Comparison between Behavior Trees and Finite State Machines

Matteo Iovino[a], Julian Förster[b], Pietro Falco[c], Jen Jen Chung[d], Roland Siegwart[b], and Christian Smith[e]

Abstract—Behavior Trees (BTs) were first conceived in the computer games industry as a tool to model agent behavior, but they received interest also in the robotics community as an alternative policy design to Finite State Machines (FSMs). The advantages of BTs over FSMs had been highlighted in many works, but there is no thorough practical comparison of the two designs. Such a comparison is particularly relevant in the robotic industry, where FSMs have been the state-of-the-art policy representation for robot control for many years. In this work we shed light on this matter by comparing how BTs and FSMs behave when controlling a robot in a mobile manipulation task. The comparison is made in terms of reactivity, modularity, readability, and design. We propose metrics for each of these properties, being aware that while some are tangible and objective, others are more subjective and implementation dependent. The practical comparison is performed in a simulation environment with validation on a real robot. We find that although the robot's behavior during task solving is independent on the policy representation, maintaining a BT rather than an FSM becomes easier as the task increases in complexity.

Note to Practitioners—Behavior Trees (BTs) have gained significant traction in robotics, with over one hundred research papers published annually in the past five years. BTs have demonstrated their advantages in various applications, such as mobile manipulation, underwater and aerial robotics, and humanoids. Choosing the right policy representation is crucial for successful task completion. While Finite State Machines (FSMs) have been popular for their simplicity, they can become cumbersome as tasks grow more complex. We show this by implementing BTs and FSMs in a mobile manipulation task, tested in simulation and on a real robot. As task difficulty increases, FSMs become harder to maintain due to the rising number of transitions. Our findings provide practical guidance for engineers, helping them choose the best policy representation for their specific robotic applications.

Index Terms—Behavior Trees, Finite State Machines, Robot Control, Mobile Manipulation, Collaborative Robotics

I. INTRODUCTION

In modern industrial applications robots share their environment with humans, so they need to handle the unpredictability that may arise from unexpected outcomes of actions or different types of failures and errors. To this end, robots need to be controlled by reactive policies since fixed sequences of actions may lead to faulty behaviors. These modern manufacturing environments require flexible policies that can adapt rapidly to new tasks, thus featuring constraints in terms of modularity, to allow for reusability. Furthermore, policies need to be human readable, to allow for monitoring and debugging but also to make robot intentions understandable to close-by human operators.

Behavior Trees (BTs) are a task switching policy representation with the properties of reactivity, readability, and modularity and they are becoming state-of-the-art for robot control, especially in the research environment [1]. In industry however, Finite State Machines (FSMs) are still the preferred policy representation, due to their intuitive and simple design and a more mature exploitation in the field.

We believe that industry still needs to be provided with concrete proofs about the differences on the policy behavior in robotic tasks. The goal of this paper is to compare BTs with FSMs in terms of reactivity, modularity, readability, design choices, and more practically in a set of mobile manipulation tasks. By readability of a decision structure we mean how easily a human operator can understand and debug the agent's behavior by examining its layout, which in turn simplifies tracking the robot's execution. For the comparison to be fair, we use the same low level implementation for the robot skills, changing only the encapsulating container to a behavior for a BT or a state for an FSM.

The main contribution of this paper is to provide a set of concrete examples that shows what these properties mean in robotic applications. We only consider a mobile manipulation task as example to facilitate the comprehension of our argumentation. However, the findings of this paper can be extended to other tasks and agents. For example, authors in [2] find BTs modularity better suited for the design of industrial control software. In [3], authors prefer a BT policy to modularize and scale their architecture to control an autonomous agent navigating in a crop field. Interestingly, authors in [4], propose a hybrid design where operators can switch between FSM modes that are implemented as a BT controlling a complex robotic system in an autonomous mining application. Finally, in [5] the inherent task hierarchy of BTs makes them the ideal choice for converting high level instruction into actionable skills in a humanoid robot. The results of this paper are applicable to these cases where autonomous agents are controlled by a reactive policy. If the policy of the agent is only sequential, then BTs do not offer any advantages over FSMs, as we discuss in Section IV-D.

We support the comparison with common metrics for algorithm complexity and graph distances that suggest the advantages of using BTs as an alternative to FSMs. The reader
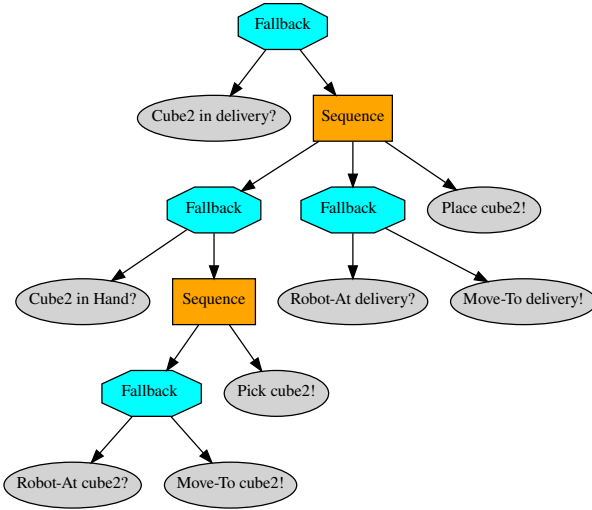
Fig. 1: Behavior Tree automatically generated by a planner using the backchaining method.

Fig. 2: State Machine executing a sequence of actions to solve a mobile pick and place task.

is referred to [6] for a more in-depth analysis on metrics for evaluating BTs. This paper extends our previous work [7] by:

1) comparing BTs and FSMs on the reactivity and readability properties other than the modularity;
2) extending the comparison on the modularity aspect by providing more examples;
3) extending the comparison to Hierarchical FSMs;
4) extending the experimental section by evaluating the metrics on concrete examples and by implementing the two policies on a real robotic system.

The remainder of this paper is organized as follows. Section II provides background on the two policy representations and on related work that compares them. Section III describes how policies can be automatically generated and presents the design choices that are considered for both BTs and FSMs. The metrics as well as the policy properties considered for the comparison are detailed in Section IV, while the practical realization is presented in Section V. Finally, Section VI summarizes the results of the comparisons.

II. BACKGROUND AND RELATED WORK

This section briefly defines BTs and FSMs and presents the theoretical foundations and early results on the comparison between the two representations.

A. Behavior Trees

Behavior Trees are a representation for task switching controllers that originated in the gaming industry but then rapidly gained success in robotics applications [8]–[11].

A BT is a directed tree that is recursively ticked with a depth-first pre-order traversal. Internal nodes are called *control nodes*, (polygons in Fig. 1), where the most common types are *Sequence*: executing children in a sequence, returning once all succeed or one fails, *Fallback* (or *Selector*): executing children in a sequence but returning when one succeeds or all fail, and *Parallel*: executing children in parallel, returning when a pre-determined subset of children is successful. Leaves are called *execution nodes* or *behaviors* (ovals in Fig. 1) and are
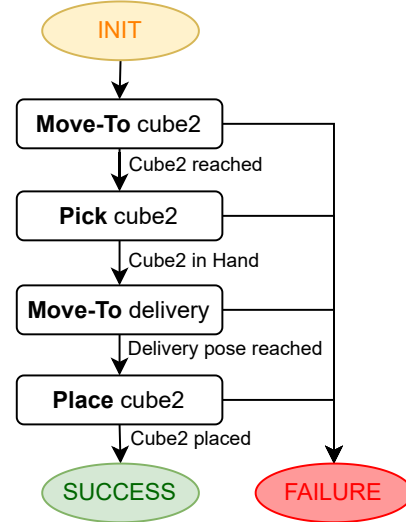
either (i) Action nodes that execute a behavior when ticked and return one of the status signals {*Running*, *Success*, *Failure*}, or (ii) Condition nodes that encode status checks and sensory feedback and immediately return *Success* or *Failure*.

BTs have explicit support for task hierarchy, action sequencing, and reactivity [1]. BTs are inherently modular since every node takes the tick signal as input and returns a status signal. Thus, every subtree can be moved around and reused without compromising the logical correctness of the policy. Moreover, modularity ensures that every building block is independent and thus can be tested separately. Reactivity is ensured by the *Running* return state, which allows the whole tree to be logically evaluated at every tick and running actions to be preempted if some behavior with higher priority needs to be executed. BTs are functionally close to Decision Trees with the main difference in the *Running* state that allows BTs to execute actions for longer than one tick.

B. Finite State Machines

Finite State Machines (FSMs) derive from state automata and feature a set of states and transitions between them (ovals and arrows in Fig. 2, respectively). Every state encodes a controller for robot behavior which produces effects in the environment upon execution. The effects trigger an event that transfers the execution from one state to the next. Since FSMs also include Sequential Function Charts (SFCs), a graphical programming language for Programmable Logic Controllers (PLCs), they are widely used in industry [12] mainly due to their intuitive design and implementation simplicity.

As noted in [8], the execution of an FSM is similar to the GoTo statement in early programming languages, where the flow of execution leaps from one section of the program to another and proceeds from that point. In programming, and by extension also in robotics, GoTo statements are considered to be harmful, as reported by Dijkstra in [13]. On the other end, the execution of a BT can be compared to a function call, where the execution flow returns to where the function was initially called.

The main shortcoming in FSM design is the trade-off between reactivity and modularity. To be reactive, an FSM must feature numerous transitions that need to be handled individually when states are added or removed, reducing its modularity and scalability. This modularity issue is somewhat alleviated by logically grouping states into hierarchies. Even if more reusable, it is still hard to add or remove states in Hierarchical Finite State Machines (HFSMs) and often hierarchies are handcrafted, actually shifting the modularity problem to inner layers. Reactive HFSMs often result in fully connected graphs [8].

Modularity in FSMs can also be achieved by parallel composition [14]. With this operator it is possible to design FSMs for different subtasks independently and then use the parallel composition to automatically generate an FSM for the complete task. With parallel composition, in the generated FSM the number of states grows combinatorially with the states in the sub-FSMs that compose it. Therefore, this operator grants modularity at the cost of higher complexity and loss of readability. Due to this complexity, parallel composed FSMs are often represented symbolically and not graphically.

C. Related Work

BTs have been compared to FSMs in earlier studies, but these comparisons have been either purely theoretical [8], [15], [16], or speculative [17]. In [8], the authors outline the pros and cons of both approaches, and in [15], they theoretically demonstrate how BTs can modularize FSMs. They also propose a design for an HFSM that functions like a BT. In [16], the authors compare BTs with other related architectures (Decision Trees, Teleo-reactive Programs and in particular FSMs) in terms of reactivity, readability and expressiveness. They demonstrate that **FSMs are more expressive than BTs** since they can build behaviors based on actions that have access to internal variables and to past decisions, while **BTs are more reactive** because they restart the execution from the root at each new input **and more readable** as they do not encode past decisions in the representation.

In [18], the same authors formalize modularity for reactive control architectures, highlighting that BTs possess structural interfaces that enable every component to interact with others: a subtree is a BT, and an action behavior is a degenerate form of a BT. In contrast, FSMs lack such structural interfaces and are not considered modular in their analysis unless a structure is imposed (e.g., in HFSMs). Modularity is measured by the Cyclomatic Complexity, defined as

$$CC = a + s - n + 1, \tag{1}$$

where a, s, n represent the number of arcs, sinks (terminal nodes) and nodes in a decision structure, respectively. This measure is applied to BTs that have been transformed to graphs with single entry and exit nodes. This implies that **BTs have optimal modularity** as their Cyclomatic Complexity is 1. In addition to this analysis, we measure modularity in terms of the efforts required to modify a structure by adding or removing elements in it. We propose to quantify using the computational complexity of such operations and the graph

edit distance between a baseline structure and its modified versions.

