If memory is disabled, higher priority checks will be re-checked each time. With memory enabled, once a priority is selected, it will be pursued until it either fails or succeeds.

Since a sequence will return as soon as one child succeeds, it can be used for fallback behavior. The highest priority action should be put as the first child, and later children will be fallback behaviors that we can do if our high priority action fails.

Parallels

Parallels in py-trees have a form of pseudo-parallelism that allow you to tick multiple nodes “at the same time” (in reality, they are not ticked at exactly the same time, but they will both be ticked during the same tree tick).

Parallels have three success policies:

1. SuccessOnAll: the parallel will only return success if all children have succeeded
2. SuccessOnOne: the parallel will return success as long as at least one child has succeeded
3. SuccessOnSelected: the parallel will return success as long as the specifically selected children all have succeeded

If using SuccessOnAll or SuccessOnSelected, we can choose to *synchronize* children so that once a child returns success, it will not be ticked again until the policy criteria is met.

Decorators

Decorators in py-trees can be thought of as the “hats” that you can put on a node to affect its functionality. There is a long list of them that can be found here, and I will not describe each in detail. Most of them are fairly self explanatory. Some examples include SuccessIsFailure (any success returned will be turned into a failure), EternalGuard (a check that will be run on the node every time it is ticked, and the node will return failure if the check fails), and Timeout (give a time limit to a node, fail if the time limit exceeds).

The Blackboard

The blackboard is extremely important for BTs, as it provides a proxy for passing information between nodes. It essentially acts as a giant dictionary that nodes can request read/write permissions to, and they can then write/read data given a key and/or a value.

We commonly use the blackboard, since we often want to pass data from ROS topics to behavior nodes (BN), and it is often easiest to do so through the blackboard.

Creating Nodes Using Py-Trees and Py-Trees-ROS

When working with the mission planner, you will need to define individual nodes that can be connected with composite structures. There are two main ways to do this:

1. Create a fully custom BN
2. Use the BNs in the py-trees-ros library.

Creating a Custom Behavior Node (BN)

The py-trees documentation for this here is very good for this step. I will give a brief summary.

When you create a node, you will need to extend the py_trees.behaviour.Behaviour class. This will result in you having the following methods you can overload:

1. **__init__()**: This is where you will want to do one time initialization, like reading in the arguments to the node.
2. **setup()**: This is called only once. When the entire BT is being set up, a setup() call will cascade down to every node, and will be called on your behavior. This is where you will want to set up any ROS publishers/service clients so that they exist when the tree starts running.
3. **initialize()**: This is called *the first time your behavior is ticked, and any time it is not running thereafter*. Essentially, this is called on the first tick of the node, and will be called again if the node is ticked after it has already succeeded or failed (i.e., we are repeating some execution and the node will be ticked again).
    a. This is where you will want to start your node's actual functionality, or reset state variables that are used when updating your node.
4. **update():** This is called when your behavior is ticked. You should do any status updating necessary, and return a node status: either SUCCESS, RUNNING, FAILURE, or INVALID depending on what information you see during the update.
    a. As an example, this is where you would check to see if a service call result has come back, or update
    b. Make sure you do not block in this function
5. **terminate()**: this is called whenever your node switches to a non running state: either SUCCESS, FAILURE, or INVALID. This is where you should do any required node cleanup.

To be clear, **not all of these are required in every behavior**. Most behaviors will require a setup, initialize, and update step, but many do not require any terminate step. __init__ is mandatory.

Using the Behaviour Nodes in the py-trees-ros Library

Luckily for us, many of the node types we would frequently need already exist for us in the py-trees-ros extension to py-trees. The module API can be found here, and for most purposes, we prefer to use already existing nodes in this library when possible.

Some of the things this library already has created for us are:

1. A BN that creates a ROS action client and updates based on the action
2. BNs to read information from ROS topics and move it to the blackboard where other nodes can use it

Some of the things the library unfortunately does not provide for us:

1. A BN to publish to a ROS topic
2. A BN for ROS service clients

Interfacing with ROS

To interface with ROS, we use the py-trees-ros nodes as well as some custom nodes. Details for how to do this for publishers, subscribers, actions, and services follows below.

When discussing this, it is important to understand how py-trees-ros works with ROS in general. When we create a BT in py-trees-ros, it can be passed a ROS node to use for any ROS functionality within the tree. All nodes of the BT then **share this same ROS node** for any publishers, actions, etc. To do this, the nodes need to retrieve the ROS node from the tree, which can be done with the following code in the setup() function:

py-trees-ros built in behaviors will do something like this quietly under the hood.

