# MOOD2Be:
# Models and Tools to design Robotic Behaviors

Davide Faconti

Autonomous System Group
Eurecat Centre Tecnològic
Barcelona, Spain

Final Report: April 2019

**Acknowledgements**

MOOD2Be received funding from the European Union's Horizon 2020 Research and Innovation Programme under the RobMoSys project, under grant agreement No 732410

# 1 Introduction

The goal of the project MOOD2Be is to create a set of open source libraries, software tools and methodologies, which simplify the development of Robotics Behaviors using "Behavior Trees" (BT).

Behavior Trees are a better alternative to Finite State Machine; using a Domain Specific Language, it is possible to easily create behaviors that are **composable**, **configurable** and **reusable**.

Our C++ implementation offers both the type safety and efficiency of this programming language and the flexibility of a scripting one; C++ is also the most used programming language in robotics and it is therefore the ideal candidate to achieve a seamless integration with existing software.

In this document, we will present our software framework and discuss how its meta-model and implementation were both designed from the ground up to achieve four main goals: modularity, composability, reusability and the ability to create reactive behaviors.

To achieve these goals, we leverage multiple concepts and best practices from the domains of Model Driven Software Engineering (MDSE) and Component Based Software Engineering (CBSE); the "RobMoSys approach", in particular, played a key role in understanding how the solution proposed in this document fits into the design of composable software systems.

This document will not describe in details what Behavior Trees are, since a broad literature can be found about this subject; it will focus instead on what makes the project MOOD2Be unique in the context of BT.

# 2 MOOD2Be and the RobMoSys approach

> RobMoSys envisions an integrated approach built on top of the current code-centric robotic platforms, by applying model-driven methods and tools

In many fields of software engineering, it is desirable to design systems from reusable and maintainable modules. This is particularly true in robotics, where a complete software system is usually composed by many different "nodes" or "components".

Model Driven Development offers multiple advantages to the robotic community, where Component Based Software Engineering is already the de-facto standard adopted by most of the robotic middlewares and frameworks. The RobMoSys approach uses MDSE to achieve the following goals, among others:

1. Increased **Composability**: components can be rearranged in different configurations.

2. **Replaceable Components**: it is possible to substituted a component with another implementations that shares the same contracts and interfaces.

3. More **Reusability**: the same component can be reused in different projects or context with little effort.

4. **Reduction in Development Time**: this is particularly true when we look at the cost of the entire project, including deployment and maintenance.

5. **Predictable Properties**: predictability is a fundamental requirement for real-world applications.

6. **Standardization of models and interfaces**: strongly related to reusability and composability.

7. **Commodization of base components**.

Furthermore, RobMosSys focuses on many additional goals such as: Quality of Service, Certifiable Systems, Benchmarking, etc.

The seven benefits listed earlier are those that we consider particularly related to MOOD2Be. Our project has a symbiotic relationship with RobMoSys: it learns from the latter to achieve these goals and, at the same time, it provides a ready to use toolset that achieves these benefits in the context of Behavior Design.

## 2.1 Separation of Roles and Concerns

Two key principles described in RobMoSys guided the development of our software are:

- Good software design should achieve **Separation of Concerns**.
- Good architectures should achieve **Separation of Roles**.

The "Separation of Concerns" is a well known principle in software. In the context of CBSE, it means that we should keep decoupled the "Five Cs": Computation, Configuration, Communication, Coordination and Composition.

The term "Separation of Roles", instead, refers to the fact that it should be possible to clearly identify and decouple the activity of multiple "roles" such as: Domain Experts, System Integrators, Component Developers, etc.

Our project provides methodologies and tools to keep **Coordination** decoupled from other concerns: a "robot behavior" is, in fact, nothing more than a mechanism that coordinates and orchestrates the state of each software component.

Similarly, our approach allows a good Separation of Roles, because it decouples the work of Component Developer and System Integrator from the one of Behavior Designer. MOOD2Be aims to help Behavior Designers, enhancing their ability to work independently and efficiently: implementing a robot behavior should not need domain specific knowledge nor advanced programming skills.