Klöckner [17] suggests using a BT as a control policy for UAV missions, speculating on the benefits of this design over FSMs. Previous work [19] also compares these two policy representations from a practical standpoint in the context of autonomous driving. The comparison uses as metrics the Cyclomatic Complexity (CC) and the Maintainability Index (MI). The MI of a software piece considers CC, lines of code, comment percentage, and Halstead volume (a measure of the distinct and total numbers of operands and operators). However, the CC was poorly defined compared to [18], and we argue that MI is more influenced by the library used to implement the policies, leading us to disregard it. We propose instead to count the number of elements in the structure. We are aware that this approach is not a metric per se, but we use it to indicate how the structural complexity evolves with respect to the task complexity.

Another study offering insights into the behavior of an agent controlled by both policies is [20], within the context of computer games, which inherently have a deterministic nature. The researchers evaluate the policies by solving a level in the Mario AI benchmark, comparing them on the number of nodes and a reward function $\rho(x)$. This comparison sheds light on the scalability of the two policies, as both are generated by a learning algorithm to tackle the benchmark. The main objective of the study was to propose a new method for generating BTs, making the comparison with FSMs somewhat unfair, since the generation methods for the two policies differ. However, from both [19] and [20] we can conclude that **the complexity scales linearly for BT and quadratically or worse for FSM**.

Authors in [21] instead, propose a combined design where some nodes of the BT are modeled as FSMs. The authors claim that with such design the FSMs allow the policy to easily define states with memory which would instead be modeled in BTs by conditions with complex semantics. A similar combined design, where instead states in an FSM are represented as BT, is proposed in [4]. Having an FSM at the highest level of the representation, allowed human users to control the switching between modes, while the modularity of BTs makes it preferable for the actual robot behavior implementation. A similar hybrid model is implemented in [22] where BTs are combined with HFSMs.

Finally, authors in [23] combine BTs with Stack-of-Tasks (SoT) control, a paradigm that allows the robot to fulfill a list of goals formulated as equality or inequality constraints in error space. Such goals have a priority order and can be fulfilled simultaneously. SoT is usually combined with FSM to prevent the robot from getting stuck in local minima. However, authors in [23] claim that combining SoT with BTs instead solves limitations of the FSMs in terms of reactivity, modularity and re-usability.

III. POLICY DESIGN

To allow for a fair comparison as well as a fast implementation, we selected policy implementations that are based
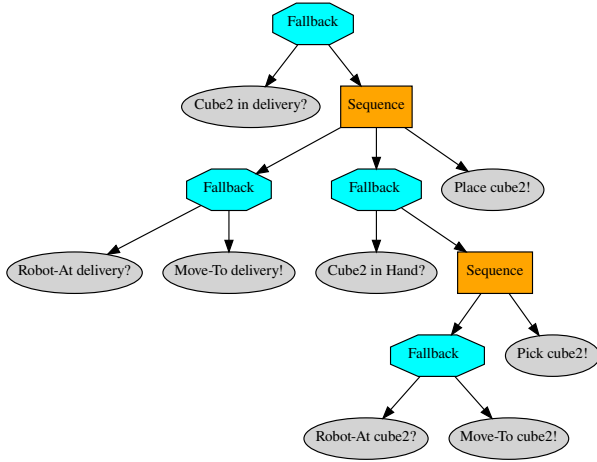
Fig. 3: Backchained BT realizing a chattering controller.

on `python` and are ROS-compatible. For BT design we use `py_trees`[1], while for FSM we use SMACH [24][2].

There exist several methods for automatically generating Behavior Trees, such as Genetic Programming [25], Learning from Demonstration [26], [27], autonomous planners [28], or hybrid combinations [29]–[31]. Some of these generation methods build a BT using a backward-chained approach [27], [28], [31], which is proven to have convergence guarantees [32]. For this method, actions are defined together with their pre- and post-conditions. Starting from the goal, pre-conditions are iteratively expanded with actions that achieve them, i.e. those actions that have that particular condition as one of their post-conditions. Then, those actions' unmet pre-conditions are expanded in the same way. Another property of backchained BTs, is that the control node types are in an alternating order, i.e. the parent of a *Fallback* node is a *Sequence* node and vice-versa, resulting in a design recommended as good practice in [8]. By providing a planner with such a set of actions and a goal state, it is possible to automatically generate backchained BTs (Fig. 1).

In BT design, the order of the actions' pre-conditions is important. In the case of the BT of Fig. 1, for example, if the pre-condition *'Robot-At delivery?'* of the *'Place cube2!'* action is expanded before the pre-condition *'Cube2 in Hand?'*, we would obtain the BT of Fig. 3. A robot controlled with such a BT will feature a chattering behavior because the robot will first navigate to the place position (the condition *'Robot-At delivery?'* will then return *Success*), then it will attempt picking the cube by navigating to the picking pose with the action *'Move-To cube2!'*. As soon as the robot moves, the condition *'Robot-At delivery?'* will immediately return *Failure*, and by consequence re-execute the action *'Move-To delivery!'*. This chattering behavior will continue indefinitely unless there is a limit on the number of ticks allowed. To avoid this problem, the order of the pre-conditions is compared against the order in which actions are executed in the plan, so the pre-condition *'Cube2 in Hand?'* for the picking action is expanded before the pre-condition *'Robot-At delivery?'* for the moving action.

[1] https://github.com/splintered-reality/py_trees
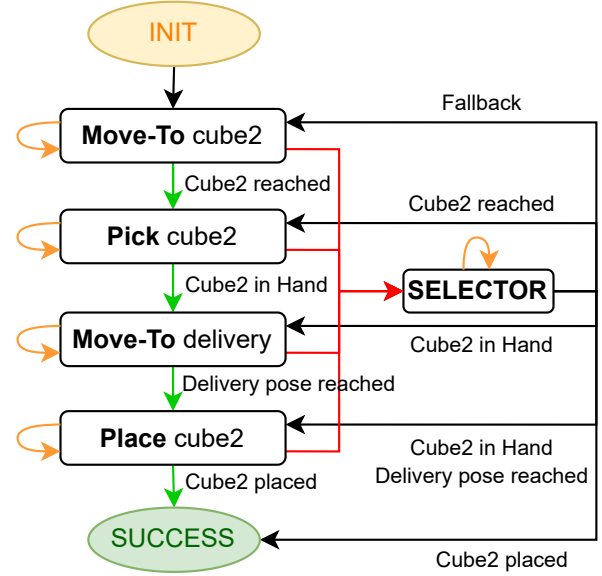[2] https://github.com/ros/executive_smach

Fig. 4: Reactive FSM solving a mobile manipulation task with transitions *Running* (yellow), *Success* (green), and *Failure* (red).
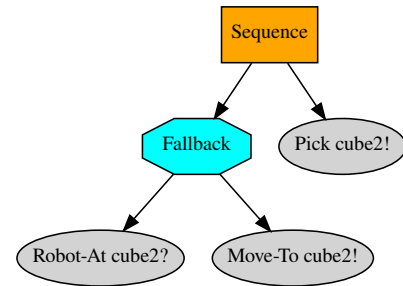

Fig. 5: Example BT subtree with equivalent HFSM in Fig. 6.

Another degree of freedom in backchained BT design is the order of actions for a given post-condition. In this case, a single post-condition would be achieved by different actions, resulting in different subtrees where those actions are expanded. A good design policy would be to sort the subtrees according to e.g., execution time, success probability, or some a priori defined cost. In this direction, authors in [33] assign a cost to every action and at runtime the robot executes the subtree with the lowest cost.

With those inputs the design of a BT is somewhat constrained, while there are multiple design choices for an FSM. A first option is to directly mimic the sequence of actions, as depicted in Fig. 2. This design has the shortcoming that it is not reactive: a robot controlled by such a policy will not react to events that disrupt the task at any given time. Such robot will not be able to recover the execution at any point of the task but only by re-executing from the beginning. To overcome this limitation, it is necessary to turn the FSM into a fully connected graph, taking into account all failure cases, or more simply, to connect every state to a SELECTOR (or failure) state where the state of the robot and the environment is inspected before the next state is reached (Fig. 4). Such an FSM features a *Running* transition that cycles back to the same state, allowing the FSM to preempt an action execution

TABLE I: Overview on the performances of the policies according to the metrics. In a BT n is the number of nodes, $n^\star$ is the number of nodes to modify in an edit operation, and M the number of action nodes. In an FSM n is the number of states and transitions, $n^\star$ is the number of states to modify in an edit operation, while M is the number of states alone. In particular we use the following notation $T_{fc} = M_{fc}(M - 1)$ where M_{fc} is the number of fully connected states in an FSM.

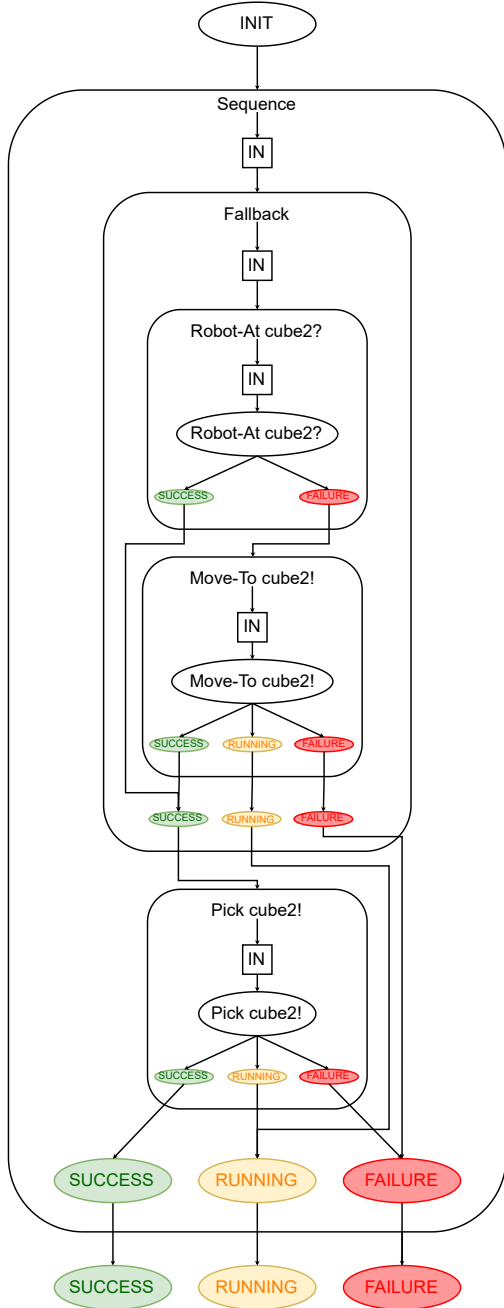| Policy | Metrics | | | | |
| | Modularity | | Reactivity | Readability | |
	Computational Complexity	Graph Edit Distance	Effort	Graphical Elements	Active Elements
Backchained BTs	$\mathcal{O}(1)$	$2n^\star$	0	$\simeq 7M - 1$	$\simeq 3.5M$
Fault-Tolerant FSMs	$\mathcal{O}(n)$	$f(n^\star, n)$	$3(M + 1) + T_{fc}$	$\simeq 5M + 4 + T_{fc}$	$\simeq 5M + 4 + T_{fc}$
HFSMs	$\mathcal{O}(1)$	$kn^\star$	0	$\simeq 36M - 3$	$\simeq 29M - 3$

