Behavior Trees: Research and Application

T. Tannous

Abstract—Behavior Trees (BTs) were developed as a modular AI framework for computer games, overcoming the scalability and adaptability limitations of Finite State Machines (FSMs). Unlike FSMs, BTs use a hierarchical decision-making structure, improving modularity, reusability, and analysis. These benefits have made BTs popular in robotics, where dynamic AI is essential for autonomous systems. This paper examines the structure of BTs, compares them with traditional models, and explores their applications in robotics. It reviews specific use cases where Behavior Trees have been effectively implemented, with a focus on robotics, and highlights research challenges for future AI advancements.

Index Terms—Behavior Trees (BTs), Decision-making models, Robotics

CONTENTS

1 Introduction

Behavior Trees (BTs) offer a modular, scalable approach to decision-making, addressing the limitations of traditional models like Finite State Machines (FSMs). Their hierarchical structure enhances flexibility, making them ideal for dynamic applications such as robotics.
This paper explores the application of BTs in robotics, identifying key use cases where they have been effectively implemented to improve autonomous systems. By examining these implementations, we aim to highlight the benefits and potential of BTs in advancing robotic AI.

2 Definition and Structure

At its core, a Behavior Tree (BT) is composed of simple components, which combine to form complex and powerful structures. A BT is represented as a directed rooted tree, where the internal nodes, known as **control flow nodes**, manage the flow of execution, and the leaf nodes, known as **execution nodes**, perform tasks. Each node is connected in a parent-child relationship, with the root node being responsible for initiating the execution process.

A BT starts its execution from the root node that generates signals that allow the execution of a node called ticks with a given frequency, which are sent to its children. A node is executed if and only if it receives ticks. The child immediately returns Running to the parent, if its execution is under way, Success if it has achieved its goal, or Failure otherwise.

In the classical formulation of BTs, there are four categories of control nodes (Sequence, Fallback, Parallel, and Decorator) and two categories of execution nodes (Action and Condition). They are all explained below and summarized in Table 1.

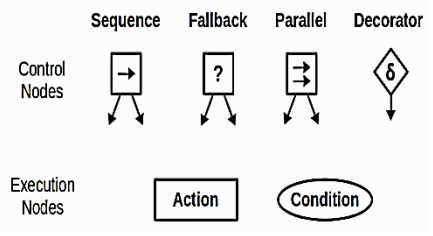
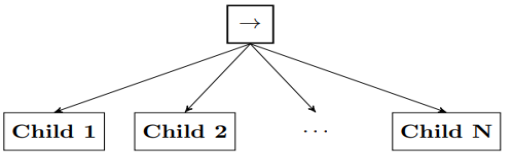


Figure 1: Overview of behavior tree nodes.

2.1 Foundational elements

2.1.1 Control Nodes

The Sequence node executes Algorithm 1, which corresponds to routing the ticks to its children from the left until it finds a child that returns either Failure or Running, then it returns Failure or Running accordingly to its own parent. It returns Success if and only if all its children return Success. Note that when a child returns Running or Failure, the Sequence node does not route the ticks to the next child (if any). The symbol of the Sequence node is a box containing the label "→", shown in Figure 2.

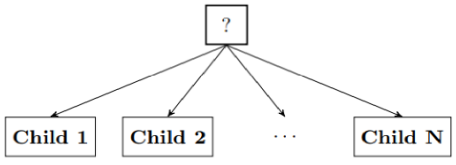


Algorithm 1: Pseudocode of a Sequence node with N children

```
for i ← 1 to N do
    childStatus ← Tick (child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Failure then
        return Failure
return Success
```

Figure 2: Graphical representation + Algorithm of a Sequence node

The Fallback node executes Algorithm 2, which corresponds to routing the ticks to its children from the left until it finds a child that returns either Success or Running, then it returns Success or Running accordingly to its own parent. It returns Failure if and only if all its children return Failure. Note that when a child returns Running or Success, the Fallback node does not route the ticks to the next child (if any). The symbol of the the Fallback node is a box containing the label "?", shown in Figure 3.