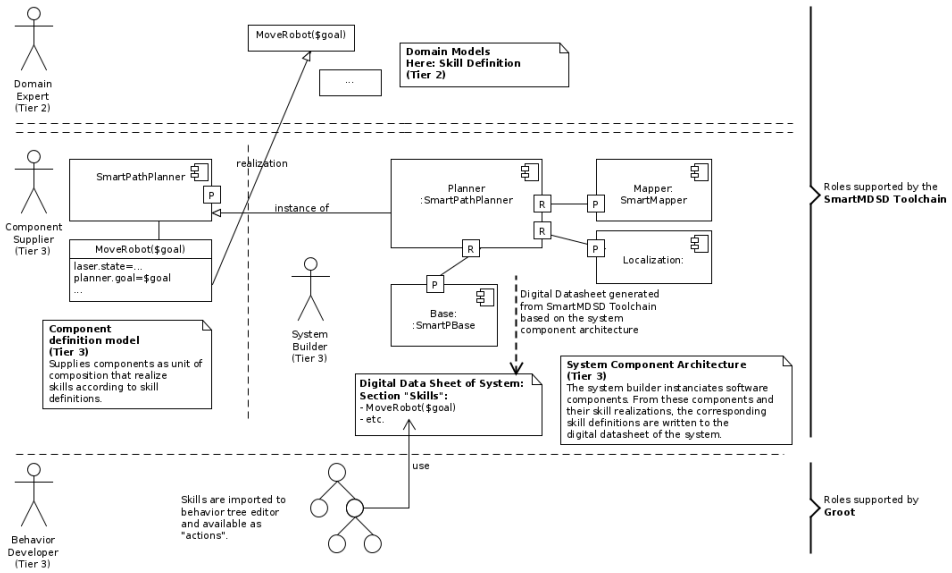


Figure 1: Separation of roles in the RobMoSys context

In figure 1 we may notice as the concept of **Skills** is used to describe capabilities of the system, from the point of view of the Behavior Designer.

Any particular system may provide a specific set of Skills, that are summarized inside the **Digital Datasheet**.

## 2.2 Orchestration of Service-Oriented Components

How these recommendations are converted into practice in a Component Bases software architecture?

A common anti-pattern that we observed very often in robotics, consists in "spreading" a little bit of the business logic inside multiple components. This makes the global behavior of the robot:

- Hard to introspect.
- Hard to debug.
- Difficult to modify.

Furthermore, this means that some components might be not reusable, because they contains a piece of business logic that is application-specific.

A better design pattern should use a single Coordinating Component that orchestrate the others. this means that component should be more stateless and be modelled as *services*.

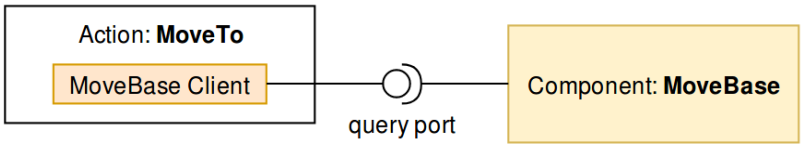


Figure 2: Relation between an Action of the BT and a service-oriented component.

This latter recommendation is the key to understand the relationship between a Coordinator based on Behavior Trees: if the software system is mostly constituted for service-oriented components, the *client code* that invokes a certain service or remote routine should be contained inside the **Actions** of the BT.

A simple example can be seen in figure 2: the orange boxes represent those piece of software that is platform and framework dependent.

The Action **MoveTo**, in this case, is nothing more than a way to invoke the execution of a callback; inside this callback we can find the specific implementation of the client that sends a query/request to a service, in this case to the component that controls the position of the robot (MoveBase).

In terms of Communication Patterns, the default recommendation is to use **asynchronous queries** to send requests to the server and receive feedback.

In behavior trees, this feedback must be interpreted as one of these three states: SUCCESS, FAILURE or RUNNING.

# 3 Differences between Behavior Trees and Finite State Machines

Finite State Machines (FSM) are one of the most known paradigm to define the behavior of either robots or software agents, such as Non-Player Characters (NPC) in games; nevertheless, Behavior Trees grew in popularity in the last decade.

Finite State Machines are affected by multiple problems which become apparent only in real-world scenarios, where the number of transitions and conditions is sufficiently large.