Fig. 6: HFSM that mimics the subtree in Fig. 5.

provided the execution is asynchronous and the environment is monitored periodically. Then, every execution state has a *Failure* transition to the SELECTOR state, so that the execution can restart in the appropriate state, depending on the task progression. In case the FSM is generated by a planner, the transition from the SELECTOR state to another state is triggered by that state's pre-condition. The other transitions depend on the execution sequence.
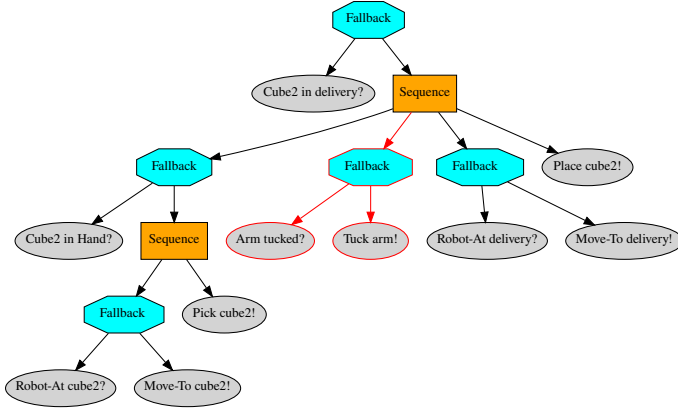
Another design alternative is proposed in [15] where an FSM is formulated as a Hierarchical Finite State Machine (HFSM) that behaves exactly as a BT. Every robot behavior is an FSM with the states *Running, Success, Failure* as outcomes; two or more behaviors are encapsulated in another FSM and transitions are made according to whether the parent FSM mimics a *Sequence* or a *Fallback* node. An example is provided in Fig. 6, where the HFSM mimics the subtree in Fig. 5. In the *Fallback* container, if the condition is successful, the execution is transferred directly to the *Success* outcome of the container, or to the next action in case of failure. In the higher level *Sequence* container, if the *Fallback* container is successful, then the execution is transferred to the next action, or to a *Failure* outcome otherwise.
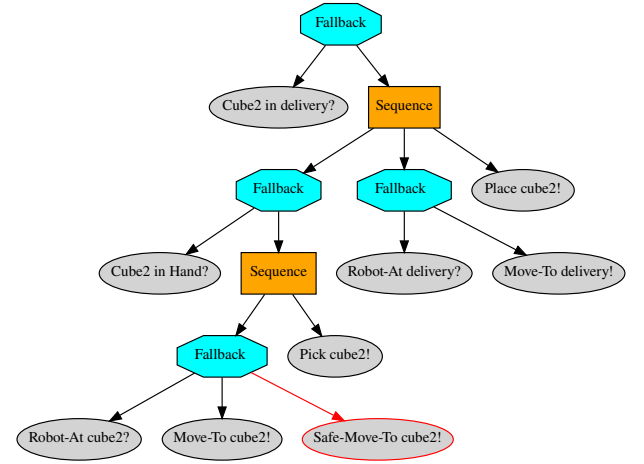
IV. METRICS-BASED COMPARISON

In this section we compare Behavior Trees and Finite State Machines in terms of modularity, reactivity, readability and we conclude it with comments on alternative designs. We will describe each property separately and provide dedicated metrics where applicable, the results are summarized in Table I.

A. Modularity

The possibility of adding, removing, and editing nodes/states, allows the users to reuse pieces of software and to modify bad design choices. It can happen, for example, that during the testing of a policy the user realizes that some behaviors were missing or that some were malfunctioning, and thus need to be removed or replaced. In this section, we will compare the addition, removal, and modification of behaviors in the two decision structures with a focus on the computational complexity of such operations and on the graph edit distance (a measure of diversity between two graphs) between a baseline structure and the modified one. To do so, we will use the baseline structures of Figs. 1 and 4 and we modify them according to the following scenarios:

This article has been accepted for publication in IEEE Transactions on Automation Science and Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TASE.2025.3610090
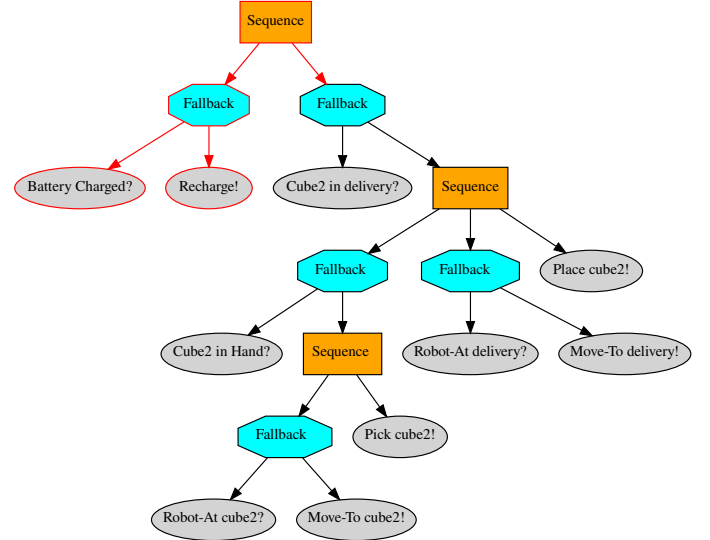
6

(a) Adding the subtree to tuck the arm in the BT: the robot will tuck the arm after grasping.

(b) Adding an alternative way to move to the target cube: if the first Move-To behavior fails, a safer motion is attempted.
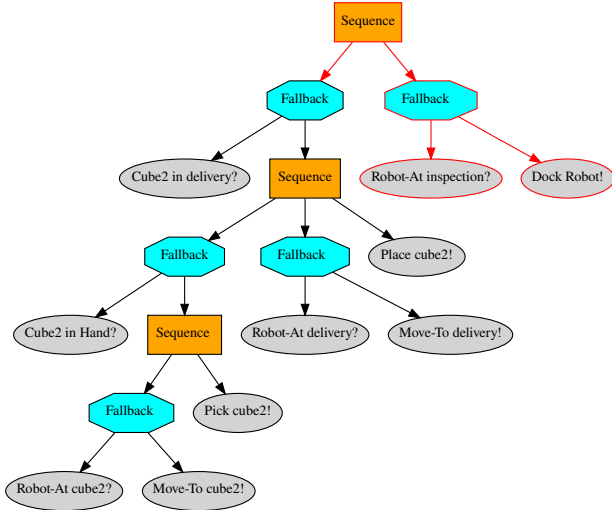
(c) Adding a final behavior for the task in a BT.

(d) A BT that charges the robot if it is low on battery.

(e) BT resulting from all previous node addition cases.

Fig. 7: Different cases of node addition in a BT. The added nodes are highlighted in red.

(a) Adding an intermediate behavior for the task in an FSM: the robot tucks the arm after grasping.
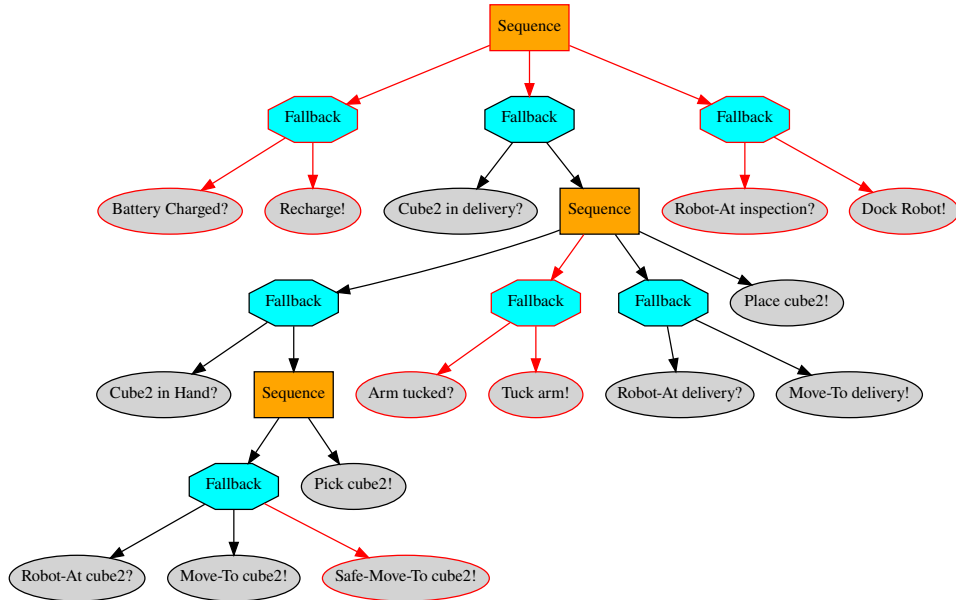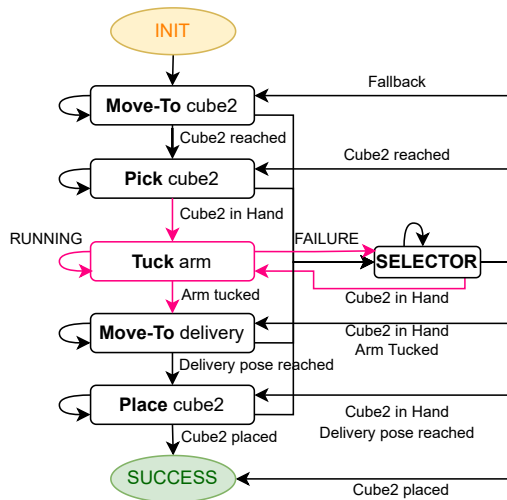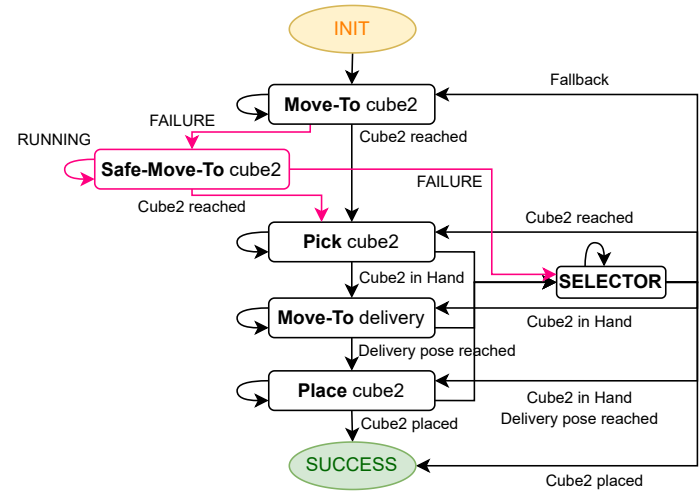
(b) Adding an alternative way to move to the target cube: if the first Move-To behavior fails, a safer motion is attempted.
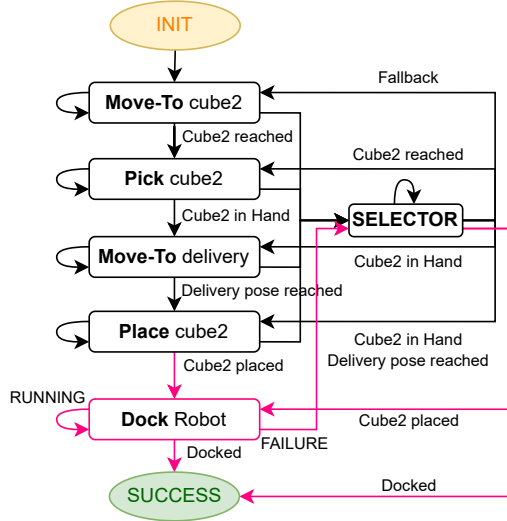
(c) Adding a final behavior for the task in an FSM: the robot docks before finishing the task.
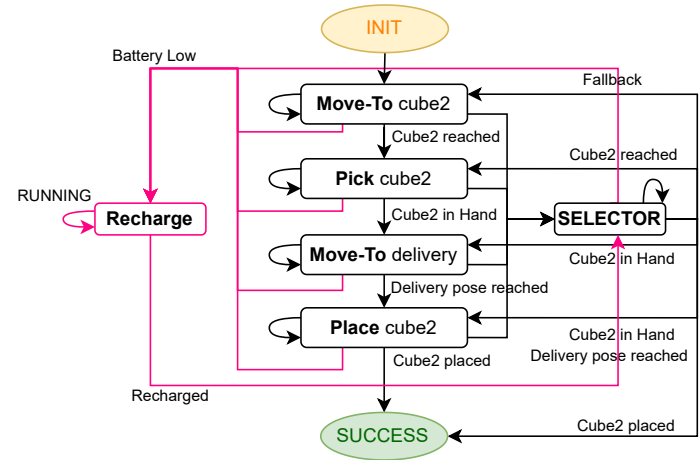
(d) Adding a connected behavior for the task in an FSM: the robot charges the batteries if low.
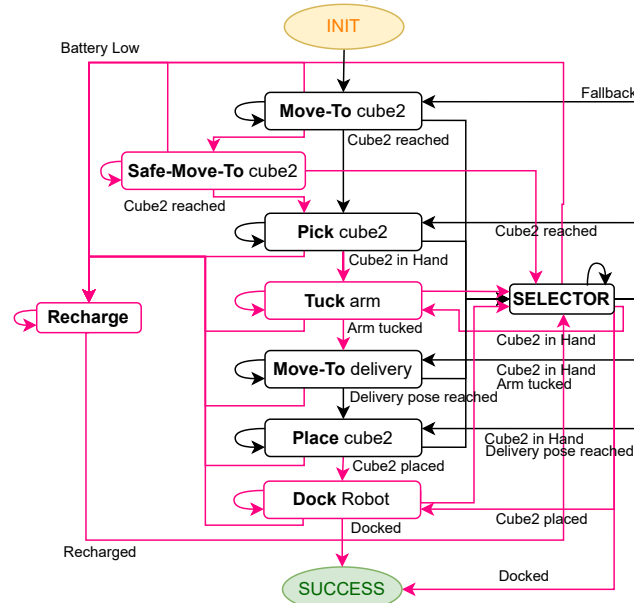
(e) FSM resulting from all previous node addition cases.

