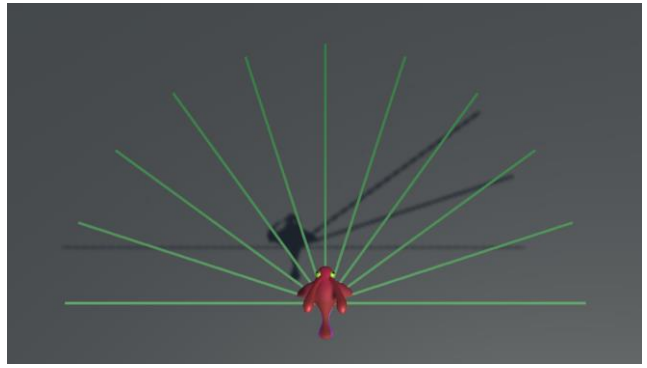
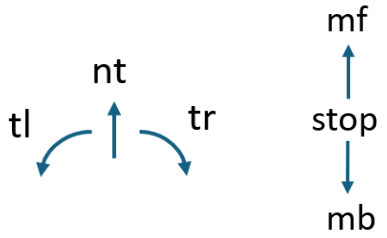
AAPE
Behaviour Trees

The Critter agent



Sensors: Ray cast sensor (configurable)
Accessible through the class RayCastSensor
Actions:

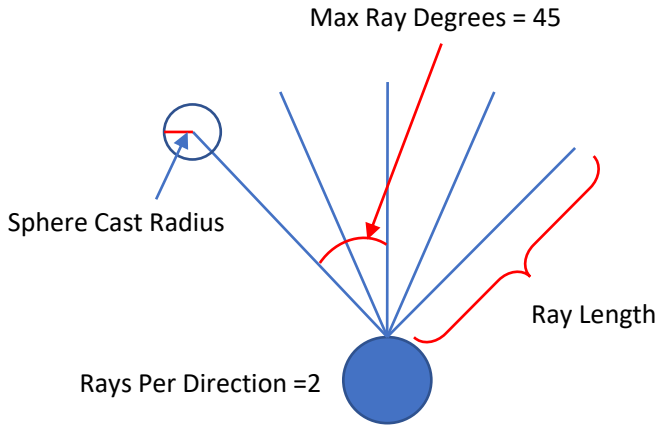
Movement actions:
Rotation (“tl” -> turn left, “nt” -> no turn, “tr” -> turn right
Translation (“mf” -> move fordward, “stop” -> stop, “mb” -> move backward)




```
{
  "Server": {
    "host": "127.0.0.1",
    "port": 4649
  },
  "AgentParameters": {
    "name": "Critter_1",
    "type": "AAgentCritterMantaRay",
    "spawn_point": 0,
    "debug_mode": true,
    "ray_perception_sensor_param": [5,90,0,5]
  }
}
```

[“Rays Per Direction”, “Max Ray Degrees”, Sphere Cast Radius”, “Ray Length”]

Rays Per Direction: Number of rays on each side of the central one.
Max Ray Degrees: Angle between the central ray and the leftmost or righmost ray.
Sphere Cast Radius: Size of the sphere at the end of the ray.
Ray Length: Length of the rays



The Critter agent

Files:

Aagent_BT.py – Is the agent class. It instantiates the agent, deals with the connection with Unity and executes the main loop.

AAgent-X.json – Configuration file. One configuration file for each agent.

Sensors.py – Deals with the information related to the sensors.

Goals_BT.py – Implements the goals of the agent.

A new file

YourBehaviourTree.py

Goals_BT.py and YourBehaviourTree.py is where you have to put your code. Now you will not call (although you still can)