- A FSM having N states has potentially NxN state transitions.
- The number of state is related to the context of execution; therefore N may grow very quickly.
- When adding or removing a state, it is necessary to change the conditions of all other states that have transition to the new or old one.
- States are strongly coupled; this affects the reusability of previously defined conditions and states.

- When the number of states is reasonably large, both the graphical and textual representation of the entire behavior become too complex to be understood by the designer.

These issues affect mostly humans, not computers. A software interpreter/executor can easily handle a large state machine, but a human will surely struggle to understand and predict the overall behavior of the system.

In other words, the main problem with FSM is that the **cognitive overhead** experienced by developers, more specifically Behavior Designers, quickly grow with the number of states, becoming unmanageable.

Hierarchical Finite State Machines (HFSM) partly mitigate the problem of reusability of subroutines and the combinatorial explosion of state transitions that characterizes FSM, but they are still affected by strong coupling and a limited grammar.

When we refer to "limited grammar" we mean the expressiveness of its metamodel and its ability to implement common design patterns.

In figure 3 a very simple FSM is represented: a robot will naively avoid obstacles which can be detected either on its right or left side.

Behavior Trees address most of these problems, improving modularity and reusability.

- They are intrinsically hierarchical. From a semantic point of view, any Subtree represents a potentially reusable behavior.
- BT provides a much richer grammar that the developer can use and extend to implement common design patterns.
- Both the textual and graphical representations are easier to "read" for a human, because the hierarchy is conveniently arranged from top to bottom and the priority of nodes from left to right.
- BT uses primarily Actions instead of States; this approach matches more closely the "mental model" used to describe a behavior and it is more consistent with the software interfaces offered by service-oriented architectures.

Using Behavior Tree, the previous example can be represented in a more intuitive and less verbose way, as represented in figure 4 and 5.

The nodes where Obstacle Detection is implemented can be either a ConditionNode, i.e. a leaf of the tree that can return SUCCESS or FAILURE, or a DecoratorNode which will execute its child node only if the internal condition is met.

It is worth repeating that the visual representation of the FSM is completely arbitrary, whilst the respective position of the nodes in the behavior tree represents the order of execution/evaluation of children nodes.

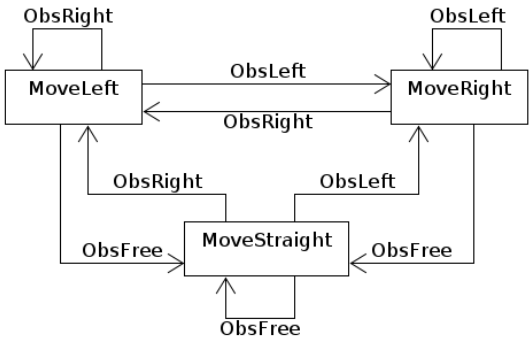


Figure 3: A FSM with 3 states and 9 transitions

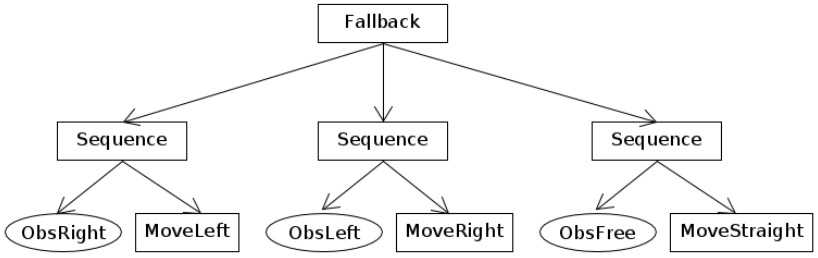


Figure 4: A BT where DetectObstacles is implemented as a ConditionNode

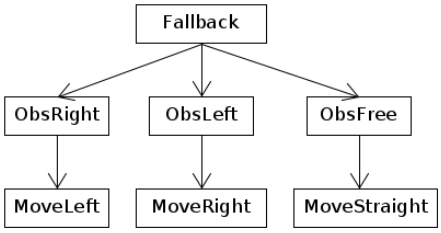


Figure 5: A BT where DetectObstacles is implemented as a DecoratorNode

# 4 Data Flow and Data Ports in Behavior Trees

According to the principle of Separation of Concerns, it is recommendable to keep the Coordination of components decoupled from the Communication of data between them. Unfortunately, this recommendation doesn't take into account some important requirements of real world systems:

- Frequently, state transitions are possible only when a specific data is available; in other words, a specific input is a **pre-condition** to the execution of an Action.
- Similarly, the **post-condition** of a state transition or Action could consist in making a certain data available.