Fig. 8: Different cases of state addition in a fault-tolerant FSM. The added elements are displayed in magenta.

1) we add to the baseline a behavior that tucks the robot arm after grasping, to have an arm configuration more suited for navigation;
2) we add an alternative way of reaching the first item to pick, by letting the robot generating a safer trajectory that allows it to move closer to obstacles but with a reduced velocity. This behavior is used when the standard Move-To behavior fails;
3) once the mobile manipulation task is finished, we want the robot to reach a docking station;
4) we make the robot recharge the batteries if they run low at any point during task execution.

The structures that we obtain by modifying the baseline are grouped in Fig. 7 in the case of BTs and Fig. 8 for FSMs.

The first comparison is made in terms of the computational complexity of edit operations in both structures.

1) Computational Complexity:

a) Adding or removing a node in a BT: Adding a node to a BT with n nodes involves inserting it into the list of children of the parent control node at the desired position. In a BT, each child node is independent of other children[3]. The insertion operation requires accessing the desired parent node and inserting the new node, with a complexity of $\mathcal{O}(1)$. Removing a node has the same complexity. Due to modularity, handling a subtree has the same complexity, where the root of the subtree to add/remove is considered as the input node.

There are special cases in which we might desire to prepend or append a subtree to modify the behavior of the robot. For example, if we want the robot to recharge the batteries as soon as they run low, we would need to prepend this subtree in a *Sequence* root node, to the subtree solving the task (so the recharging subtree has higher priority). The BT obtained this way is depicted in Fig. 7d, where the added elements are highlighted in red. In this particular case, a *Sequence* root node is created, then the subtrees to recharge the robot and solve the mobile manipulation tasks are added as children to the root. Similarly, we can append another subtree to the root, as in Fig. 7c, docking the robot after the manipulation task.

b) Adding or removing a state in a FSM: For the proposed FSM design, we identify three possible ways to add a state:

1) **sequential state**: insert a new state between two existing states (the *preceding* and the *following* state). To add the new state as a step in the execution sequence, remove the transition from the *preceding* state to the *following* state, create a transition from *preceding* to the new state, and then from the new state to *following*. An FSM modified this way is shown in Fig. 8a. If the new state is terminal, create a transition to the outcome instead. An FSM modified this way is shown in Fig. 8c.
2) **alternative state**: add a new state as an alternative strategy to an existing one. In this case, the *FAILURE* transition of the *preceding* state goes to the new state instead of the SELECTOR. The transition from *preceding* to *following* is copied to connect the new state to the

---

[3]This does not apply if Blackboard variables are used, but this approach compromises modularity and is not recommended [8], [34].

Algorithm 1: Adding a connected state in an FSM.

input: sm, new_state, condition, selector_condition

1 **for** *state* **in** sm**:**
2   *state*.register_outcome (new_state)
3   **if** *state* == SELECTOR**:**
4   *state*.add_trans (selector_condition)
5   **else:**
6   *state*.add_trans (condition)

7 $new\_trans \leftarrow$ "$RUNNING$" : new_state
  $new\_trans \leftarrow$ "$FAILURE$" : SELECTOR
  sm.add_state (new_state, new_trans)

---

following. Finally, add a *FAILURE* transition from the new state to SELECTOR. An FSM modified this way is shown in Fig. 8b.
3) **connected state**: add a new state for the task that connects to all other states (Algorithm 1). Add a transition from every other state to the new one, and handle the interaction with the SELECTOR state. An FSM modified this way is shown in Fig. 8d.

In an FSM with n states, adding or removing a new state involves checking the consistency between state outcomes and transitions, which has a complexity of $\mathcal{O}(n)$ because there could be up to n transitions to manage.

To remove a state, all transitions to and from that state must be deleted. Additionally, the target state needs to be removed from the outcomes of any other state, if applicable. This operation requires iterating through all transitions and states, resulting in a total complexity of $\mathcal{O}(2n)$. Moreover, since the switching logic in an FSM is implemented inside the state and not explicitly realized by the representation, edit operations require the modification of the program script of the states. For example, if we want to remove a fully connected state, we need to remove or change the transition implementation in all states that reference it.

A key difference with BTs—stemming directly from BT modularity—is that editing an FSM requires access to all states and transitions, making the process of adding or removing an element dependent on all other elements in the structure. Moreover, we have seen that the three insertion scenarios have to be treated slightly differently. These problems are mitigated by using an HFSM. In this case, adding a state has the same complexity as in the BT case. Once the insertion point is identified, the same add_state method as for the FSM is called (Algorithm 1). However, here the consistency is checked locally (i.e., at a specific depth in the hierarchy) so the operation has a complexity $\mathcal{O}(k)$, with k the number of children of the parent node at the insertion point. For example, if we were to insert a subtree to tuck the arm after grasping, as for the BT in Fig. 7a, the add_state method would check for consistency only among the direct children of the parent *Sequence* (i.e., the two *Fallback* nodes and the place action). Transitions would also be handled locally and in a modular way, because we enforced a structure (i.e., three return statuses for every node). However, we trade off modularity for readability, as it can be appreciated by comparing Fig. 5 and 6.

Computational complexity is independent of who implements the algorithms. When editing a BT, modifying a subtree involves removing the edge that connects the parent node to the root node of the subtree and creating a new edge from a different parent node to the subtree. These are a removal and an addition operations, with complexity $\mathcal{O}(1)$ each. In contrast, editing an FSM requires removing and reconnecting all the transitions associated with a state. Both these operations have complexity $\mathcal{O}(n)$ in the worst case of a fully connected graph (as shown in Algorithm 1, one has to loop over all the states in the FSM). Assuming an algorithm or editor facilitates these operations, the workload for a user differs between BTs and FSMs. For a BT, the editing process remains straightforward, as the user performs the same basic operation of reassigning the subtree's connections (1 operation for addition and 1 for removal). In the case of an FSM, while an editor could automate the removal of transitions from the state being edited, the user would still need to manually define and reestablish all the necessary transitions (thus adding $n-1$ transition from a state to all the others, in the worst case of a fully connected graph). This adds a greater workload compared to the relatively simpler operations required for modifying a BT. Remarkably, a connection with modularity and computational complexity is also provided in [18], where authors propose an algorithm to identify all modules in a decision structure.

The computational complexity comparison between BTs and FSMs can also be analyzed in terms of task switching speed, processing speed, and code volume. In the worst-case scenario, both approaches require a linear number of if/else condition checks relative to the number of nodes. For instance, this occurs if all conditions fail in a backchained BT or if a SELECTOR state in an FSM triggers a transition to the last node, necessitating checks for all prior transitions. However, these condition checks are not computationally expensive and do not significantly affect task switching or processing speed. Consequently, the asymptotic complexity of both BTs and FSMs scales linearly with the number of nodes.

Task switching speed also depends on the tick frequency in BTs and the thread frequency in FSMs, both of which are user-configurable. Nevertheless, the associated computational overhead is minimal.

Finally, the amount of code required to implement a given set of robot skills as either a BT behavior or an FSM state depends on the API in use. In FSMs, additional effort is needed to manage state transitions, as each edge in the FSM graph represents a transition that must be explicitly coded. This means that the number of transitions directly influences the code size.

The second comparison is made in terms of the Graph Edit Distance (GED) between a baseline structure and a modified one.

2) Graph Edit Distance: GED is a method for measuring the diversity between graphs and is defined as the minimum number of edit operations (adding, removing, or substituting nodes and edges) needed to transform one graph g_1 into another graph g_2 to make them isomorphic [35]. Using the standard tuple definition of graphs, $G = (V, E)$ with V, E

TABLE II: Graph Edit Distance of the modified BTs in Fig. 7 to the baseline of Fig. 1 and the FSMs of Fig. 8 to the baseline of Fig. 4.