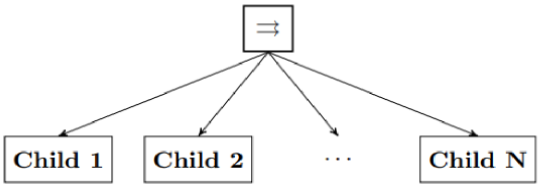


Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

```
for i ← 1 to N do
    childStatus(i) ← Tick (child(i))
if Σ_{i:childStatus(i)=Success} 1 ≥ M then
    return Success
else if Σ_{i:childStatus(i)=Failure} 1 > N − M then
    return Failure
return Running
```

Figure 3: Graphical representation + Algorithm of a FallBack node

The Parallel node executes Algorithm 3, which corresponds to routing the ticks to all its children and it returns Success if M children return Success, it returns Failure if N − M +1 children return Failure, and it returns Running otherwise, where N is the number of children and M ≤ N is a user defined threshold. The symbol of the the Parallel node is a box containing the label " ⇉", shown in Figure 4.



Algorithm 2: Pseudocode of a Fallback node with N children

```
for i ← 1 to N do
    childStatus ← Tick (child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Success then
        return Success
return Failure
```

Figure 4: Graphical representation + Algorithm of a Parallel node

The Decorator node modifies a single child node with a custom policy. A decorator has its own set of rules for changing the status of the "decorated node". For example, an "Invert" decorator will change Success to Failure, and vice-versa. While decorators can add flexibility to your behavior tree arsenal, you should stick to standard control nodes and common decorators as much as possible so others can easily understand your design. The symbol of the Decorator is a rhombus, as in Figure 5.

Figure 5: Graphical representation of a Parallel node

2.1.2 Execution Nodes

The Action node executes a command when it receives ticks. It returns Success if the action is correctly completed or Failure if the action has failed. While the action is ongoing it returns Running. An Action node is shown in Figure 6.

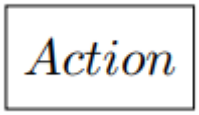


Figure 6: Graphical representation of an Action Node.

The Condition node checks a proposition when it receives ticks. It returns Success or Failure depending on if the proposition holds or not. Note that a Condition node never returns a status of Running. A Condition node is shown in Figure 7.

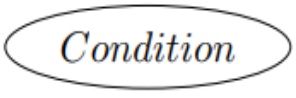


Figure 7: Graphical representation of a Condition Node.

To provide a clearer overview of the different types of nodes discussed above, a summary table is provided below in Table 1. This table consolidates key characteristics of control and execution nodes.

Node type	Symbol	Succeeds	Fails	Running
Sequence	→	If all children succeed	If one child fails	If one child returns running
Fallback	?	If one child succeeds	If all children fail	If one child returns running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	shaded box	Upon completion	When impossible to complete	During completion
Condition	white oval	If true	If false	Never

Table 1: Summary of Behavior Tree Nodes and Ther Characteristics

2.2 Behavior trees vs. finite-state machines

In theory, it is possible to express any process as a Behavior Tree (BT), Finite State Machine (FSM), or one of the other abstractions, or even as plain code. However, each model comes with its own set of advantages and disadvantages in aiding design at larger scales. Specifically, when comparing BTs and FSMs, there is a tradeoff between modularity and reactivity.

Generally, BTs offer greater ease in composition and modification, making them ideal for scalable and flexible designs. In contrast, FSMs excel at designing reactive behaviors, making them suitable for situations that demand immediate responses to changes in the environment.

To explore this comparison in more detail, let's consider a robotics example. Imagine a robot tasked with picking up an object: the robot must move to the object, close its gripper to grab it, and then return to its home position. A side-by-side comparison of a BT and FSM implementation for this simple task will show that both approaches are relatively clean and easy to follow. However, the differences in how these models handle complexity and scalability will become more apparent in larger and more dynamic systems.