goal:Behaviour

but

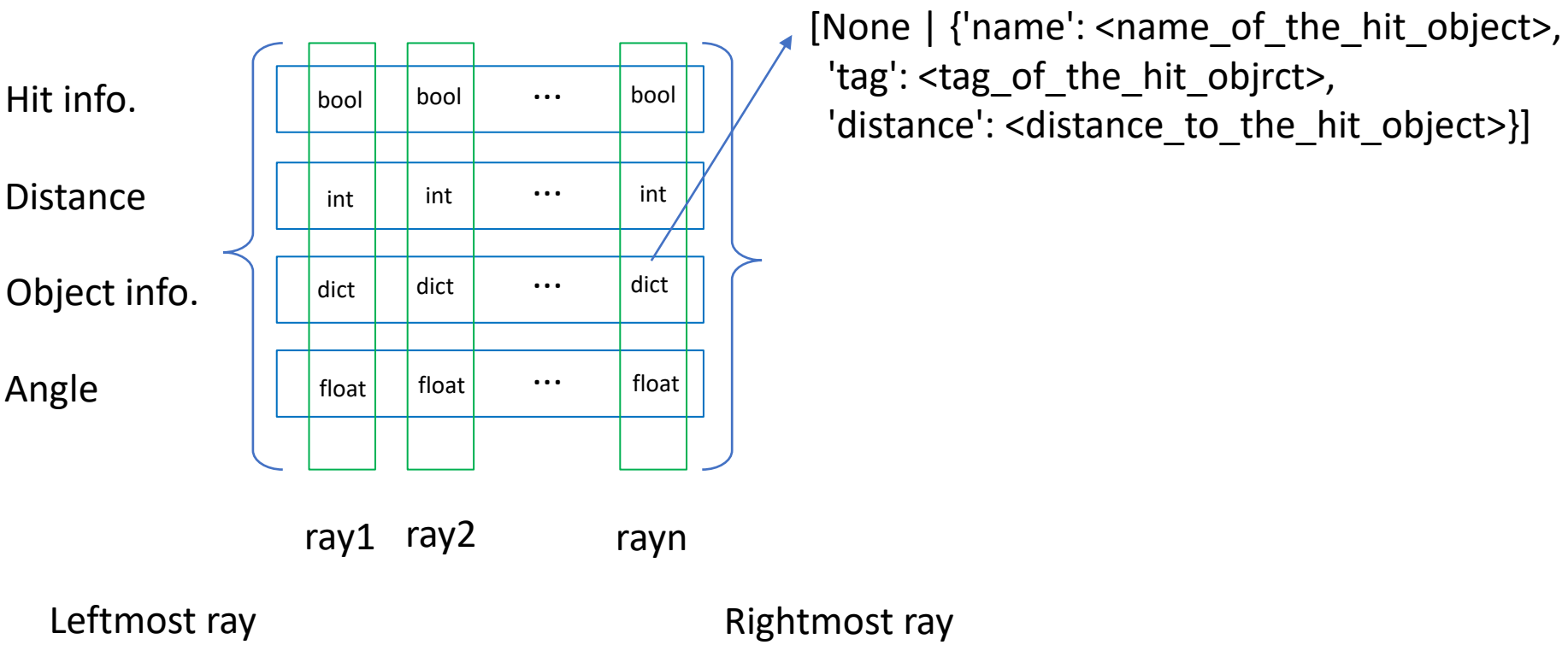
bt:YourBehaviourTree

```
# Reference to the possible goals the agent can execute
self.goals = {
    "DoNothing": Goals_BT.DoNothing(self),
    "ForwardDist": Goals_BT.ForwardDist(self, -1, d_min: 5, d_max: 10),
    "Turn": Goals_BT.Turn(self)
}
# Active goal
self.currentGoal = "DoNothing"

# Reference to the possible behaviour trees the agent ca execute
self.bts = {
    "BTRoam": BTRoam.BTRoam(self)
}
```

Class RayCastSensor

- You access the information of the ray cast sensor through the property self.rc_sensor.
- It is an array where the columns represent each ray cast going from left to right and the rows contain information about if the ray is hitting an object, the distance with the object, information about the object and the angle of the ray with the object.



Class InternalState

- You access the information of the internal State of the agent through the property self.i_state.

currentActions: list<string> Actions the agent is currently executing [“W” | “D” | “A” | “S” | “Z”]