Graph Edit Distance

Modification	BT	FSM	HFSM
Tuck Arm subtree	6	5	12
Safe-Move-To behavior	2	4	4
Dock subtree	8	5	17
Recharge Battery subtree	8	8	17

the sets of vertices and edges respectively, the GED between $g_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$, is defined as:

$$GED(g_1, g_2) = \min_{e_1,\ldots,e_k \in \gamma(g_1,g_2)} \sum_{i=1}^{k} c(e_i), \qquad (2)$$

where c is the cost of the edit operation e_i and $\gamma(g_1, g_2)$ denotes the set of edit paths to transform a graph into the other. For this analysis, we use this measure as implemented in the NetworkX python library [36].

a) Graph Edit Distance in BTs: Considering BTs as a graph, adding or removing one node $n^\star$ contributes with the editing of two elements in the graph, a vertex corresponding to the node and an edge from the node to the parent. For the examples we brought, we have the following:

- the BT in Fig. 1 has 14 nodes and 13 edges;
- the BT in Fig. 7a has 17 nodes and 16 edges;
- the BT in Fig. 7b has 15 nodes and 14 edges;
- the BTs in Fig. 7c and 7d have 18 nodes and 17 edges;

Since we add 3 nodes for Fig. 7a, 1 for Fig. 7b, and 4 for Fig. 7c and 7d, we obtain a graph edit distance of 6, 2, and 8, as reported in Table II.

b) Graph Edit Distance in FSMs: In the case of an FSM, every new state $n^\star$ contributes with one vertex and at least 5 edges (4 for the alternative state case). If however we aim to edit a fully connected state, then the GED depends on the number of the states in the FSM. In such a case, the GED is $4 + n$ (the state $n^\star$, the *RUNNING* transition, transitions to and from SELECTOR, and a transition from every other state to $n^\star$).

Similarly to what done with BTs, we use the measure to compare FSMs of Fig. 8 with the baseline of Fig. 4. Such FSMs are directed graphs with the following elements:

- the FSM in Fig. 4 has 6 nodes and 18 edges;
- the FSMs in Fig. 8a and 8c have 7 nodes and 22 edges;
- the FSMs in Fig. 8b has 7 nodes and 21 edges;
- the FSM in Fig. 8d has 7 nodes and 25 edges.

Where the outcome *SUCCESS* is also considered as a node. We observe that adding a state in some intermediate point in the execution sequence (Fig. 8a) has no substantial differences as compared to adding it as a final state (Fig. 8c) The values for the graph edit distances are reported in Table II.

c) Graph Edit Distance in HFSMs: We make a similar but briefer analysis on HFSMs. With the example portrayed in Fig. 6, we obtained the graph representation by converting every node as it follows:

- an action node is a subgraph with 1 vertex and 3 edges. The vertex being the node itself and the edges being the 3 transitions, one from each return status to the next child or the parent return statuses;
- a condition node is similar to an action node, with one edge less because conditions do not return *RUNNING*;
- a control node contributes with 1 vertex and 4 edges. The additional edge is the transition from the node to the first child.
- we add 3 more states to account for the outcome of the execution, one for each return status.

We can thus compute the GED between two HFSMs as a function of the difference between their structural components:

$$GED(hfsm_1, hfsm_2) = 3\Delta c + 4\Delta a + 5\Delta i, \quad (3)$$

where c are condition nodes, a are action nodes, and i are internal or control flow nodes.

With such design, we disregard the IN elements and the internal return statuses. Taking as example the HFSM that we would obtain from the BT in Fig. 1, we would have:

- 4 *Fallback* nodes, contributing 1 vertex and 4 edges each, for a total of 4 vertices and 16 edges;
- 2 *Sequence* nodes, contributing 2 vertices and 4 edges each, for a total of 2 vertices and 8 edges;
- 4 *Action* nodes, contributing 1 vertex and 3 edges each, for a total of 4 vertices and 12 edges.;
- 4 *Condition* nodes, contributing 1 vertex and 2 edges each, for a total of 4 vertices and 8 edges;

This makes a total of 17 vertices and 44 edges. To resume:

- the HFSM representing the BT in Fig. 1 has 17 vertices and 44 edges;
- the HFSM representing the BT in Fig. 7a has 20 vertices and 53 edges;
- the HFSM representing the BT in Fig. 7b has 18 vertices and 47 edges;
- the HFSM representing the BTs in Fig. 7c and 7d have 21 vertices and 57 edges;

The minimum GED for the considered examples is 4 when we add the behavior to move more safely, to a maximum of 17 when we add the subtrees for docking or recharging the batteries, as reported in detail in Table II.

To conclude, in Fig.7e and 8e we show the BT and FSM resulting from adding all the nodes described in the previous examples, respectively. It can be seen that the number of edits required to perform in an FSM increases with the number of states added and it soon becomes cumbersome to maintain, while this maintenance cost is mitigated in the case of a BT.

Another example that highlights the benefits of modularity in a BT, is when we need to add behaviors with high priority and later decide the priority order among them. In particular, we consider as a baseline the BT in Fig. 7d and the FSM in Fig. 8d. We now want the robot to shut down in case of emergency. We obtain the policies in Fig. 9 where in the BT the subtree realizing the mobile manipulation task of Fig. 1 is collapsed into anode called *'Moma Task!'* for a better visualization and to stress that editing a BT does not modify other behaviors in the tree. As another example, we could design a high priority fallback subtree with a condition node that checks if the robot is connected to the network and an action node that attempts reconnecting to the network if the connection is lost. Since a BT executes with tick signals that propagate from left to right, the subtree with the highest priority is placed in the leftmost part of the tree. To change the priority order it is enough to re-arrange the subtrees in the list of the children of the root node. In the case of the FSM, the priority depends on how the transitions are designed. If we want to change the priority order in the FSM of Fig. 9b, we would need to edit every implementation of the states to trigger the transition to 'Recharge!' before the one to 'Shut Down!', then to make a transition from 'Shut Down!' to 'Recharge!'. From a code design perspective, having copies of the same instruction block in multiple places is a source of error.

B. Reactivity

A reactive control policy should achieve two behaviors:

1) if during the task execution, part of the task is done or undone by an external agent or disturbance, the robot should react to it, either by jumping over the already executed steps, or by redoing those steps that have been undone;
2) if during task execution, an action with a higher priority needs to be executed (i.e. recharge the robot if it's low on battery), the robot should interrupt the action that is currently executing to run the one with higher priority instead.

Taking as example the BT in Fig. 7d, where a subtree for handling battery recharge has been added (additional elements highlighted in red), it achieves both behaviors:

1) if the place pose is in the neighborhood of the grasping pose, then the robot wouldn't need to execute the second navigation action as its condition would already be satisfied, thus jumping directly to the execution of the place action. The BT is ticked also if the root returns *Success*, so if the robot completes the task but then an agent moves the cube from its target pose, then the robot would attempt grasping and then placing again.
2) subtrees with high priority actions are found in the top left part of the tree. Since a BT is recursively ticked from left to right and nodes return *Running*, any action that the robot is executing can be preempted as soon as the condition monitoring the battery status returns *Failure*, thus transferring the execution to the 'Recharge!' action.

In an FSM, the only way to achieve the same level of reactivity, is to have a transition from any state to every other, or to adopt a solution like the one proposed in Section III. This is manageable for small problems, but it quickly grows out of hand if the task is complex. Moreover, with such a choice, modularity is sacrificed. To quantify the price for reactivity, we can count the number of operations required to transform the sequential FSM of Fig. 2 into the reactive version of Fig. 4. The former has 5 nodes and 4 edges, while the latter 6 nodes and 18 edges. The price for the transformation is a total of 15 operations (obtained by using the Graph Edit Distance
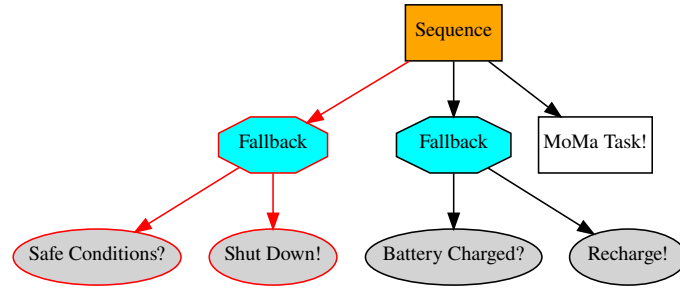
(a) Adding the a high priority subtree to a BT does not affect the other behaviors in the policy.
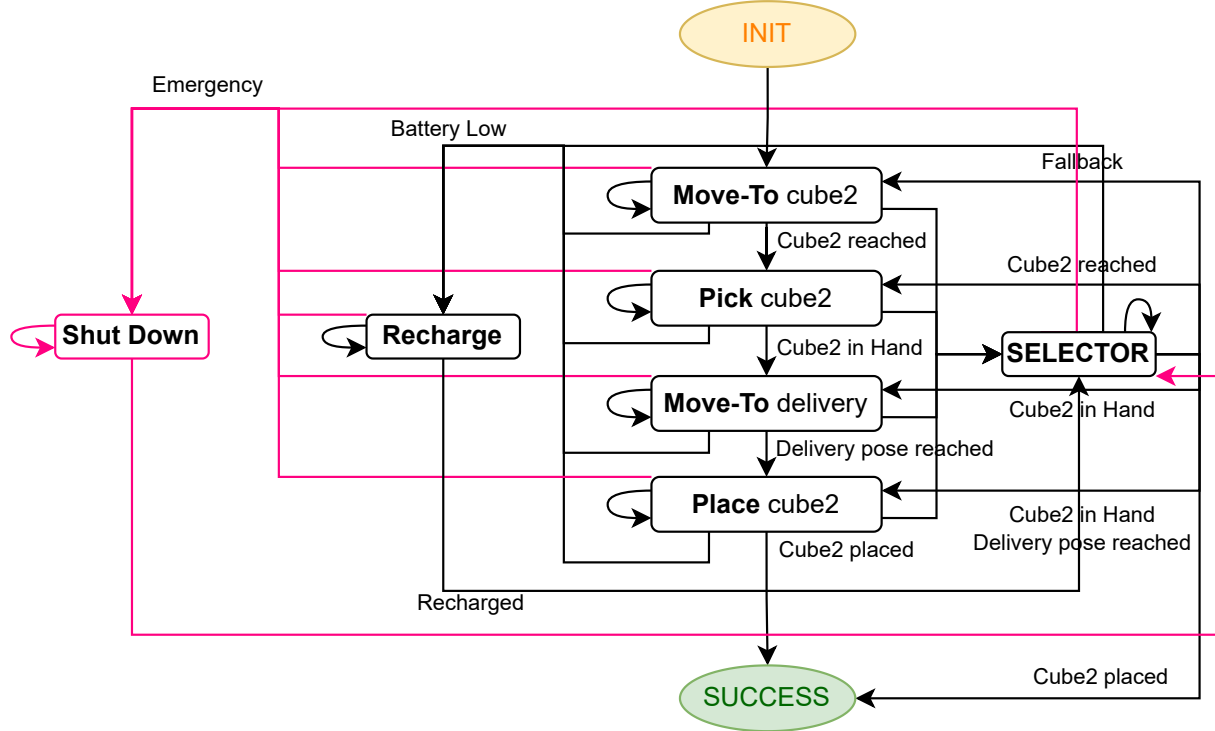
(b) Adding connected states to the FSM complexifies the structures and the maintenability of the code.