As a consequence, we can hardly imagine a useful Coordination framework that doesn't handle explicitly this contract between Actions/States. During the development of **BehaviorTree.CPP**, it became apparent that a data port meta-model was needed to explicitly address these requirements.

## 4.1 Input Ports as function arguments

One of the most basic Actions that we may ask a mobile robot to perform is "GoTo" or "MoveTo".

Arguably, we may agree that it is unreasonable to create a specialized Action for each location, such as "GoToKitchen" or "GoToLivingRoom"; this approach would be hard to maintain and it would compormise reusability.

Many implementations of Behavior Trees use **Blackboards** to pass data and states between nodes; a Blackboard is nothing more that a globally accessible Key/Value storage, where the Key is usually a string.

An action such as "GoTo" would read implicitly, i.e. in the source code of its implementation, the location of the target from the Blackboard.

| | **Syntax** | **Description** |
|---|---|---|
| 1 | GoToKitchen | Destination hard-coded in the implementation. |
| 2 | GoTo( "kitchen" ) | A label that identifies a particular destination. |
| 3 | GoTo( 0.3, 4.1, 0.7 ) | Location represented as Pose2D (X, Y, Theta). |
| 4 | GoTo( {destination} ) | The string "destination" is the key to be used to find a Pose2D in the Blackboard. |

Table 1: Different ways to parametrize the GoTo action.

Initially we addressed this problem using **NodeParameters** to configure a generic Node. In a general purpose programming language, NodeParameters would be the equivalent of arguments passed to a function, either by value or by reference.

From a semantic point of view, NodeParameters share the same meta-model of **Input Ports**; we will use further the latter name, because it is much more consistent with the port model described in the next session.

## 4.2 The Data Ports Meta-Model

Blackboards are conceptually simple to understand and to implement, but they hides some potential issues.

For instance, the entries of a Blackboard can be accessed directly (and implicitly) in the source code; in other words, data flow is not modelled. As a consequence, static analysis tools can't be used to verify the correctness of a tree.

For the same reason, whenever it is necessary to read/write a different entry of the Blackboard, the source code of the Node must be modified.

In fact, a Blackboard is just a "glorified" set of global variables and it is well known that global variables may (and eventually "will") break encapsulation of software modules.

Analogously, in the context of Behavior Trees and Blackboards, we need to address the problem of **name clashing** in the Key/Value table

We want to be able to reuse Subtrees in many different scenarios and to implement complex behaviors through hierarchical composition; unfortunately multiple trees may use the same generic key name, such as "target", "goal", "result", "pose", etc., for different purposes.

To solve this issue, we propose a meta-model that formally defines Data Ports and Data Flow as elements of the Node model:

1. The Blackboard is replaced by Input and Output Ports; these ports must be defined explicitly in the model of the Node.
2. Any Port has a string identifier, called "Key".
3. Ports within the same Subtree are automatically connected when they share the same Key.
4. Ports are strongly typed. Connected ports must have the same type and type consistency can be checked both statically or at deployment time.
5. Ports sharing the same Key, but located in different Subtrees, will not be connected automatically.
6. A parent tree can access the ports of a child Subtree only when explicit connections are provided.

## 4.3 Example

In the example depicted in figure 6, we may notice that Output Ports pointing to a certain Key are connect automatically to Input Ports using the same identifier.