Notice that you can have more than one action active, for example [“W”,”D”] (going forward and turning right at the same time)

isRotatingRight: <bool>

isRotatingLeft: <bool>

movingForwards: <bool>

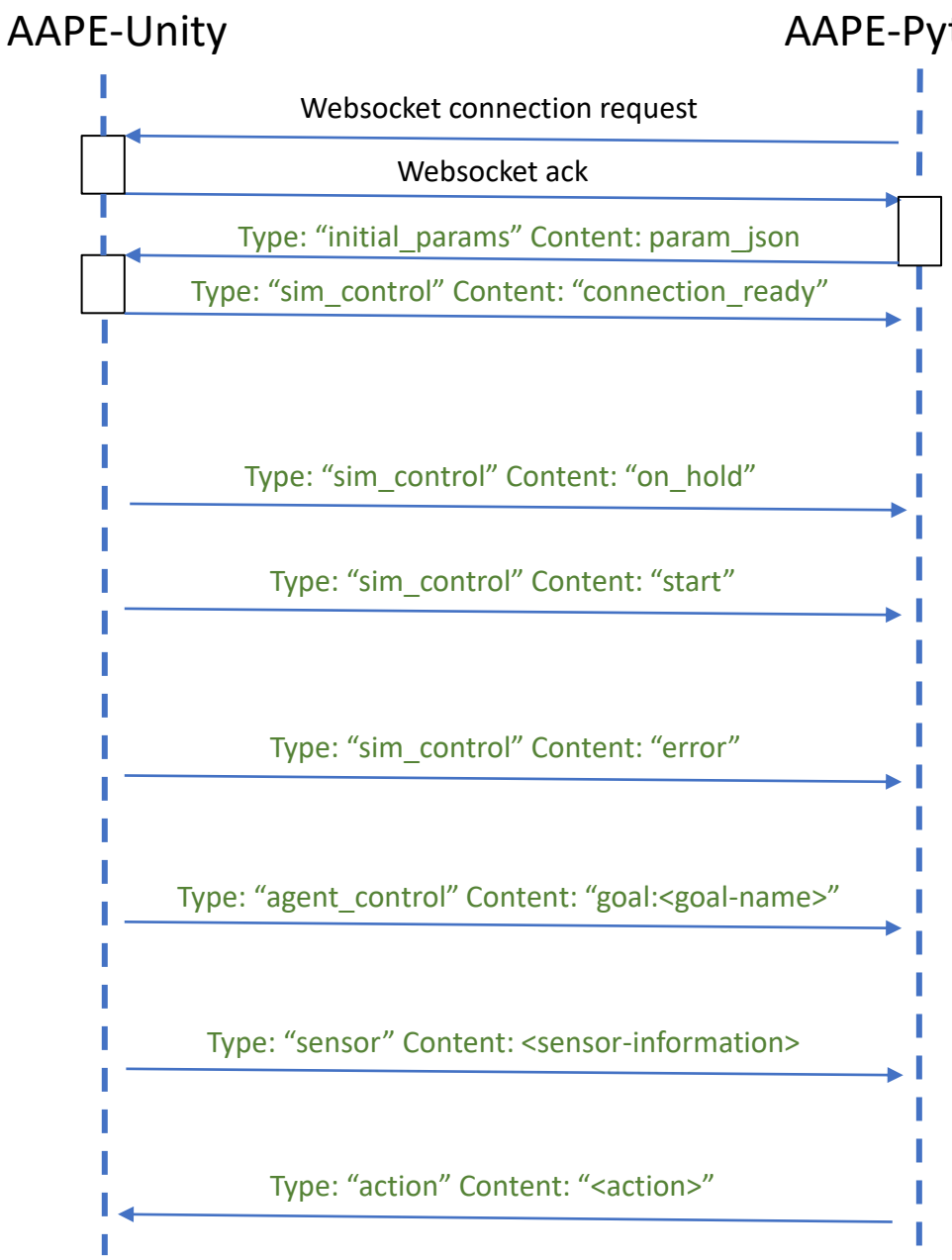
movingBackwards: <bool>

speed: <float> Agent speed

position: <dict { "x": <float>, "y": <float>, "z": <float> } Position using world coordinates

rotation: <dict { "x": <float>, "y": <float>, "z": <float> } Rotation y - Yaw, x - Pitch, z - Roll

Interaction diagram



Connection handshake

The simulation is on hold

The simulation has started

Error creating the agent in AAPE-Unity

Goal execution

Sensor information

Action request

Goals

- ~~A new goal is defined as a class that inherits from the Goal class.~~ This is no longer necessary

```
self.a_agent -> Reference to the AAgent parent class.
self.a_agent.rc_sensor-> Access to the ray cast sensor information (reference to
                         an object of type RayCastSensor defined in Sensors.py).
self.a_agent.i_state -> Access to the internal state of the agent.
```