Fig. 9: Editing high priority behaviors in a policy representation: modularity allows to interchange the priority of subtrees in a BT without requiring to modify other behaviors in the policy.
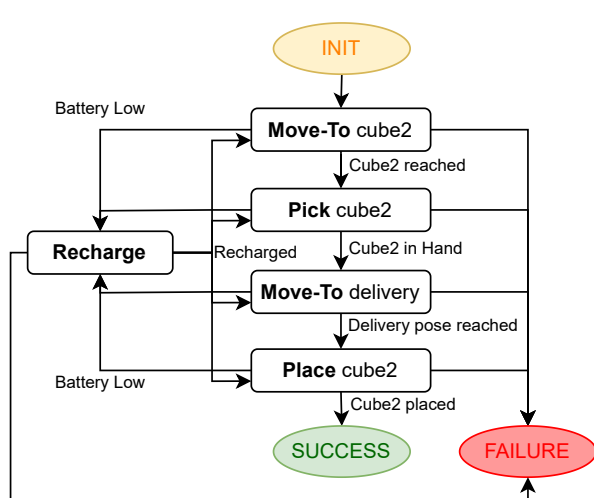

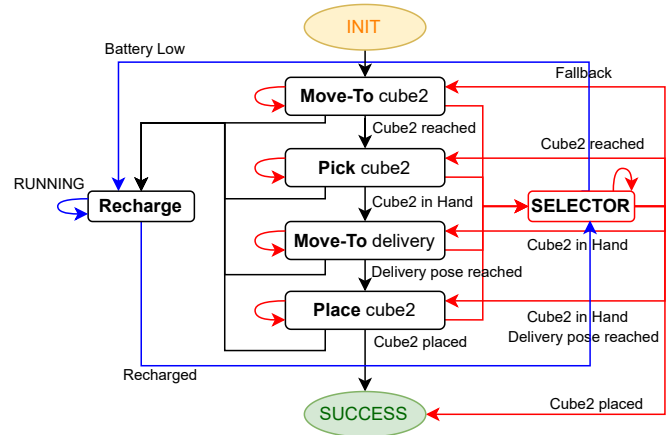Fig. 10: Non reactive FSM with $M_s = 4$ and $M_{fc} = 1$.

Fig. 11: Reactive version of the FSM in Fig. 10. This is the same FSM of Fig. 8d, were we highlight different elements. The red and blue elements correspond to the operations related to M_s and M_{fc} in Equation 4, respectively.

measure). We can define this measure as *Effort* and make it a function of the number of states. To transform an FSM with M_s sequential states and M_{fc} fully connected states in a reactive FSM, with our proposed design, the effort is:

$$E(M_s, M_{fc}) = 3(M_s + 1) + M_{fc}[(M_s + M_{fc} - 1) + 3]. \quad (4)$$

Let's consider the sequential FSM example of Fig. 10 with $M_s = 4$ and $M_{fc} = 1$. Here, the fully connected state is the 'Recharge' state. The effort to transform this FSM into the fault-tolerant version of Fig. 11 requires the following operations:

1) add the SELECTOR state plus a *'RUNNING'* self-transition and a transition to the *SUCCESS* outcome—3 operations;
2) for each M_s: add a *RUNNING* self-transition, a *FAILURE* transition from the state to SELECTOR and another one from SELECTOR back to the state to resume the execution—3 operations;
3) for each M_{fc}: remove transitions from this state to any other state—$(M_s + M_{fc} - 1)$ operations—then add the interaction to SELECTOR as in the previous point—3 operations.

Defining $M = M_s + M_{fc}$, Equation 4 can be rewritten as:

$$E(M, M_{fc}) = 3(M + 1) + M_{fc}(M - 1). \quad (5)$$

To conclude, BTs are naturally reactive, so $E = 0$, independently on the number of nodes. The same holds for HFSM, that are reactive by construction as we enforced a structure.

C. Readability

Readability is an inherently subjective measure, as it depends on user preferences and how easily a human operator can comprehend and debug the agent's behavior simply by examining the decision structure. This definition highlights the importance of facilitating the tracking of the robot's execution. While subjective assessments could provide valuable insights, they would require large-scale user studies. Such studies, however, would be influenced by factors like the users' expertise, the specific task the agent is solving, and other contextual variables, making the results difficult to generalize. Therefore, to compare BTs and FSMs in terms of readability, we aim to use an objective measure that quantifies how the size of the policy grows with increasing task complexity by computing the number of graphical elements in policy as a function of the robot skills (action behaviors in a BT and states in an FSM). A backchained BT (like the one in Fig. 1) would usually have as many action behaviors as conditions (unless an action has two or more post-conditions or the same post-condition is achieved by more than one action, as in the case of Fig. 7b). Then, since every post-condition is coupled with an action achieving that, we would have a *Fallback* node to make each coupling. Finally, the number of *Sequence* nodes is variable, as it depends on the number of actions that require expansion (i.e., those actions that have pre-conditions). We can assume as a rule of thumb, that the number of *Sequence* nodes is one half of the number of action behaviors as in the backchained design a sequence node would at least have two action nodes

among their children [28]. Thus, if we express the BT design complexity for M action nodes, we would have a total of $N = 3.5M$ nodes. Those will count for *Active Elements* in Table I (i.e., those elements that can be manipulated). If we had to consider all graphical elements, we should include every edge connecting each parent node to its children, for a total of $T = N - 1$ connections. This makes it a total of $S = N + T = 7M - 1$ elements.

Using the same analysis for an FSM, for every state we would have a transition connecting it to the next one, with the last transitioning to a final *Success* state, or the outcome of the task. To achieve full reactivity, we would need a *Running* self-transition for each state. An additional SELECTOR state is also needed to allow the robot to reactively restart the execution where it stopped in case of errors, with a transition to this state from every other state. With such a design, the FSM described by M action states, would have a total of $N = M + 1$ nodes (we add the SELECTOR state). Then we have the following transitions:

- N *Running* transitions, one per state;
- $N - 1 = M$ *Failure* transitions, one per action state;
- $N - 1 = M$ transitions from the SELECTOR state to any other action state to resume execution;
- M transitions from every action state to the next;
- 1 more transitions connecting the SELECTOR state to the outcome.
- $M - 1$ more transitions for every fully connected state M_{fc}, thus amounting for $T_{fc} = M_{fc}(M - 1)$

This makes a total of $T + T_{fc}$ transitions where $T = N + 3M + 1 = 4M + 2$ and a total of $S = T + N + 1 + T_{fc} = 5M + 4 + T_{fc}$ elements, where we also added the outcome state. An alternative design representing a reactive FSM as a fully connected graph, would have instead $N - 1$ transitions for every state, making it a total of $S = M(M - 1)$ elements.

Since a HFSM is built from a BT, we will have one condition and one fallback node for every action and approximately half of it for sequence nodes. This means M actions, M conditions, M fallback nodes and $0.5M$ sequence nodes. To compute the *Active Elements* in a HFSM, we can use the same analysis we described to build the graph, in Section IV-A, where actions contribute for 10 elements, conditions for 7, control nodes for 8. For the *Graphical Elements* we need to add 2 elements to each node to take into account the IN state plus a transition. Finally, for both cases we remove 3 transitions from the root return statuses. We thus obtain the numbers reported in Table I.

Even if performing an user study is outside the scope of this paper, we can still describe how an operator would behave if they had to supervise an agent controlled both by an FSM and a BT. Since for every state of an FSM, the possible next robot actions are univocally determined by the transitions from that state to the others, the operator has a clear understanding of what is going to happen. This is due to the fact that the execution in an FSM is causal (i.e., the execution of a state depends on the past events). With a BT instead, because of reactivity and the fact that at every iteration the execution resumes from the root, any action can be preempted and the execution flow can jump to any other part in the tree. To
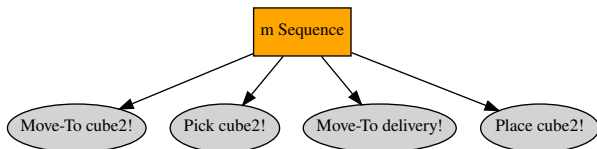
Fig. 12: A BT rooted with a *Memory Sequence* node, achieving open loop control.
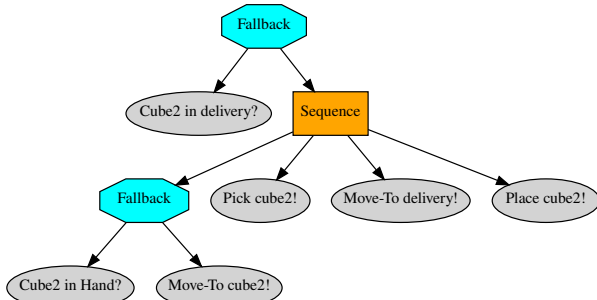

Fig. 13: A BT that could be generated by Genetic Programming, according to the results in [25].

adapt to this behavior of the decision structure, an operator would certainly require more training. There are however functionalities that allow highlighting the currently executing behavior, together with its return status, to greatly support human supervision. As an example, with the modified version of `py_trees` provided by authors in [7][4], at runtime the currently running behavior is highlighted in yellow, successful ones in green, and failed ones in red. In this way, tracking the robot execution is easier.

Finally, even if the preference of a tree structure over a graph structure is subjective, the reader must remember that in the examples provided here, the robot has to achieve a very simple mobile manipulation task, so the graph representation might lose clarity with increasing task complexity since the number of intersections among the transitions will increase. With reference to Section V for an example, we highlight the fact that in a tree structure it is possible to collapse subtrees into single nodes (as in Fig. 16a) to remove detail complexity. This is not a feasible option in FSMs due to the many transitions (as seen in Fig. 16b), unless states are organised in hierarchies but with the limitations that HFSMs introduce (as explained in Section II).

D. Comments on alternative designs

FSMs have been around for much longer than BTs, becoming a standard in the design of control policies for robotics. Their success is mainly due to their design simplicity, which allows for intuitive understanding of the robot behavior. An FSM like the one in Fig. 2 solves a mobile manipulation task sequentially, without granting fault tolerance or reactivity towards unforeseen events. Upon failure, the whole sequence has to be executed again. This design realizes a feed-forward control, as any feedback from the sensors is used as long as it triggers a transition, any unforeseen disturbance will not result in any transition, thus leading to a failure of the controller.

[4]The code repository found under https://github.com/ethz-asl/bt_fsm_comparison.

If it is argued that simplicity and readability of the design are the most important aspects of a control policy, then the same feed-forward control can be achieved by a BT as well. By introducing the *Memory* node (with symbol *mSequence* or *Sequence**, instead of a *Sequence* node) that remembers the state of the last executed child (thus not executing again a child that had previously succeeded), we would design a BT like the one in Fig. 12. If feed-forward control is desired, there is no benefit in using BTs over FSMs [8].

Throughout this paper we have used a backchained BT design because of the associated convergence guarantees [32], but other designs may guarantee a more compact BT. One example is a BT generated by the Genetic Programming (GP) method proposed in [25] and used later in [30], [31]. The learning method takes as input a task goal and a set of robot actions and conditions, then it evolves BTs to solve a mobile manipulation task similar to the one proposed as example for the analysis. If we were to use the same set of actions and conditions of this paper in the GP method, we would obtain a BT like the one in Fig. 13. We can observe that the GP learns to protect the action node driving the robot to the cube with a condition node (here, the condition checks that the cube is grasped), to avoid the chattering problem that we have previously discussed. This solution has the same properties of reactivity and modularity as the backchained one (as it is an inherent property of all BTs), but it is more compact and simpler as compared to its direct reactive FSM counterpart of Fig. 4 (9 nodes and 8 edges versus 6 nodes and 18 edges).

With that being said, it can be argued that a steeper learning curve is associated with using BTs. Thus, if the aim is to generate a control policy for a short task where the robot operates in a controlled environment, FSMs are likely to be easier to conceive and realize. If however the aim is to generate a robust control policy that might be subsequently modified or re-used, then BTs have a clear advantage. The design complexity can be mitigated by automatic generation methods that are already available in literature.