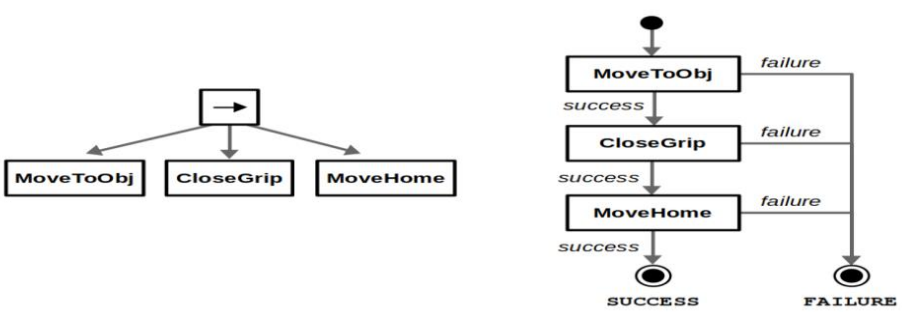


Figure 8: Behavior Tree(Left) and Finite-State Machine(right) for a robot picking example

A key advantage of **BTs** over **FSMs** is their ability to accommodate modifications with minimal structural changes. For instance, if we need to verify the **pre-grasp position** and correct it before closing the gripper, a **BT** allows for the direct insertion of a **subtree** within the existing sequence of actions. In contrast, an **FSM** would require the **reconfiguration of multiple transitions**, increasing complexity. This demonstrates the superior **modularity and scalability** of BTs, making them well-suited for dynamic robotic applications.

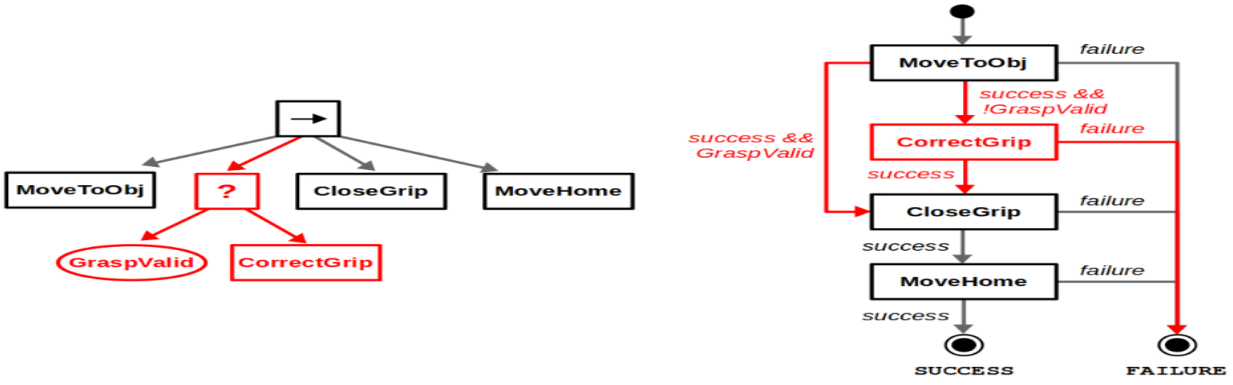


Figure 9: Modifications to the BT (left) and FSM (right) to incorporate the added pre-grasp correction behavior.

On the other hand, designing reactive behaviors presents certain challenges. Consider a scenario where a robot operates on a finite power source and must return to a charging station whenever the battery level is low before resuming its task. While this behavior can be implemented using BTs, finite state machines (FSMs) provide a more straightforward approach for achieving full reactivity—where the robot immediately transitions to charging regardless of its current state. FSMs enable this reactivity by defining transitions between any two states, ensuring an immediate response to critical conditions.

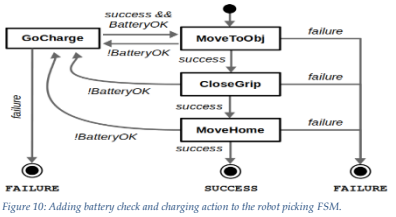


Figure 10: Adding battery check and charging action to the robot picking FSM.

However, this often results in a more complex FSM structure. A common criticism of FSMs is the risk of creating "spaghetti state machines," where excessive transitions make the design difficult to manage. While hierarchical finite-state machines (HFSMs) mitigate this issue through structured state management, they still require careful design to maintain clarity. Hierarchical Finite-State Machines (HFSMs) help streamline the structure by organizing states into higher-level groupings. In this case, a superstate named 'Nominal' is introduced, clearly distinguishing between two operating modes: normal operation and charging.