Publishers

Since py-trees-ros does not provide any helper nodes for publishing data, we usually need to write behaviors that publish data ourselves. We do not often need to do this, but it is quite simple when we do. We can simply create a publisher in the setup() function, and publish the data in the initialize() function. Then, on update() we can essentially just return success.
Example file: behaviours/drop_dropper.py

Subscribers

Subscribers are the most complicated piece to interface with using py-trees. To explain why, consider how subscribers work in ROS. They will pick a topic to listen to, and create a callback

function that can be run whenever new data is found on the topic. However, when you think about this in a py-trees context, there is a bit of a complication: what happens if we get new data when the node is not being ticked? What happens if we don't get any data when the node is being ticked?

While there are ways to work around this, we typically find it easiest to just interface directly with the already existing py-trees-ros subscriber functionalities. We can then pass any information to the blackboard so other nodes can access it. py-trees-ros has a built-in subscriber-to-blackboard node that we use for this.
Example file: components/data_gathering.py

Actions

py-trees-ros has a built in action node that serves our purposes very well. You simply need to pass in the action topic, type, and goal. There are two methods of passing in the goal:

1. Pass a constant goal
2. Pass a blackboard variable that contains the goal

If we know the goal will always be the same, it is easier to use a constant. If the goal might depend on some runtime calculation, you will have to pass it through a blackboard variable first. To do this, just create a BB var that contains the Goal() object type you need, and pass the name to the node.
Example file: components/movement.py

Services

The structure for services is very similar to publishers. In setup() we initialize a service client, in initialize() we make the request. The only difference is that now, in the update() method, we should check if the service has returned before we return success. If the service has any information we need to check in its response body, we can do that in update() as well.
Example file: behaviours/swap_vision_model.py

Developing for the Mission Planner: Project Structure

The structure of the mission planner attempts to make the code easy to re-use between missions. We also want to abstract away the details of low level sections of the tree so that you do not need to understand the implementation details of a movement node to write a mission.

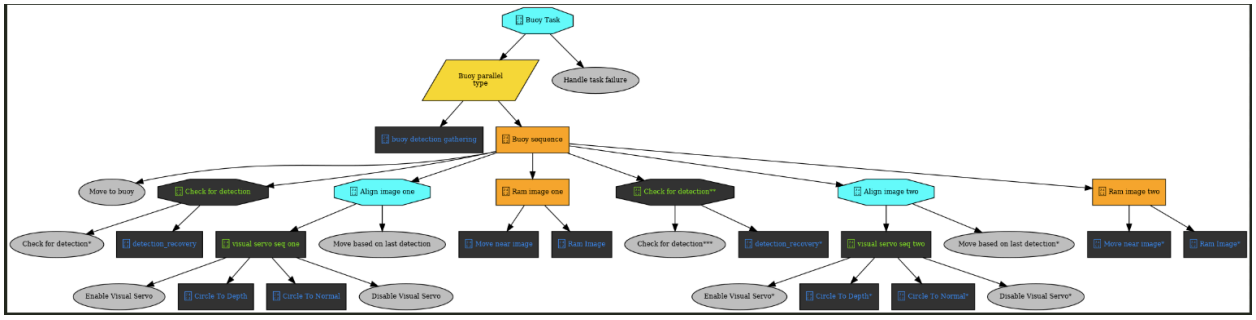
We use the following folder structure:

- **mission_planner_node.py**: the main ROS node for the mission planner. This runs some overhead to make each tree run only once. For the most part, this will not change.
- **\behaviours** - This is where we create behaviour nodes directly. The folder houses all the individual behaviour nodes that we need. These will be used to create components and missions.

- **\components** - this folder is where we create our layers of abstraction. We implement features here as functions that take in parameters and return a small behaviour tree that accomplishes a small, specific task (such as sending a movement command).
    - We organize these functions into files that generally serve a similar purpose.
    - While each function in these files DOES return a behaviour tree, these trees CANNOT be directly run from the node.
- **\missions** - this folder houses the actual, runnable mission files.
    - To create a mission, create a file here. Then, in the __init__.py file in the folder, add the line `from .[mission_name] import *` to allow your mission to be run as a launch argument.
    - The file must have a function with the same name as the file. One mission per file only.

You can think of the folders as organizing into “size”: behaviours are the smallest, as they are individual nodes. Components are bigger - they aggregate behaviours into small, specific trees. Missions are the largest - they aggregate components and behaviours into large trees that accomplish a task. **Behaviours** → **Components** → **Missions**, where each layer will use the layers below it.

Rendering to Visualize your Trees



We use the [py-trees-render program] to render our trees. It generally is run in the following way:

```
py-trees-render -l [detail level] [folderpath to file].[file name].[function name]
```

This will generate a .png, .svg, and .dot file in the folder you run the command that will contain the tree.

Trees can get quite large, and many of the fine details of the tree are not important to view. For this reason, we use a feature called “blackboxing” to collapse nodes in the images into “boxes” that abstract away the unnecessary information.

To do this, we need to give every function in **components** a blackbox level. This is done in the code, and can be seen in all component files. Then, when we render, we can specify a detail level. Any components with detail levels at or below the level given in the command will be collapsed. Most components use the “DETAIL” level.