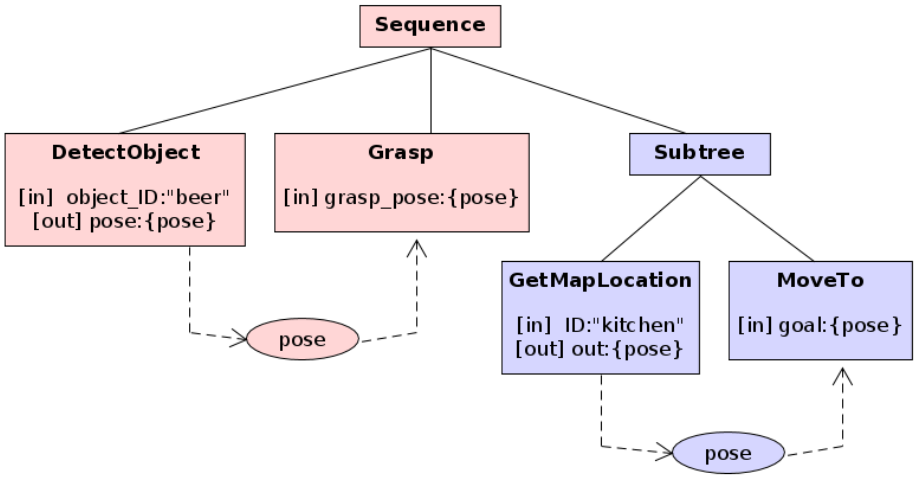


Figure 6: Ports are connected by name, but only inside the same Subtree.

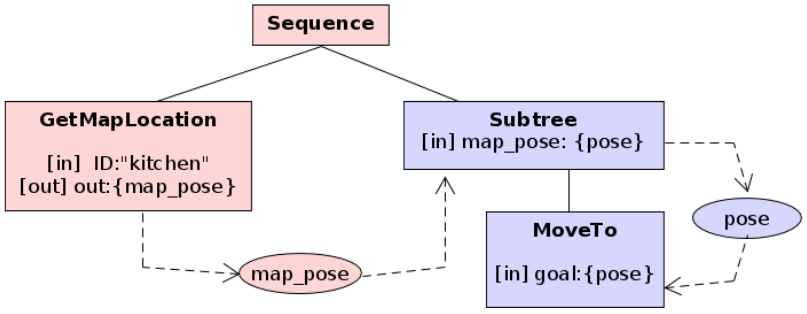


Figure 7: Ports must be remapped to connect different namespaces.

For instance the 3D pose computed by **DetectObject** is used as input of the action **Grasp**.

The pair of actions **GetMapLocation** and **MoveTo** are connected in a similar way, but the key "pose" has in this case a different type (it is a 2D coordinate) and must not be confused with the pose of the object.

Nevertheless, since the latter pair of actions are located inside a different Subtree, no connections is established with the former pair, since each Tree/-Subtree has its own *namespace*.

Two different colors were used to better visualize the namespaces.

If we need to connect a port in the parent Tree with another in the child Subtree, we need to explicitly remap the ports, as shown in figure 7.

In this particular example the only way to connect **GetMapLocation** to **MoveTo** is remapping explicitly the key "map_pose", that is visible only in the

parent namespace, to "pose", that is located inside the child namespace.

# 5 Reactive Behavior Trees

Action could be **synchronous**, i.e blocking from the point of view of the caller, or **asynchronous**. An asynchronous action returns the state RUNNING and allow the tree to execute other Nodes or entire branches in parallel.

A particular set of ControlNodes should be used to create "reactive behaviors", i.e behaviors were asynchronous Action are aborted if the right conditions are not met.

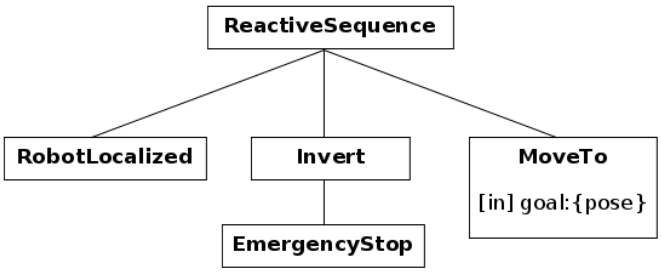


Figure 8: Check if robot is localized and emergency stop is pressed.

Our meta-model and the corresponding software framework discussed in section 6 address explicitly the problem of concurrency.

Figure 8 contains an example of **ReactiveSequence**:

- "MoveTo" is started only if "RobotLocalized" return SUCCESS and "EmergencyStop" returns FAILURE.
- These two Conditions are executed repeatedly, as long as "MoveTo" returns RUNNING.
- If any of these conditions changes, "MoveTo" is aborted.

# 6 Software library: BehaviorTree.CPP

**BehaviorTree.CPP** is a C++ implementation of the meta-models we described.

The purpose of this software library is to wrap existing code into custom Nodes of the tree and to simplify the creation of Behavior Tree Executors, i.e Software Components that load a BT at run-time and execute it.

