

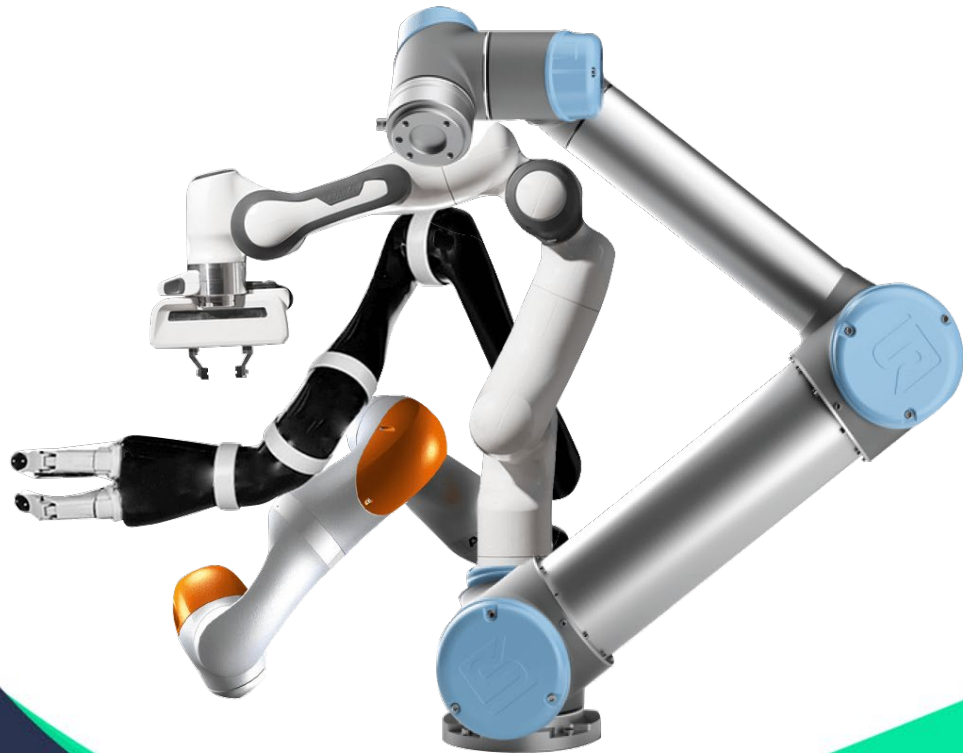


BehaviorTree.CPP 4.0

New features and how they make your life easier

October 21st, 2022

Davide Faconti
Staff Engineer
davide.faconti@picknik.ai



⋮ About me



Davide Faconti, nice to meet you

Working at Picknik (but not on Behavior Trees)

Almost 20 years in robotics doing:

- Humanoid robots design
- Bipedal Dynamic Walking
- Low hardware drivers
- Navigation and Localization
- Perception
- Manipulation
- Task planning
- Tooling and monitoring
- System architectures
- Project Management
- Product development
- But....