V. EXPERIMENTS AND RESULTS

In this Section, we conduct a comprehensive comparison between BTs and FSMs in the context of controlling a robot for a simulated mobile manipulation (MoMa) task by evaluating the metrics of Table I in concrete examples, with results summarized in Table III .

We target tasks that are typical and representative of the MoMa domain. Our primary focus is on the **Cleaning Up** task, a variant of the RoboCup@Home benchmark [37], wherein the robot is tasked with collecting unknown objects scattered throughout a known environment. The complexity of the task requires the robot to navigate, recognize objects, manipulate them, and place them in a predefined area.

We perform three sets of experiments, where the first and the second are taken from [7]. In the first set of experiments (Experiments V-B to V-D), we simplify the task to fetching a single known object, enabling a clear evaluation of both BTs and FSMs. The evaluation methodology involves pairs of experiments, progressively increasing in complexity as new
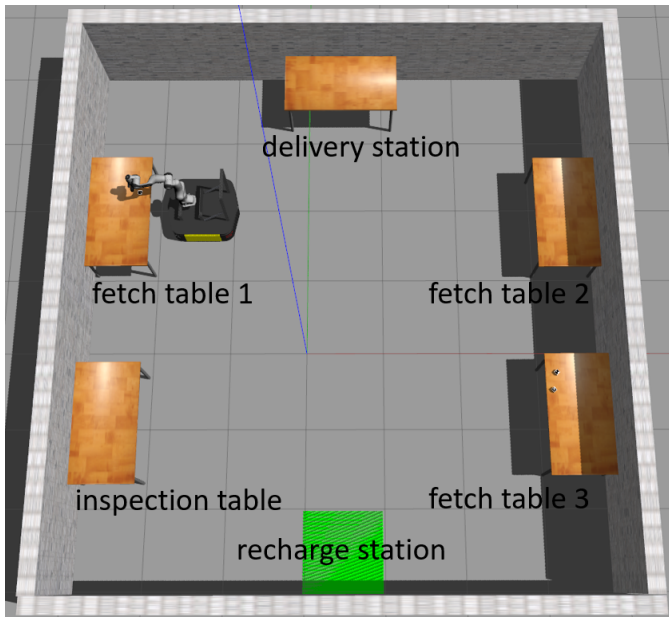
Fig. 14: Simulation environment with the mobile manipulator implemented in the Gazebo simulator.

behaviors are incorporated into the policy. In this way, we wish to simulate a scenario where an engineer has to develop the policy representation for the task, starting from the MoMa baseline represented by the BT of Fig. 1 and the FSM of Fig. 4, and progressively modifying them. To augment the task's complexity, two specific scenarios are introduced:

1) **Battery Recharge Scenario:** The robot is required to recharge its batteries if they deplete at any point during task execution.
2) **Docking Scenario:** Upon completing the task, the robot must dock at a specified location.

The results of these experiments are supported by the accompanying video published with [7][5].

In the second set of experiments (Experiment V-E) we focus on scalability: the effort required to modify the structures if they are already complex. Finally, in Experiment V-F we expand experiments in [7] by transferring the simulated task on a real scenario.

The simulation environment (shown in Fig. 14) features a robot comprising a Franka Emika Panda arm mounted on an omnidirectional Clearpath Ridgeback base. Equipped with an Intel RealSense RGBD camera, the arm's end-effector is instrumental in executing the manipulation tasks. In the simulation, the robot docks at the *'inspection table'* and places the object at the *'delivery station'*.

A. Skill Description

As the paper's primary focus is on comparing the high-level behavior of control policies, certain assumptions are made to streamline the execution of low-level skills. In terms of navigation, the robot is constrained to a specific set of target poses, with each pose corresponding to a station or table within the environment of Fig. 14. The navigation functionality is

[5]https://www.youtube.com/watch?v=Z0GAkClVx-I

realized through the utilization of the ROS Navigation Stack. AprilTags are employed to locate items within the environment. The trajectory planning for manipulation is implemented using ROS MoveIt!. To compute the grasping points, a shape-based grasp synthesis method named the Volumetric Grasping Network (VGN) [38] is employed.

Within the ROS framework, skills are instantiated as action servers. A client interface enables the interaction between the policy (BT or FSM) and the low level implementation of the skills in terms of initialization, goal transmission, status monitoring during execution, and goal cancellation.

We define the action skills and stauts checks as it follows:

a) Move-To!: this action allows the robot to navigate to a target pose given explicitly or reconstructed from the object marker.

b) Robot-At?: this condition checks that the robot reached the desired pose with the desired $[x, y, yaw]$ tolerance.

c) Pick!: this action moves the robot arm to a target pre-grasp pose on top of the target object. Then, VGN is used to compute the grasping pose that is sent to the robot arm. Once the grasp is successful, the arm is tucked in a compact configuration for navigation.

d) In Hand?: this condition checks if the robot is holding an object.

e) Place!: this action moves the robot arm to a target pose and drops the object. Then the robot moves the arm to a configuration that allows it to monitor the scene and evaluate the end pose of the object with the wrist camera.

f) Object-At?: this condition checks that the object is at the desired pose with the desired $[x, y, z]$ tolerance.

g) Recharge!: this action makes the robot navigate to the recharge station and then it instantaneously fills up the battery level.

h) Battery Lv?: this condition determines when the robot shall recharge its batteries.

i) Dock!: this action makes the robot dock at the *'inspection table'*. The achievement of the action is monitored by 'Robot-At?'.

For the Scalability experiment, we also include the following:

j) Search!: this action makes the robot follow 5 view-points (one per table in Fig. 14), where the robot stops and inspects the table with the camera to look for markers.

k) Found?: this condition determines which markers to search for and by consequence to stop the search behavior once they are all found.

B. Experiment 1: Baseline

In this experimental scenario, the robot is tasked with retrieving a cube located on *'fetch table 1'* as illustrated in Fig. 14, and transporting it to the designated *'delivery station'*. The robot initiates the task from the room's center, and the cube's predetermined position is known in advance. This specific task is also attempted using the sequential FSM depicted in Fig. 2. The BT and the fault-tolerant FSM employed for solving the task are presented in Fig. 1 and Fig. 4, respectively.

The sequential FSM successfully accomplishes the task only under the condition that all actions execute correctly and

TABLE III: Performances of the policies designed for the experiments according to the metrics defined in Table I. In the 'Development' experiment, we gradually complexify the baseline structure simulating an engineer that progressively adds features to the base design (numbers in parenthesis quantify the steps such additions require). We first add a recharging behavior and then a docking one. In the 'Scalability' experiment, we add a recharging behavior to a more complex baseline structure.

Experiment		Metrics (BT/FSM)			
		Cyclomatic Complexity (CC)	Graph Edit Distance (GED)	Graphical Elements (+/-)	Active Elements (+/-)
Development	Baseline	1 / 14	- / -	27 / 24	14 / 24
	Recharge	1 / 20	8 / 8	35 (+8) / 32 (+8)	18 (+4) / 32 (+8)
	Docking	1 / 24	6 / 8	41 (+6) / 38 (+6)	21 (+3) / 38 (+6)
Scalability	Baseline	1 / 68	- / -	153 / 114	77 / 114
	Recharge	1 / 92	6 / 26	159 (+6) / 140 (+26)	80 (+3) / 140 (+26)

within defined tolerances. Failures during execution necessitate a task reset. In contrast, the BT and the fault-tolerant FSM reattempt failed actions. Additionally, due to the recursive ticking from the root in the BT, it continues to execute even upon successful completion. This characteristic enables the robot to respond to unexpected changes, such as a human operator relocating the cube to another table after task completion, allowing the robot to adapt and retrieve the cube again, provided it knows its updated location.

In terms of computational complexity when adding behaviors to the base task, the BT in Fig. 1 consists of a graph with 14 nodes and 13 edges, while the FSM in Fig. 4 has 6 nodes and 18 edges, with a cyclomatic complexity of 14.

C. Experiment 2: Recharge Scenario

In this experiment, a recharging behavior is introduced to the existing manipulation task, necessitating the engineer to modify the policy as depicted in Fig. 7d and Fig. 8d. In the case of the BT, the engineer needs to insert the recharge subtree into a new *Sequence* root node, prioritizing the recharging subtree over the subtree handling the manipulation task. This involves eight elementary operations: creating the four nodes (root node and nodes in the left subtree) in Fig. 7d, adding two leaves to the *Fallback* node, and finally incorporating the recharge subtree and the pre-existing BT for the MoMa task into the new root.

It's important to note that the assumption is made that the recharge behavior instantaneously replenishes the batteries. This assumption could be justified by scenarios where an operator swiftly switches the robot's batteries when it halts at the recharge station. If the batteries are gradually recharged while the robot remains in the recharge station, the recharge subtree could be designed differently to preempt charging if a more critical task arises. An example of the BT implementation of this logic is proposed in [39]. The BT in Fig. 7d comprises 18 nodes and 17 edges, resulting in an ED of 8 with respect to the baseline.

For the fault-tolerant FSM, eight operations are also required: creating the recharge state and the *Running* transition, adding a transition from the recharge state to the SELECTOR state, and from every state to the recharge state. Two considerations emerge. Firstly, the number of new transitions

depends on the number of existing nodes, unlike the BT where the operation count is independent of tree size. Secondly, the internal logic of the states must be adjusted to implement the case triggering a transition to the new state. This underscores a distinct advantage of BTs, where the switching logic is explicitly implemented in the representation. The FSM in Fig. 8d encompasses 7 nodes and 25 edges, resulting in an ED of 8 with respect to the baseline and has a cyclomatic complexity of 20.

From a behavioral standpoint, the task execution remains unchanged, and the robot successfully proceeds to recharge the batteries if their level falls below 20%.

D. Experiment 3: Docking Scenario

In this instance, we introduce a docking behavior, as illustrated in Fig. 15. Since the *Sequence* root node already exists, editing the BT entails: (1) creating the *Fallback* node, the condition, and the action, (2) adding the two leaves to the control node, and (3) appending the subtree to the root. Once again, the remainder of the tree remains unaffected by this addition.

For the FSM, apart from creating the new state, the *Running* transition, and the transitions to and from the SELECTOR state, it is necessary to eliminate the transition from the *'Place'* state to the *Success* outcome and create a transition from the *'Dock'* state to the outcome.

In this experiment, similarly to the preceding cases, there is no significant divergence in the executed robot behavior between the two policies. The BT consists of 21 nodes and 20 edges, resulting in an $ED = 6$ compared to the previous case of Experiment V-C. On the other hand, the FSM comprises 8 states and 30 transitions, yielding an $ED = 6$ and $CC = 24$.

E. Experiment 4: Scalability

In the previous experiments, we deliberately focused on a scenario involving the retrieval of a single item. While this simplified task allowed for a detailed examination of implementation and programming efforts, it may not fully capture the advantages of modularity that BTs offer over FSMs. As we've noted, the insertion operations in FSMs are dependent on the existing number of states and transitions within the FSM.