- ~~Also, you can rewrite the following methods:~~
  - ~~`__init__(self, a_agent)`~~
  - ~~`update(self)`~~
- self.a_agent will be used to call the send_message method. Because in this part of the code we are requesting for actions, the call will have the form:

  await self.a_agent.send_message("action", "<action_to_perform>")

  Where <action_to_perform> is a string with one of the possible actions [tl, tr, nt, mf, mb, stop]. For example, to move forward you will send the message self.a_agent.`send_message("action","mf")`. To turn to the right `self.a_agent.send_message("action","tr")`, and so on. **You can pass a single action per message.**

Goals

- Because the actions requested are not performed immediately (for example, turning the agent takes some time) we have to keep track of those actions already requested and actions currently running to avoid asking for already requested/executing actions.
- Methods **requested(self, action)**, **executing(self, action)** and **update_req_actions(self)** in class **Goal** are used to manage that tracking.

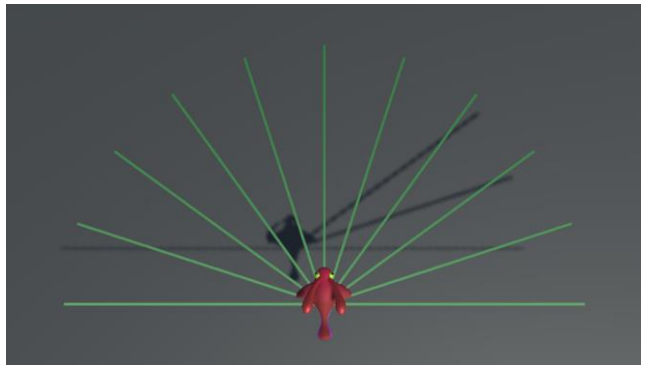
Example of a goal DoNothing

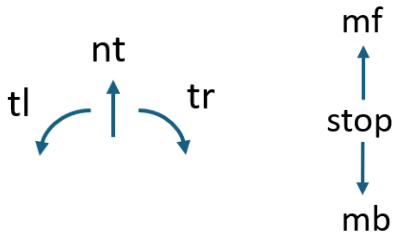
```
class DoNothing(Goal):
    """
    Does nothing
    """
    def __init__(self, a_agent):
        super().__init__(a_agent)

    async def update(self):
        await super().update()
        print("Doing nothing")
        await asyncio.sleep(1)
```

NOT APPLICABLE ANYMORE. However…

You explicitly have to stop a turning right of left (“nt”), and the same with moving forwards or backwards (“stop”).





Behaviours

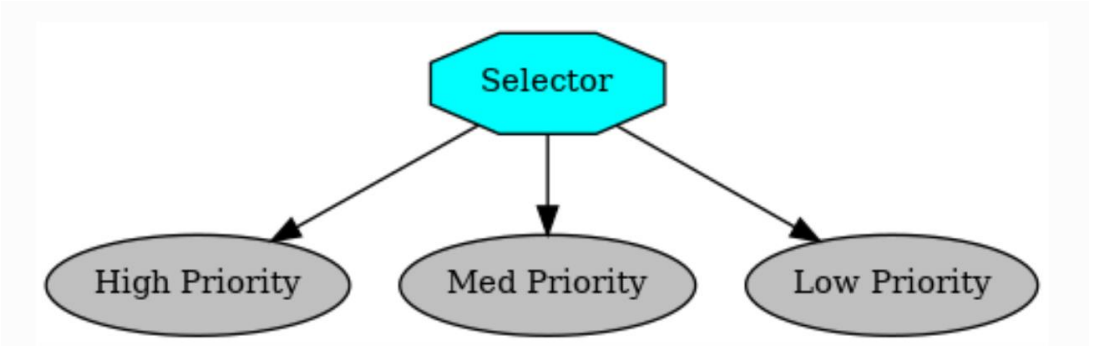
Behaviours: https://py-trees.readthedocs.io/en/devel/behaviours.html