Custom Nodes are usually **Actions**, that the robot can execute through Skills, or **Conditions**.

The framework provides also some extendable mechanisms to monitor, log and debug the execution of a tree. Using C++ was a strategic decision; this

computer language is the most popular in the robotic community and this allows the users to easily integrate the library with their legacy code.

Very often, FSMs and BTs are implemented using scripting languages such as **Python or Lua**. This provides some important advantages:

- New scripts/behaviors can be modified and loaded at run-time, without stopping, recompiling and restarting the main application.
- Using a dynamic language is more comfortable for Behavior Designers.

Unfortunately, this "mixed language" approach often requires the creation of "binding code"; this might be a considerable entry barrier and/or overhead.

*BehaviorTree.CPP* solves this problem providing a scripting language, currently based on XML, to define both particular trees and models of custom Nodes. The proposed work flow involves the following steps:

1. Either the Component Developer or the System Integrator creates *once* the custom Nodes (Actions and/or Condition) in C++. These are highly reusable and application-independent building blocks.
2. Behavior Designer can use the domain specific scripting language to compose built-in and custom Nodes into hierarchically trees.

Since ports are connected at deployment time, i.e when the tree is instantiated by the Executor, it is impossible to check these types at compilation-time.

On the other hand, the type-check can be performed at run-time, before executing the Behavior Tree for the first time; in the future, it should be relatively easy to create a static analyzed that performs this operation off-line.

From the point of view of Model Driven Development, it is worth noting that we took an unusual approach: instead of using the popular **model-to-code** work flow or, in other words, implement code generation, we used template meta-programming techniques in C++ to automatically generate the model from the source code.

## 6.1 Implementation details

### 6.1.1 Factory and plugins

The design pattern known as "Factory" is used to decouple the C++ implementation from the scripting language.

The Executor parses the textual representation of the Tree, allocates instances of either custom or built-in Nodes and compose them into a tree.

Each Node type must be registered once into the Factory; this can be done directly in the C++ code, using static linking, or at run-time loading new Nodes from a set of plugins. We provide a simple and well documented way to create create self-registering dynamic libraries which encapsulate the custom Nodes.

The Factory generates a model of the custom Nodes during the registration phase.

### 6.1.2 Safe type erasure

In our meta model, Input and Output Ports are type-safe. This, in terms of C++ code, means that we need to create an object contained that is both type-erased and type-safe. We decided to use a modified version of **std::any** that was tailored to provide few advantages, when compared with other implementations of "any":

- **Small Object Optimization (SOO)**: object smaller that 16 bytes can be stored in-place without any memory allocation.
- **String Optimization**: storing a string requires only a single memory allocation instead of two.
- **Safe Numerical Conversion**: the standard version of "any" require users to cast the value to the original type. Our implementation is less strict and allows conversion between numerical numbers, as long as there is no underflow, overflow or loss of numerical precision.

An important implication of this approach is that, since input ports can read the actual value of a port from the XML file, the developer must provide a simple parsing mechanism to convert a string into a custom type. This can be done providing the template specialization of the function **convertFromString** as shown in the following code sample.

```
// We want to be able to use this custom type
struct Pos2D
{
  double x;
  double y;
};

// Template specialization to parse Pos2D from string
namespace BT {
template < >  Pos2D convertFromString(StringView str)
{
    // Real numbers separated by semicolons
    auto parts = splitString(str, ';');
    Pos2D output;
    output.x = convertFromString<double>(parts[0]);
    output.y = convertFromString<double>(parts[1]);
    return output;
}
} // end namespace BT
```

### 6.1.3 Logging and monitoring with Observers

The library implements the Observer Pattern to monitor the state transitions of each Node of the tree.

This mechanism can be used to build two families of tools, which collect information from the Executor in a non intrusive way:

- **Loggers**, which once "attached" to a tree can record any single state.
- **Monitors**, that are used to publish in real-time state transitions to external applications.

The library provides, out of the box, few Loggers and Monitors that the developers can use to record, analyze or replay the execution of a tree.

These tools are particularly useful when used together with Groot, that will be described in the next session.