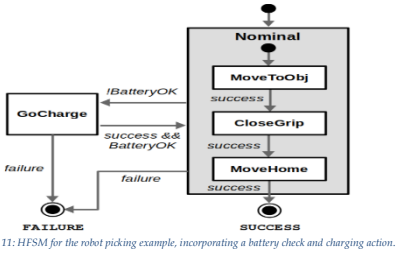


Figure 11: HFSM for the robot picking example, incorporating a battery check and charging action.

Incorporating a battery check and charging action into a BT is straightforward (depicted in Figure 12); however, this check is not inherently reactive, it is only evaluated at the start of the sequence. Enhancing reactivity within BTs requires additional constructs, such as Reactive Sequences, which allow for continuous monitoring of battery levels throughout execution. This construct ensures that previous children in a sequence remain active even after returning Success, enabling a BT to dynamically terminate an ongoing task if the battery level drops below a critical threshold. This adaptation allows for more flexible and responsive behavior, addressing the challenges of reactivity within BT-based designs.

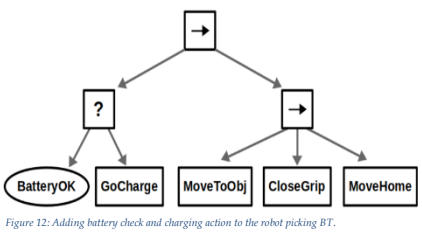


Figure 12: Adding battery check and charging action to the robot picking BT.

Due to the tradeoff between modularity and reactivity, FSMs are well-suited for managing higher-level operating modes (e.g., normal operation vs. charging), while BTs are particularly effective at constructing complex behavior sequences, especially in handling failure recoveries. This approach offers the best of both worlds: high-level mode switches are handled by an FSM, and mode-specific behaviors are managed with BTs. Therefore, a hybrid approach combining both FSM and BT elements would be an ideal solution, as demonstrated below:

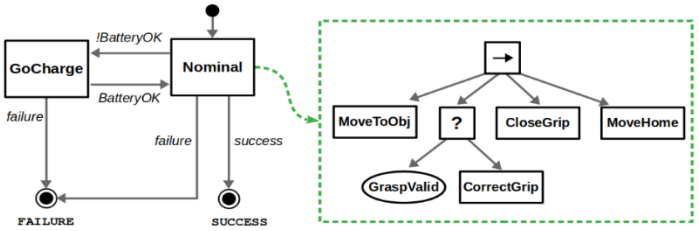


Figure 12: Hybrid FSM and BT design: High-level mode switching managed by the FSM, while mode-specific behaviors are handled by the BT.

To further understand the key differences between Behavior Trees and Finite State Machines, the following table highlights the features, advantages, and challenges of each model in various aspects of design and implementation:

Feature	Behavior Trees (BTs)	Finite State Machines (FSMs)
Scalability	**Highly scalable**: BTs can handle complex and dynamic behaviors by simply adding new nodes or subtrees without affecting the existing structure. The hierarchical nature of BTs enables easy extension and modification.	**Becomes complex with more states**: FSMs become increasingly difficult to manage as the number of states and transitions grows. Complex state diagrams can lead to a "spaghetti" effect, where states and transitions become tangled, making the system harder to maintain.
Reusability	**Nodes can be reused**: BTs allow for high reusability, where nodes (such as action or condition nodes) can be used across different contexts and behaviors without modification. This modularity allows for easy composition and efficient updates.	**Less reusable components**: FSMs generally define distinct states, and transitions between these states are often tightly coupled to the specific context of the system. This makes FSM components harder to reuse in different scenarios, as the transitions and actions are state-dependent.
Flexibility	**Dynamic execution flow**: BTs allow for more flexible behavior since the execution flow can adapt based on the success or failure of different nodes. Behavior can be dynamically altered by adding or removing nodes, making it easier to implement complex recovery strategies, condition-based branching, and adaptive responses.	**Fixed state transitions**: In FSMs, the execution flow is fixed, based on predefined transitions between states. Once a state is entered, it follows a rigid, linear progression unless explicitly modified. This can limit flexibility, especially when dynamic behavior is needed.
Debugging	**Easier to debug**: Due to the hierarchical structure of BTs, debugging is often simpler. Each node is discrete, making it easier to isolate issues, track status, and modify behavior. The visual representation of BTs also helps identify problems quickly.	**Can be challenging with many states**: As FSMs grow in complexity, debugging becomes more difficult. With a large number of states and transitions, it's harder to trace issues back to specific transitions or state changes, particularly when behaviors are deeply nested or depend on multiple states.