Composites

Composites: https://py-trees.readthedocs.io/en/devel/composites.html

PyTree Composites

Selector

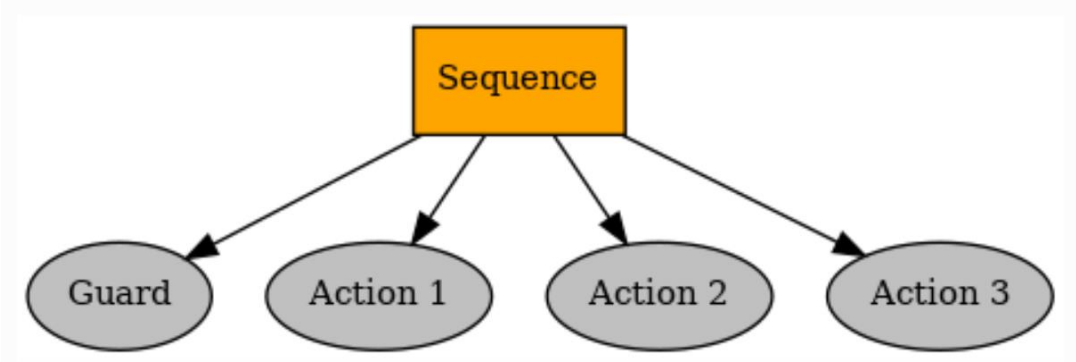


A selector executes each of its child behaviours in turn until one of them succeeds (at which point it itself returns RUNNING or SUCCESS, or it runs out of children at which point it itself returns FAILURE. We usually refer to selecting children as a means of choosing between priorities. Each child and its subtree represent a decreasingly lower priority path.

Switching from a low -> high priority branch causes a stop(INVALID) signal to be sent to the previously executing low priority branch. This signal will percolate down that child's own subtree. Behaviours should make sure that they catch this and destruct appropriately.

If configured with **memory**, higher priority checks will be skipped when a child returned with running on the previous tick. i.e. once a priority is locked in, it will run to completion and can only be interrupted if the selector is interrupted by higher priorities elsewhere in the tree.

Sequence

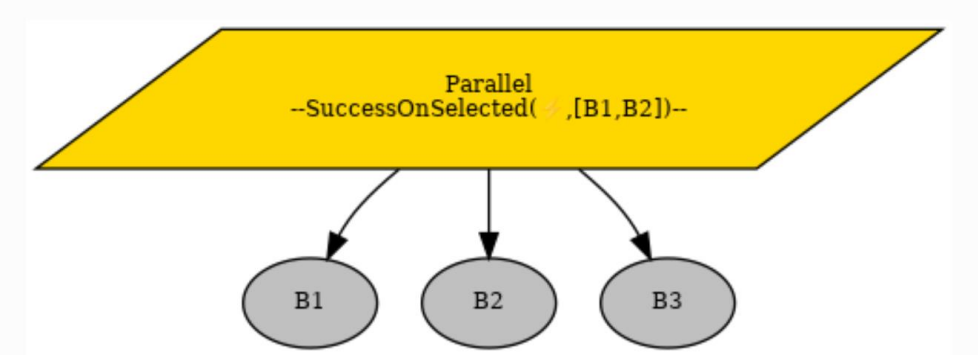


A sequence will progressively tick over each of its children so long as each child returns SUCCESS. If any child returns FAILURE or RUNNING the sequence will halt and the parent will adopt the result of this child. If it reaches the last child, it returns with that result regardless.

The sequence halts once it engages with a child is RUNNING, remaining behaviours are not ticked.

If configured with memory and a child returned with running on the previous tick, it will proceed directly to the running behaviour, skipping any and all preceding behaviours. With memory is useful for moving through a long running series of tasks. Without memory is useful if you want conditional guards in place preceding the work that you always want checked off.