# 7 Groot: Integrated Development Environment for BTs

One of the main goals of MOOD2Be project was to improve the productivity of the Behavior Designer. For this reason, we developed **Groot**, a Integrated Development Environment (IDE) that can be used to either create, edit, monitor or debug Behavior Trees.

The designer composes the tree using a palette of built-in and/or custom Nodes, using an intuitive and modern drag-and-drop graphical interface.

It is worth mentioning that Groot is *not* a BT Executor itself. The connection point between the graphic application and the software component containing the executor is suaully the XML file containing the representation of the tree.

The XML representation was designed to be "human readable", i.e easy to edit manually a textual editor. Therefore, Groot is completely optional in terms of work flow.

Ironically, we may even say that Groot is a "fancy XML editor" rather than an IDE.

But, when we think about other formats such as HTML, it becomes apparent that even when textual editors are sufficient or even faster to use to quickly edit a portion of the file, domain-specific graphical tools can improve development speed and lower the entry barrier perceived by early adopters.

To better understand the relationship between Groot and the Executor, we should refer to the diagram 10:

- Custom Nodes are implemented in C++ and compiled as dynamic libraries (plugins).
- These plugins can be loaded either by the Executor or by Groot; the latter will extract the Node model from the plugins.
- Once Groot has loaded the models of the Nodes, the latter can be composed into trees and subtrees, which are saved using the XML format.