⋮ About me

Davide Faconti,

Working at Pick

Almost 20 years

- Humanoid

- Bipedal D

- Low hard

- Navigation

- Perception

- Manipulation



Monitoring

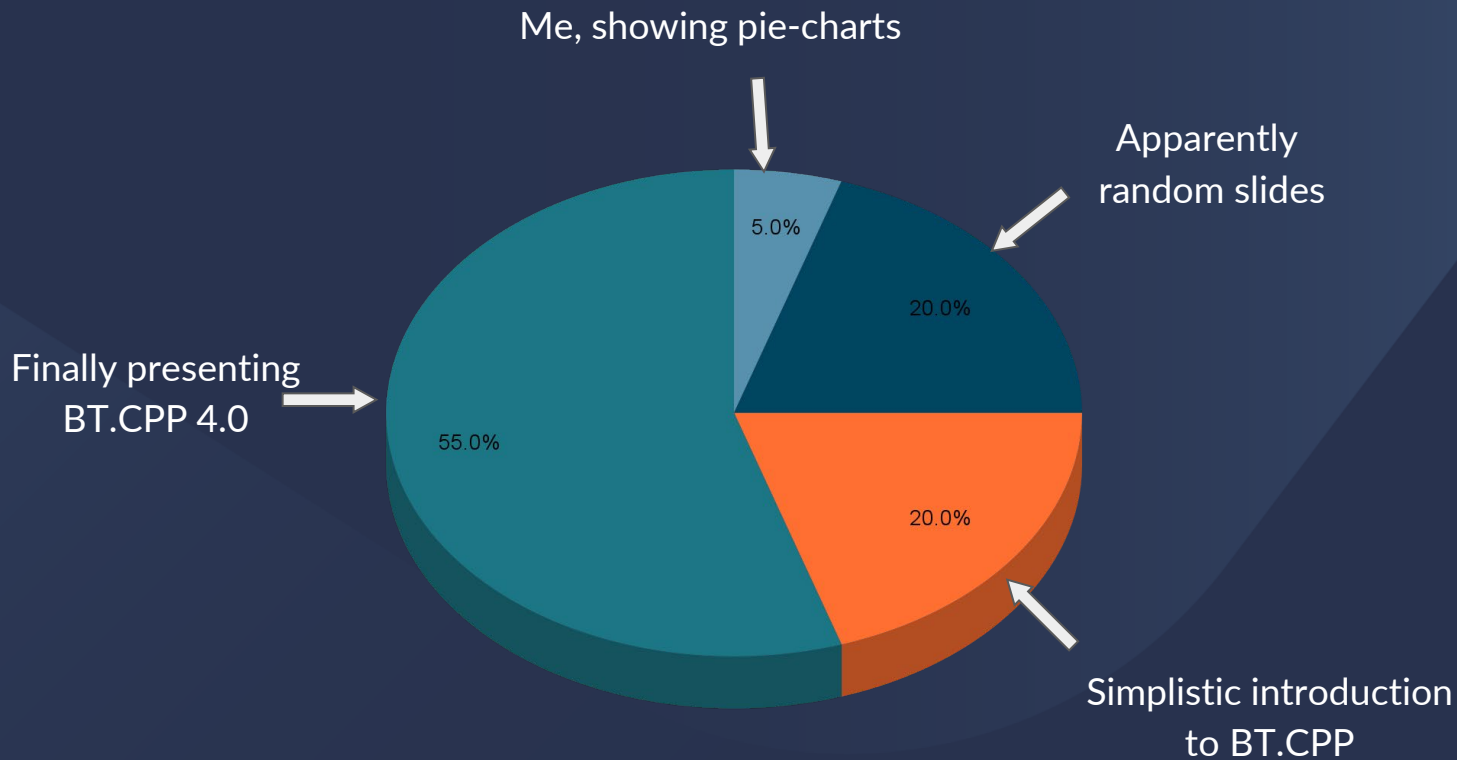
ictures

ement

- Product development

- But....

What to expect in the next 20 minutes



I am not going to explain what Behavior Trees are



What makes software “good” and why do we want that?

Good software is always about “getting things done”, but considering the human first

- 01 Writing software that a computer can interpret and run makes the software “correct”, but not “good”.
- 02 The bottleneck to scale up a software system is usually the ability of the **human mind** to manage complexity.
- 03 This is the reason why a good software is one that **decreases the perceived complexity**.



How do you react to the concerns of your user?

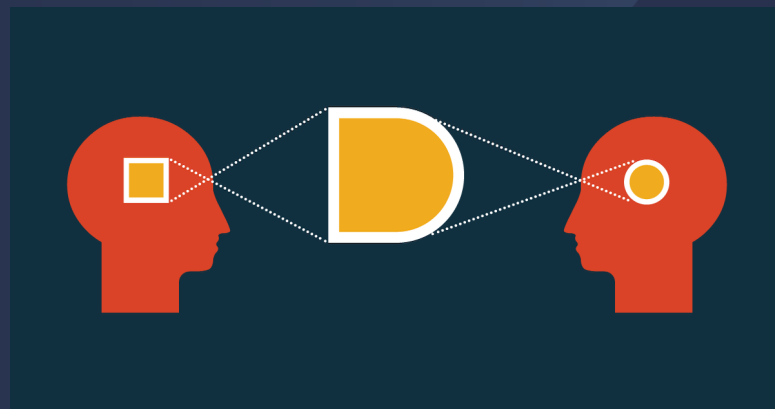
Option 1:

“You are...”



Option 2:

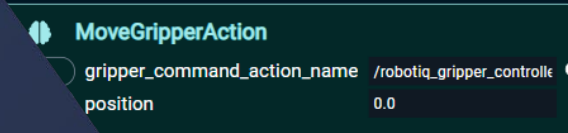
Or we can use **empathy** to understand your user's point of view