Table 2: Comparison of Behavior Trees (BTs) and Finite State Machines (FSMs) in Terms of Scalability, Reusability, Flexibility, and Debugging.

2.3 Behavior tree software libraries

Behavior Trees (BTs) can be implemented using various libraries. Two of the most widely used ones in robotics are **py_trees** (Python) and **BehaviorTree.CPP** (C++). These libraries provide the fundamental tools for designing and executing BTs, making them ideal for robotic decision-making systems.

py_trees (Python)

Developed by Daniel Stonier, **py_trees** is a flexible Python-based library that enables the creation, execution, and visualization of behavior trees.

Advantages of py_trees:

- Written in Python, making it highly flexible for runtime modifications.
- Well-integrated with ROS 2, making it a strong choice for modern robotics applications.
- Actively maintained with new features, including enhanced visualization and logging tools.
- Ideal for research projects and real-time adaptive BTs.

Considerations:

- Some terminology differs from classic BTs. For instance, **Fallback nodes** are referred to as **Selector nodes**, and their behavior slightly varies.
- While it supports ROS 1, many of its latest features are optimized for ROS 2.
- The flexibility of Python can be both an advantage and a drawback, depending on project requirements.

Resources for py_trees:

- py_trees documentation
- PyTrees-ROS Ecosystem

BehaviorTree.CPP (C++)

Developed by **Davide Faconti and Michele Colledanchise**, **BehaviorTree.CPP** closely follows the **Behavior Trees in Robotics** book notation and is a **C++ library** widely adopted in the **ROS ecosystem**.

Advantages of BehaviorTree.CPP:

- Follows the **standard BT notation** from the **Behavior Trees in Robotics** book.
- **Highly optimized for ROS 2**, particularly in the **ROS 2 Navigation Stack (BT Navigator)**.
- Uses an **XML-based workflow**, making it ideal for **predefined BT structures**.
- Integrated with **Groot**, a GUI tool that allows for **visualizing and editing BTs**.
- Encourages modularity, where **nodes are registered as C++ classes** and combined in XML.

Considerations:

The **XML-first approach** works well when the BT structure is predefined, but makes **runtime modifications** mor- difficult.

While **programmatic modifications** are possible, they are **not well-documented or recommended**.

The **visualization tools (Groot) do not support runtime modifications** effectively.

Resources for BehaviorTree.CPP:

BehaviorTree.CPP GitHub

Groot: BT Editor and Visualizer

3. APPLICATIONS:

3.1 Use Cases of Behavior Trees in Robotics

Behavior trees (BTs) have become a pivotal tool in robotics, particularly in the context of ROS (Robot Operating System) applications. Their modular and hierarchical structure allows for the development of complex behaviors in a manageable way. Below are some key use cases that illustrate the versatility and effectiveness of behavior trees in robotic systems.

Multi-Agent Coordination

Behavior trees are particularly effective in multi-agent scenarios where coordination among agents is crucial. By structuring behaviors hierarchically, BTs enable agents to respond dynamically to the actions of others. For instance, in a search and rescue operation, multiple drones can utilize behavior trees to coordinate their movements, ensuring coverage of the area while avoiding collisions.

Autonomous Navigation

In autonomous navigation tasks, behavior trees can manage various navigation strategies, such as obstacle avoidance, path planning, and goal reaching. A BT can be designed to prioritize tasks based on environmental conditions, allowing a robot to adapt its behavior in real-time. For example, if an obstacle is detected, the BT can switch from a direct path to a more complex navigation strategy that includes rerouting.

Human-Robot Interaction

Behavior trees facilitate natural interactions between robots and humans. By defining behaviors that respond to human actions or commands, robots can engage in more intuitive and effective communication. For instance, a service robot in a restaurant can use a BT to manage tasks like greeting customers, taking orders, and serving food, adjusting its behavior based on customer responses.