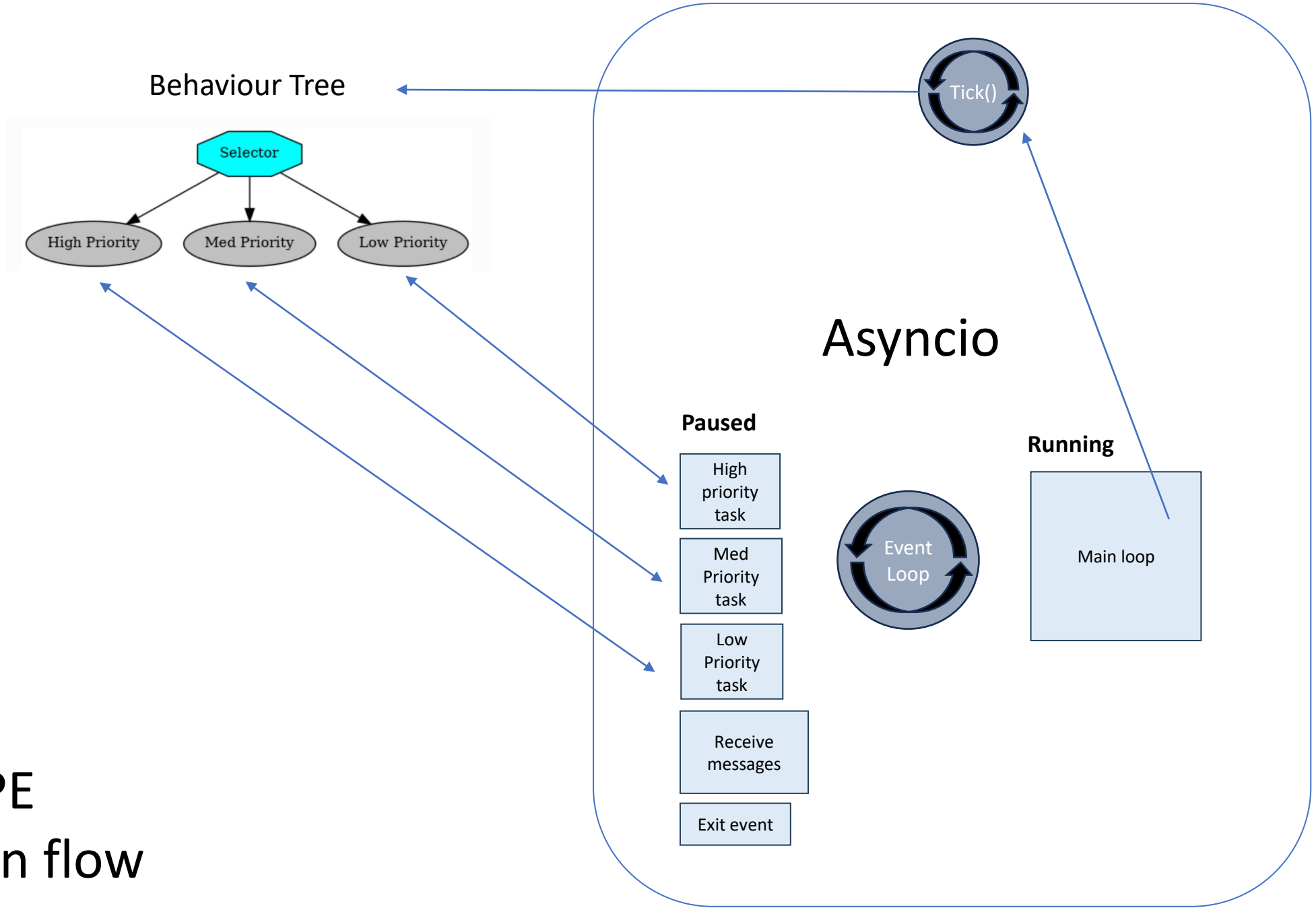
Parallel



A parallel **ticks every child every time the parallel is itself ticked.** The parallelism however, is merely conceptual. The children have actually been sequentially ticked, but from both the tree and the parallel's purview, all children have been ticked at once.

The parallelism too, is not true in the sense that it kicks off multiple threads or processes to do work. Some behaviours may kick off threads or processes in the background, or connect to existing threads/processes. The behaviour itself however, merely monitors these and is itself enclosed in a py_tree which only ever ticks in a single-threaded operation.

Parallels will return FAILURE if any child returns FAILURE
Parallels with policy **SuccessOnAll** only returns SUCCESS if all children return SUCCESS
Parallels with policy **SuccessOnOne** return SUCCESS if at least one child returns SUCCESS and others are RUNNING
Parallels with policy **SuccessOnSelected** only returns SUCCESS if a specified subset of children return SUCCESS



AAPE execution flow

Examples

- We want our critter to perform these behaviours:

1 – Turns, move, does nothings in a loop.

2 – The same but better looking…

3 - Goes roaming around, moving straight, turning... until it finds a flower. At that point, it stops and remains like this forever.

You will find the examples in the provided code (BTRoam.py). They are named VERSION 1, 2 and 3 respectively.