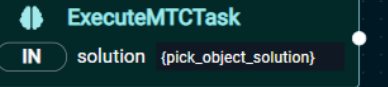
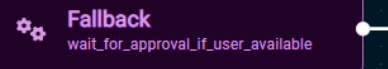
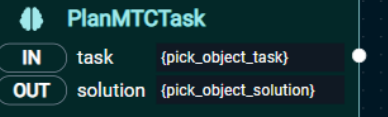
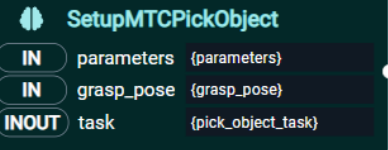
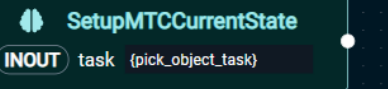
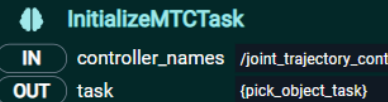
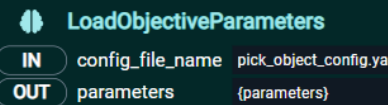
Advantages of Behavior Trees

Advantages of Behavior Trees:

- **Easier to “read”**
Graphical representation of FSMs rapidly become “spaghetti”.
- **Intrinsically hierarchical**
Similar to Hierarchical State Machines.
- **Focus on actions, not states**
We usually model our problem in terms of actions, not states.
- **Extensibility of the “language”**
Decorators and Control logic provide powerful abstractions
- **But... they are not as intuitive for the user :(**
People are very familiar with state machines, and mapping their ideas into behavior tree could be challenging



Sequence
pick_object_main





BehaviorTree.CPP

- **Nodes are defined in C++, Trees in XML**

The best of compiled and interpreted languages together

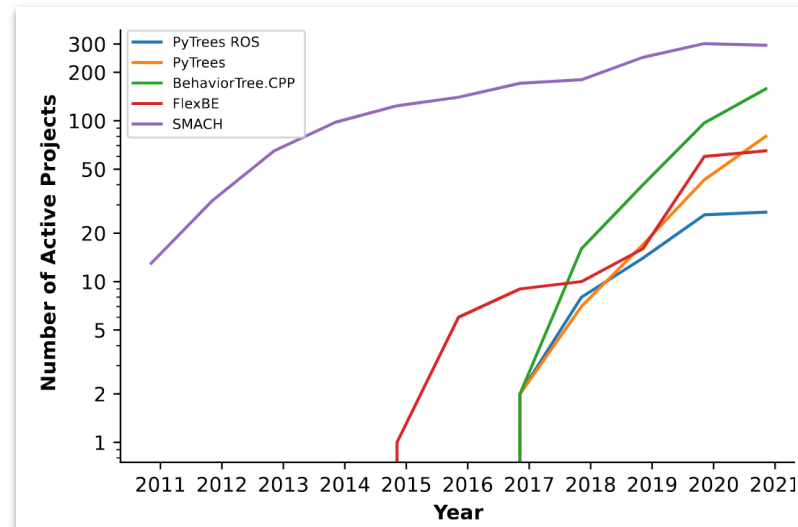
```
<root main_tree_to_execute = "MainTree">

  <BehaviorTree ID="DoorClosed">
    <Sequence name="door_closed_sequence">
      <Inverter>
        <IsDoorOpen/>
      </Inverter>
      <RetryUntilSuccessful num_attempts="4">
        <OpenDoor/>
      </RetryUntilSuccessful>
      <PassThroughDoor/>
    </Sequence>
  </BehaviorTree>

  <BehaviorTree ID="MainTree">
    <Fallback name="root_Fallback">
      <Sequence name="door_open_sequence">
        <IsDoorOpen/>
        <PassThroughDoor/>
      </Sequence>
      <SubTree ID="DoorClosed" />
      <PassThroughWindow/>
    </Fallback>
  </BehaviorTree>

</root>
```

- **Nodes are defined in C++, Trees in XML**
The best of compiled and interpreted languages together
- **Increasing popularity in the ROS community**
Driven probably by the early adoption of Nav2



“Behavior Trees and State Machines in Robotics Applications”

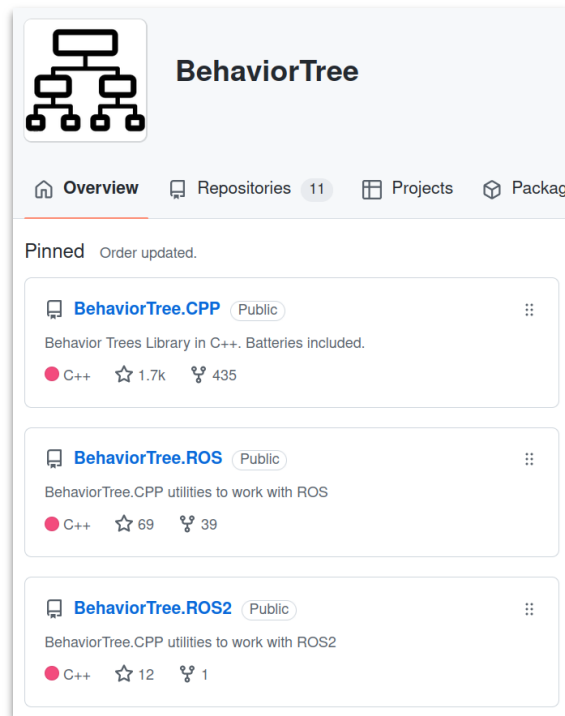
Razan Ghzouli, Swaib Dragule, Thorsten Berger,
Einar Broch Johnsen, Andrzej Wasowski