Simulation and Training

In simulation environments, behavior trees are used to create realistic agent behaviors for training purposes. For example, in military simulations, BTs can model the decision-making processes of soldiers or vehicles, allowing for the testing of strategies and tactics in a controlled setting. This application is particularly valuable for developing and refining operational plans before real-world implementation.

Integration with Machine Learning

The integration of behavior trees with machine learning techniques enhances their capabilities. For instance, a BT can be used to structure the decision-making process while machine learning algorithms optimize the parameters of the behaviors based on performance data. This hybrid approach allows for continuous improvement of robotic behaviors in dynamic environments.

3.2 Use Cases of Behavior Trees in Video Games

Behavior Trees (BTs) have proven to be highly effective in video game AI design due to their modularity, flexibility, and ability to create dynamic behaviors. Below are three key use cases where BTs are widely implemented in video games:

AI NPCs (Non-Player Characters)

BTs are commonly used to control NPC behavior, enabling them to perform tasks such as patrolling, engaging in combat, and reacting to player actions. For example, in stealth games, enemy NPCs can transition from patrolling to searching for the player when alerted and switch to combat mode upon detection.

Enemy AI and Combat Systems

BTs are well-suited for controlling enemy AI, allowing them to adapt to the player's actions by adjusting combat strategies, evading attacks, or calling for reinforcements. In action games, this dynamic response enhances the challenge and depth of combat encounters, keeping players engaged.

Companion NPCs and Team Dynamics

In team-based or RPG games, BTs help manage the interactions between the player and AI-controlled companions. These NPCs can use a BT to decide when to heal, assist in combat, or follow the player, ensuring their behavior adapts to the player's actions and the evolving game environment.

3.3 Use Cases of Behavior Trees in Software Automation

Behavior Trees (BTs) offer an effective way to model complex workflows and automation processes. Their modular structure and dynamic behavior make them particularly useful in various automation scenarios. Below are three key use cases where BTs are applied in software automation:

Automated Testing

BTs are frequently used in automated testing frameworks to simulate user interactions and ensure software behaves as expected under various conditions. For example, BTs can model sequences of actions like filling out forms, clicking buttons, and validating output, allowing for easy modification and extension of test cases as new features are added.

CI/CD Pipeline Automation

Behavior Trees can be used to manage the sequence of tasks in Continuous Integration and Continuous Deployment (CI/CD) pipelines. BTs enable dynamic and conditional execution of tasks, such as building code, running tests, and deploying applications, depending on the state of the system or the environment, improving flexibility and error handling.

4. Conclusion

Behavior Trees (BTs) have proven to be a powerful tool in robotics and software automation due to their modular, hierarchical structure, which enhances scalability, flexibility, and maintainability. In robotics, BTs are effectively used for multi-agent coordination, autonomous navigation, and human-robot interaction, while in software automation, they streamline tasks like testing and robotic process automation. The clarity and modular nature of BTs make them especially attractive for developers working with ROS, enabling the creation of intricate behavior patterns that improve adaptability and responsiveness.While BTs excel in modeling complex behaviors, their reactivity can be a challenge, though constructs like Reactive Sequences help address this limitation. Compared to Finite State Machines (FSMs), BTs offer better modularity and adaptability, while FSMs are more suited for reactive tasks. A hybrid approach combining both can leverage the strengths of each, offering optimal solutions for complex systems. For more detailed insights, refer to the official ROS documentation on behavior trees.

5. REFERENCES

[1] Iovino, M., Scukins, E., Styrud, J., Ögren, P., & Smith, C. (2022). A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems*, *158*, 104096. https://doi.org/10.1016/j.robot.2022.104096

[2] Marzinotto, A., Colledanchise, M., Smith, C., & Ögren, P. (2014). Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 5420–5427). IEEE. https://doi.org/10.1109/ICRA.2014.6907656

[3] Colledanchise, M., & Ögren, P. (2022). *Behavior Trees in Robotics and AI*. arXiv preprint. https://arxiv.org/abs/1709.00084