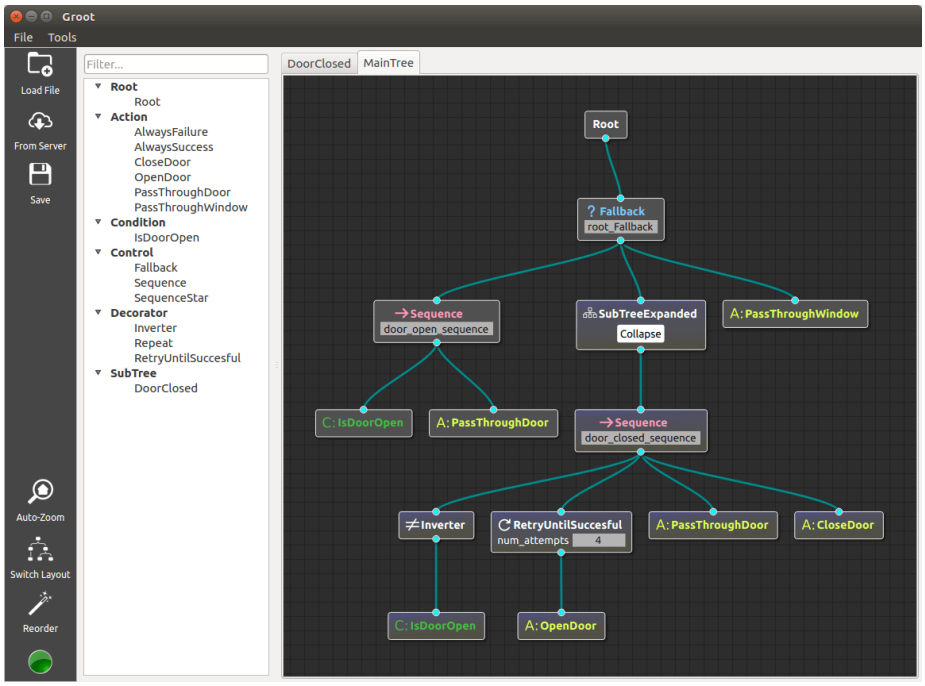


Figure 9: Groot in Editor mode

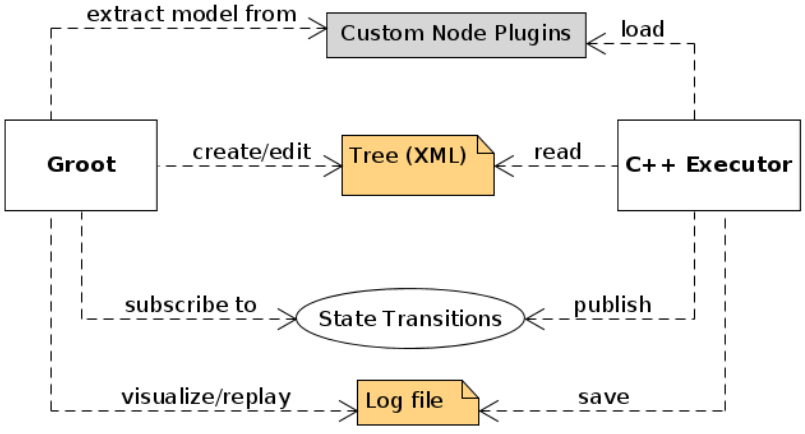


Figure 10: Relation between Groot and the C++ Executor

- The Executor can load the XML file, instantiate the tree and start executing it.
- Furthermore, the Executor can publish state transitions in real-time using a TCP socket; Groot can subscribe to this information remotely and monitor the state of the tree.
- All the state transitions can be recorded and visualized step by step offline, using a binary log file generated by the Executor.

### 7.1 The "Chicken or Egg" dilemma

What comes first, the model or the implementation?

One of the main challenges (and theoretical dilemma) that was made apparent by Groot, was that the model representation of a custom Node, used by the graphic interface, and the actual binary implementation, built using *BehaviorTree.CPP*, must be kept synchronized.

At the beginning, during the conceptual phase, it is convenient to create from scratch the model of the custom Nodes, even if there isn't any corresponding C++ implementation and/or associated plugin.

This allow the user to quickly prototype a tree, reason about the logic and share ideas with other people in the development team.

But, as a consequence, it becomes up to the developer to ensure that this manually created models remain in-synch with the actual implementations.

This intrinsic problem can be mitigated in the future by better tools, being a "model repository" built on top of a distributed database, the most promising approach.

## 8 Results and KPI

The project MOOD2Be proposes a new kind of BT design that allows users to create reusable, reactive and hierarchical robot behaviors.

Even if we present in this document an innovative meta-model, the main focus of this R&D project has always been "development" rather than "research". In fact, we believe that well documented, mature and industrial-grade tools can have a more significant and positive impact in the robotic community than pure theoretical results.

In other words, the main Key Performance Indicator (KPI) of our project is the impact in the community and the rate of adoption.

During the execution of the project, our software framework and methodology has been used in multiple projects, both internally by our team or externally.

We released our open source software relatively early on *Github* and this allowed us to receive a very valuable feedback from other experts in the robotic community.

We are happy to admit that the current Input/output Port model, that is probably the most note-worthy innovation in our meta-model, is the result of an open minded discussion with the community.

Both *Behaviortree.CPP* and *Groot* are experiencing, at the time of writing, a good "traction" in terms of number of downloads, active contributors and number of users.

One of the KPI initially proposed was a measurable reduction in the development time; after one year of work, we don't have any strong argument or data that can support the fact that this specific goal has been achieved.

On the other hand, when the role of Coordination and Orchestration in a Component Based software system is understood by the developer and promoted by a specific software framework, we observed that each of the components in the system becomes more reusable and well encapsulated than it was initially.

In other words, we may say that a nice side-effect of our approach is a system wide improvement in terms of modularity.

# 9 Final remarks

This project taught us an important lesson: to impact a community, we should be open to new ideas, listen to opinions different than ours and understand what user really need.

We, developer and researchers, focus too often on teaching what we consider the "right" solution to others, rather than learning from them.

Non-functional requirements such as documentation and support can make a huge difference and are too often neglected in many research projects.

Our constructive feedback to the RobMoSys project is that their proposed methodologies provide a useful "mental framework" that help us reasoning about reusability and composability, but there are also recommendation and abstractions that are difficult to use in practice.

For instance, our Behavior Tree meta-model used to have poor composability when we tried to keep the concepts of Data Flow completely decoupled from Coordination; these two software Concerns, in our opinion, need to be modelled together, to be useful in practice.

About the Separation of Roles, our experience revealed that we can hardly implement an idealized work flow where Skills are defined by Domain Experts, implemented by Component Developers and composed by Behavior Designers.

In practice, behaviors are the result of a two-ways and iterative dialog between these experts, rather than one-way pipeline where models and artifacts are made available using a top-down approach.

The Skills themselves are, in our opinion, an abstraction that might be too "coarse grained", because being as general as possible, they may fail to take into account specific corner cases that only the Behavior Developer or the System Integrator may be able to understand and/or predict.