arXiv:2208.04211



BehaviorTree.CPP

- **Nodes are defined in C++, Trees in XML**
The best of compiled and interpreted languages together
- **Increasing popularity in the ROS community**
Driven probably by the early adoption of Nav2
- **Technically decoupled from ROS**
A double-edged sword: easy to include in any project, but not idiomatic in the context of ROS.



- **Nodes are defined in C++, Trees in XML**

The best of compiled and interpreted languages together

- **Increasing popularity in the ROS community**

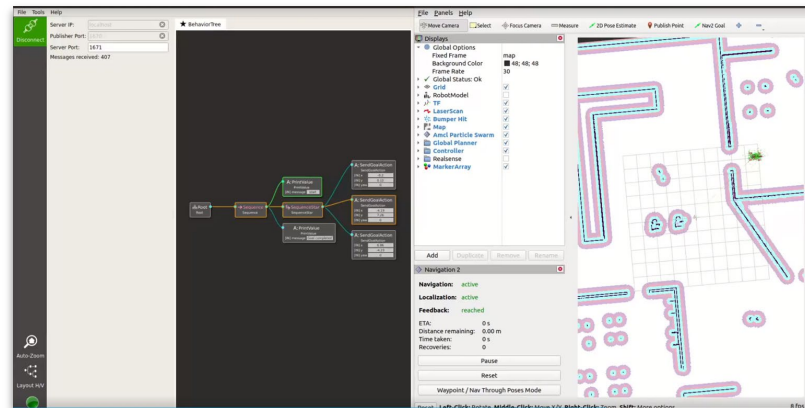
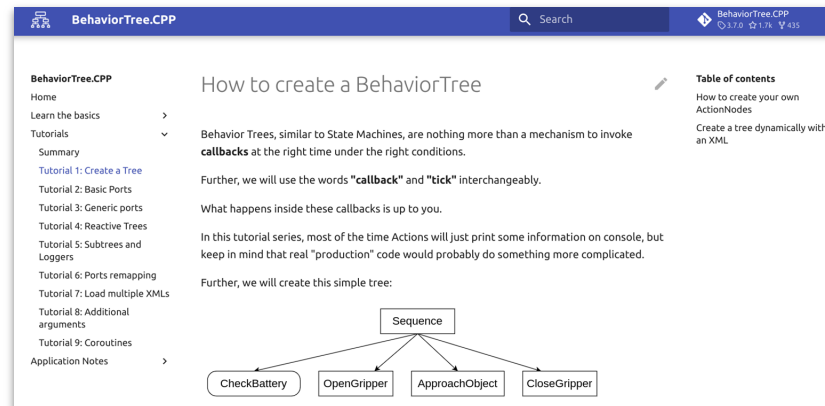
Driven probably by the early adoption of Nav2

- **Technically decoupled from ROS**

A double-edged sword: easy to include in any project, but not idiomatic in the context of ROS.

- **Tooling and Graphic interfaces**

Groot is a useful tool that provides editing and real-time monitoring. **Movelt Studio** integrates a BT editor, too.

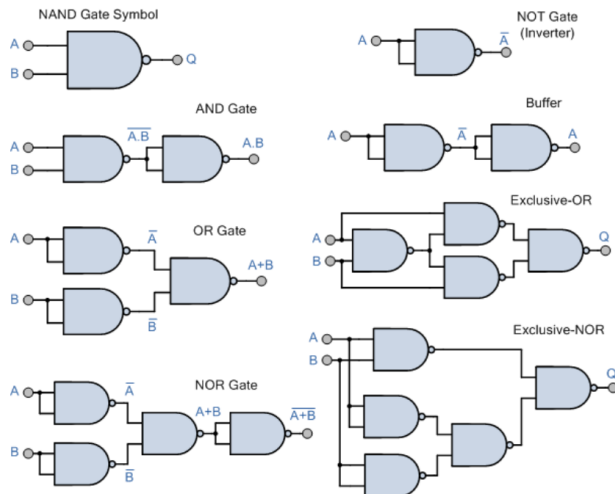


Behavior Trees as extensible Domain Specific Language

In electronics, the NAND gate is the **building block** that you can use to build the others.

Similarly, the most atomic concept of FSM is the **state transition**.

Universal Logic Gates using only NAND Gates



The way Behavior Trees provide **high level of abstraction** goes **beyond hierarchical compositions**:

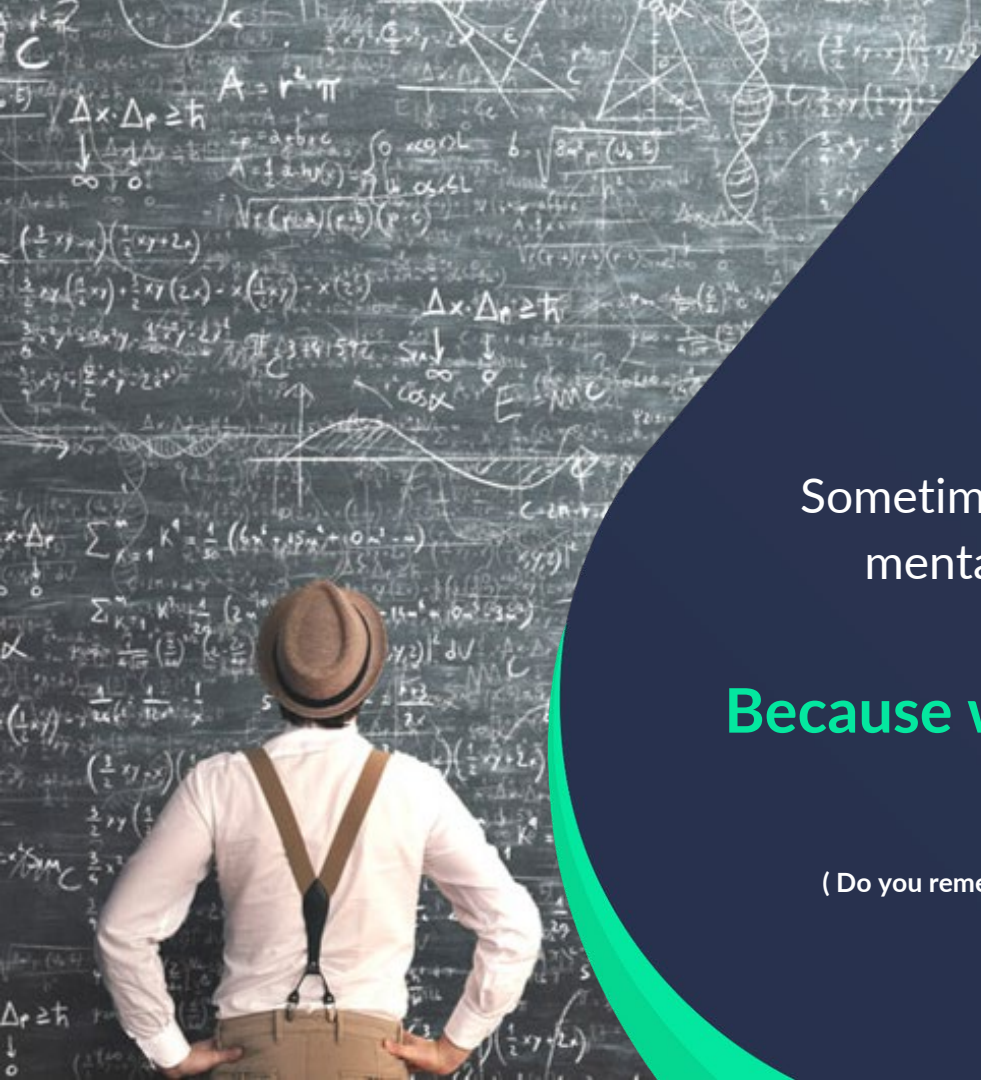
- You can create complex behaviors by **composition of Nodes and SubTrees**
- But you can also “extend the language”, creating your own **custom** Decorators and Controllers.

Current problems with BT.CPP

Sometimes, it takes too much time to map your mental model of the problem into a tree!!

Because we still think in terms of “states”

(Do you remember what we said about “you are doing it wrong” and empathy?)



Let me rephrase...



Behavior Trees don't want you to think in terms of states.

What if **that is our weakness?**

Introducing...



BehaviorTree.CPP 4.0



⋮ Goals of version 4.0

01

Reduce the **cognitive effort** of both the person designing the tree or reviewing it.

02

Translate more effectively the “**mental model**” of the designer into a tree.

03

Add more **expressivity** to the XML code and the GUI representation.

04

In short: **enhanced productivity**



• A scripting language inside BTs

We can now add simple piece of code (“one liners”) to express **equality, assignment, comparison, arithmetic operations** and **if-then-else**. Our variables are the **elements of the blackboard**.

Examples:

- `param_A = 5.0 ; param_B = 'hello'; error_code = 42`
- `(param_A != param_B) && (error_code == 0)`
- `speed = (max_speed / 2) + 4`
- `target = (voltage < 10) ? 'recharge_pose' : 'load_pose'`

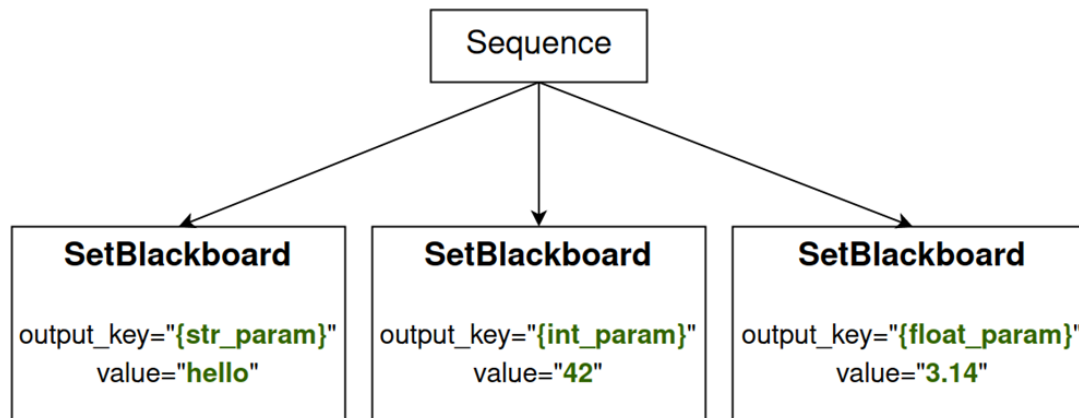
Assignment

Logic operators

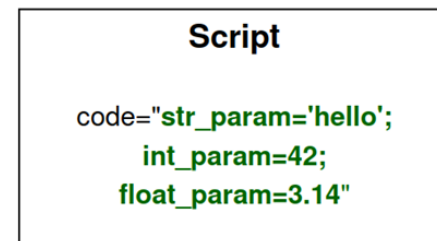
Arithmetic

If-then-else clauses

Example: Initializing blackboard variables



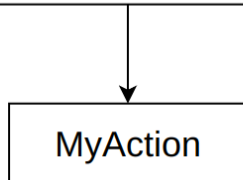
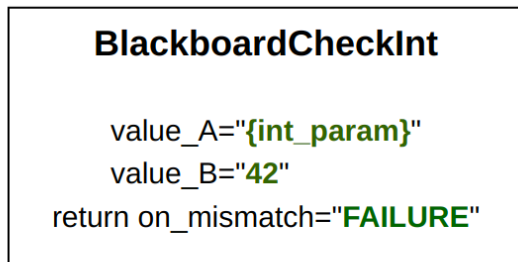
Before



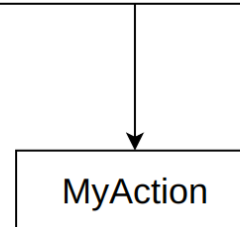
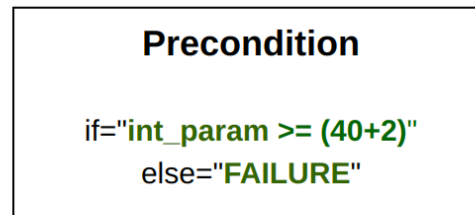
After

(multiple commands using semicolons)

Example: the Precondition Decorator



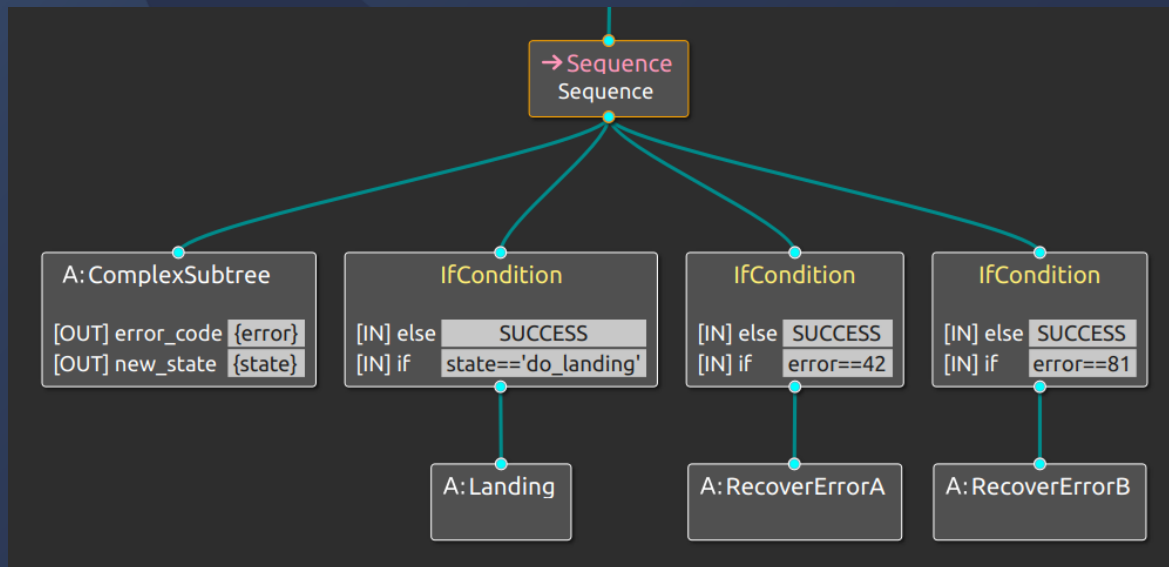
Before



After

(not just equality)

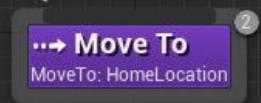
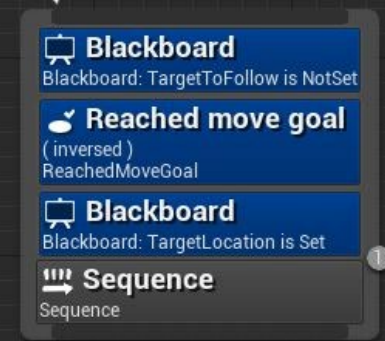
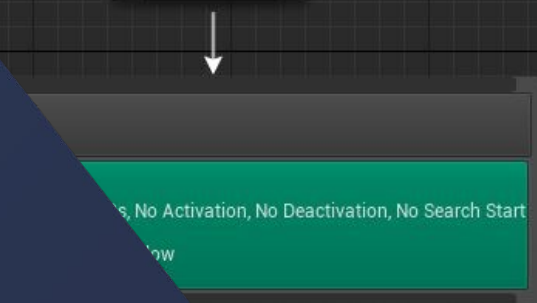
States can make your BT layout simpler



This solves the most common limitation of BTs: sometimes you DO want to think in terms of **states** and you need to return **multiple results**

• Pre and Post conditions

- Now that we have unleashed the power of scripts, we can go a step further, adding **Pre and Post conditions** to **every Node**
- Shamelessly inspired by **Unreal Engine BT**
- It needs **GUI support to become a game changer** (but you will have to wait).
- Implemented as **optional XML attributes**





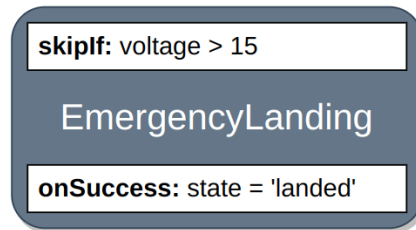
Example

This is a Node (or an entire, complex SubTree) executed only if (**voltage** ≤ 15).

Blackboard variable **state** changed to string “**landed**” if the Node return SUCCESS.

```
<EmergencyLanding  
  _skipIf = "voltage>15"  
  _onSuccess = "state='landed' "/>
```

XML

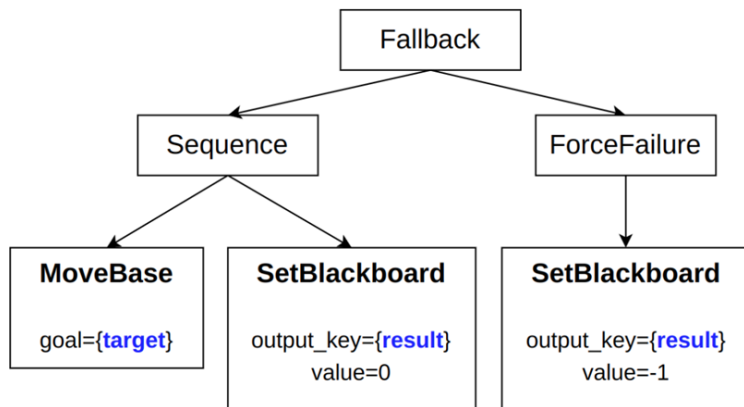


GUI

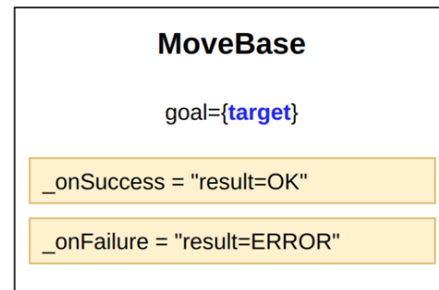


One more examples

Here, instead of recovering locally to the FAILURE of **MoveBase**, I want to do the recovery routine in another part of the tree. I use the port “results” to remember that.



Before



After
(with enums!)

Pre-conditions

- Optional Scripts executed before the actual **tick()**
- Can be used to “skip” the execution of a Node and its children.
- Being able to halt a **RUNNING** Node, it is technically equivalent to **ReactiveSequence** (but better?)

skipIf: voltage > 15

EmergencyLanding

onSuccess: state = 'landed'

<code>_skipIf</code>	Skip (don't execute this node) if condition is true
<code>_failureIf</code>	Skip and return FAILURE
<code>_successIf</code>	Skip and return SUCCESS
<code>_while</code>	Don't start, or halt a RUNNING Node, if the condition becomes false



Post-conditions

- Optional Scripts executed after the actual **tick()**
- Can be used to set variables (states, error codes, etc.)
- In 3.X it was impossible to detect if an action was halted. Now we can use **onHalted()**

skipIf: voltage > 15

EmergencyLanding

onSuccess: state = 'landed'

<code>_onSuccess</code>	Script executed if Node returns SUCCESS
<code>_onFailure</code>	Script executed if Node returns FAILURE
<code>_onHalted</code>	Script executed if a RUNNING Node was halted
<code>_post</code>	Script executed if Node returns either SUCCESS or FAILURE

Documentation, tutorials and Migration guide



BehaviorTree

Tutorial

Editors

Migration from 3.X

4.0

Blog

GitHub

Search

BehaviorTree.CPP 4.0

The C++ library to build Behavior Trees.
Batteries included.

Tutorials

```
graph TD; Root(( )) --> Seq[Sequence]; Seq --> Fallback[Fallback]; Seq --> EnterRoom[EnterRoom]; Seq --> CloseDoor[CloseDoor]; Fallback --> IsDoorOpen[IsDoorOpen]; Fallback --> Retry[Retry]; Retry --> OpenDoor[OpenDoor];
```

Think in terms of Actions, not states

Unlike state machines, behavior trees empathize executing actions, not transitioning between states.

Build extensible and hierarchical behaviors

Behavior Trees are **composable**. You can build complex behaviors reusing simpler ones.

The power of C++, the flexibility of scripting

Implement your Actions in C++ and assemble them into trees using a scripting language based on XML.

BehaviorTree

Tutorial

Editors

Migration from 3.X

4.0

Blog

GitHub

Search

About

Learn the Basic Concepts

Introduction to BTs

Main Concepts

The XML schema

Tutorial - Basics

01. Your first Behavior Tree

02. Blackboard and ports

03. Ports with generic types

04. Reactive behaviors

05. Using SubTrees

06. Port Remapping

07. Use multiple XML files

08. Pass additional arguments

Tutorial - Advanced

Nodes Library

About

Version: 4.0

About

About this library

This C++ library provides a framework to create **BehaviorTrees**. It is designed to be flexible, easy to use and fast.

Even if our main use-case is **robotics**, you can use this library to build **AI for games**, or to replace Finite State Machines in you application.

BehaviorTree.CPP has many interesting features, when compared to other implementations:

- It makes **asynchronous Actions**, i.e. non-blocking routines, a first-class citizen.
- Trees are created at run-time, using an **interpreted language** (based on XML).
- It includes a **logging/profiling** infrastructure that allows the user to visualize, record, replay and analyze state transitions.
- You can link statically your custom TreeNodes or convert them into plugins which are loaded at run-time.

What is a Behavior Tree?

A Behavior Tree (**BT**) is a way to structure the switching between different tasks in an autonomous agent, such as a robot or a virtual entity in a computer game.

About this library

What is a Behavior Tree?

Main Advantages of Behavior Trees

"Ok, but WHY do we need BehaviorTrees (or FSM)?"

I hope you enjoy reading this as much as I hated writing it ❤️

What to expect next

- **Currently in Alpha.**
More stable releases by the end of the year.
- **Feedback from the community**
AKA, “you”. Try it, ask questions, give ideas.
- **Editors supporting the new features**
MovIt Studio and Groot 2.0
- **Documentation and design patterns**
We need to create idiomatic use of BTs



MovIt Studio Developer Platform combines BehaviorTree.cpp with MovIt