This article has been accepted for publication in IEEE Transactions on Automation Science and Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TASE.2025.3610090
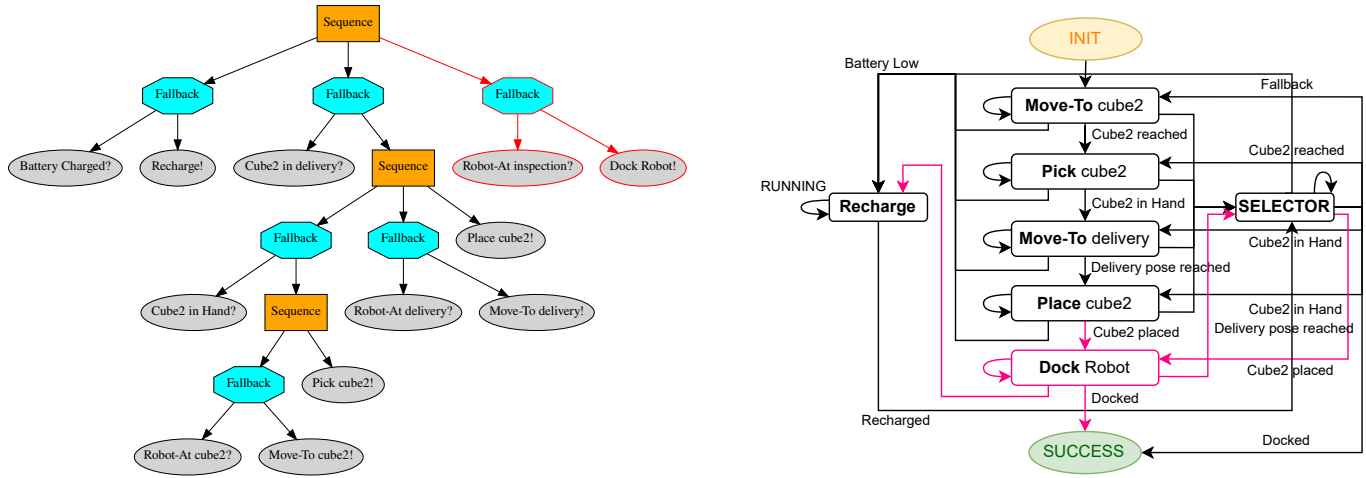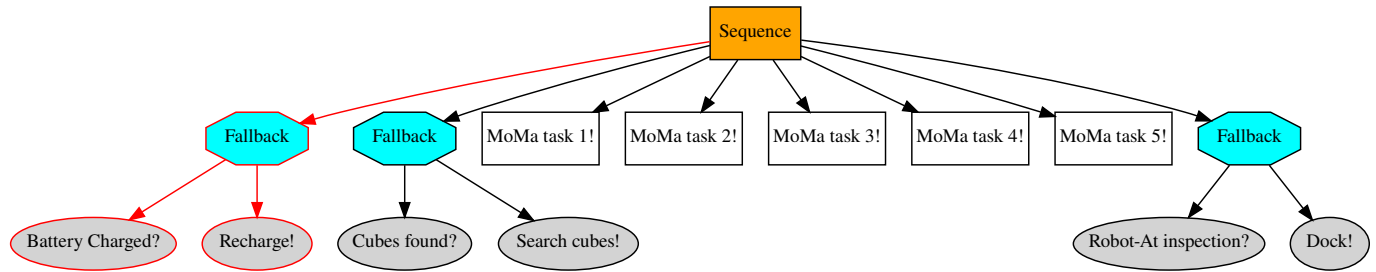
16

Fig. 15: BT and FSM for the Experiment 3 obtained by adding a docking behavior to the representation of Experiment 2.

(a) BT fetching five cubes, comprising the recharge and docking behaviors.
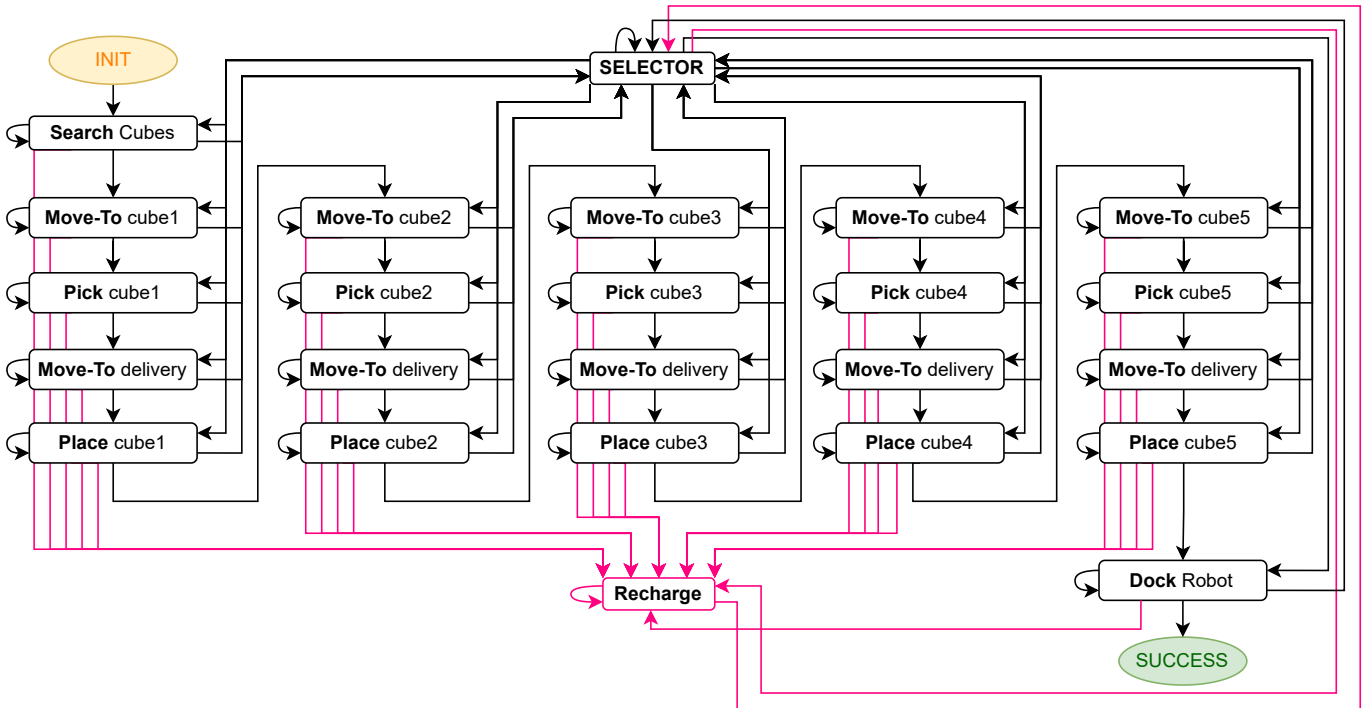
(b) FSM fetching five cubes, comprising the recharge and docking behaviors. Transitions are omitted to improve readability.

Fig. 16: **Experiment 4: Scalability.** In FSMs, adding connected states has complexity dependant on the number of states in the representation, resulting in hard to read structures. In BTs, this complexity is constant.

Expanding the task complexity to a scenario where the robot is required to search for and fetch five cubes before docking, we observe a BT with 77 nodes and 76 edges, and an FSM with 24 nodes and 90 transitions, considering the presented set of skills. At this point, incorporating a recharge behavior into the existing policy representations becomes more cumbersome for an FSM. This is because adding the transition to the recharge state from all other states is necessary. Consequently, the BT would have an $ED = 6$ (similar to Experiment V-D, where the root is already present), while the FSM would have an $ED = 26$. This is due to the final BT (shown in Fig. 16a) featuring 80 nodes and 79 edges, and the final FSM (shown in Fig. 16b) comprising 25 nodes and 115 transitions.
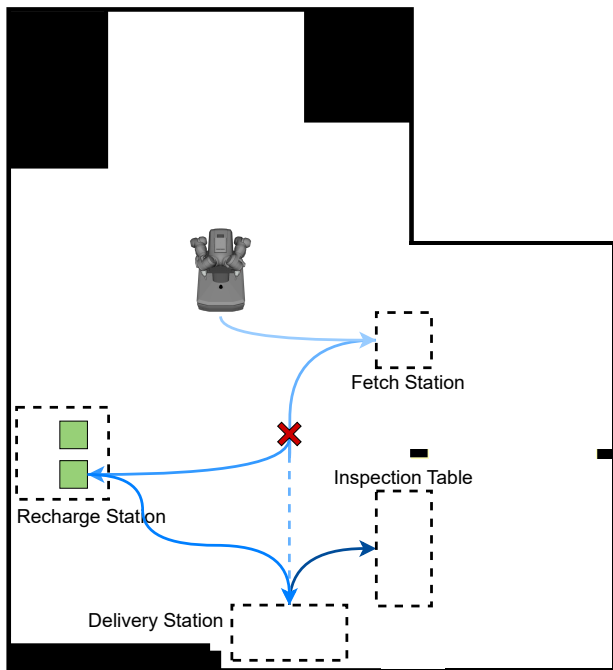
F. Experiment 5: Real Case Scenario

Fig. 17: A schematic representation of the execution of Experiment 5 in the laboratory environment of the WARA Robotics. The solid black lines and areas represent walls and non accessible spaces and are therefore considered as obstacles by the robot. The arrows represent the trajectory followed by the robot during the task.

In this experiment we implement the same policies of Fig.15 from Experiment 3 on a real robotic system. The goal of this experiment is to show the following:

1) since the policies are a plan at the task level, their representation as a BT or an FSM are platform-agnostic. Only the low level implementation of the skills are platform-dependent.
2) as shown in the previous experiments, provided that the implementation of the skills is the same, the robot behaves similarly if controlled by a BT or an FSM.

Algorithm 2: Pseudo-code of the MoveTo behavior in the BT.

```
1 def initialise(skill, goal, reference):
2     skill.move(goal, reference)

3 def update(skill, goal):
4     if not skill.navigation_done():
5         return RUNNING
6     elif skill.navigation_succeeded(goal):
7         return SUCCESS
8     else:
9         return FAILURE
```

Algorithm 3: Pseudo-code of the MoveTo state in the FSM.

```
1 def execute(skill, goal, reference, recharge_state):
      # if low batteries trigger transition
        to recharge state
2     if not skill.battery_low():
3         return recharge_state
      # send the goal only the first time
4     if not goal_sent:
5         skill.move(goal, reference)
6     if not skill.navigation_done():
7         return RUNNING
      # if goal reached trigger transition
        to next state
8     elif skill.navigation_succeeded(goal):
9         return SUCCESS
10    else:
11        return FAILURE
```

To this extent, we run the experiment in the WASP Research Arena for Robotics (WARA Robotics[6], Fig. 17) where we use an ABB Mobile YuMi Research Platform (shown in Fig. 18). The robot skills are implemented in the ROS2 framework. The Nav2 library is used for navigation, ArUco marker detection is used for the perception, and for manipulation proprietary ROS2 drivers are used to send pose targets to the robot controller that computes and executes a feasible trajectory and handles the opening and closing of the grippers.

To better understand the details of the implementation, we report the pseudo code of a behavior in the BT and a state in the FSM responsible to move the robot base to an input goal in Algorithms 2 and 3 respectively. Note that, according to the py_trees API for BTs, the initialise function is called only when the behavior is ticked the first time, then the update function is called for following ticks. In case of the FSM instead, the execute function is called cyclically until the execution transitions to another state. It can be appreciated how we designed a single API for all the skills to then wrap them in a behavior or state class for BT and FSM respectively. By looking at the proposed implementation, a behavior in a BT does not require any information about other behaviors. It is the BT structure through its control nodes
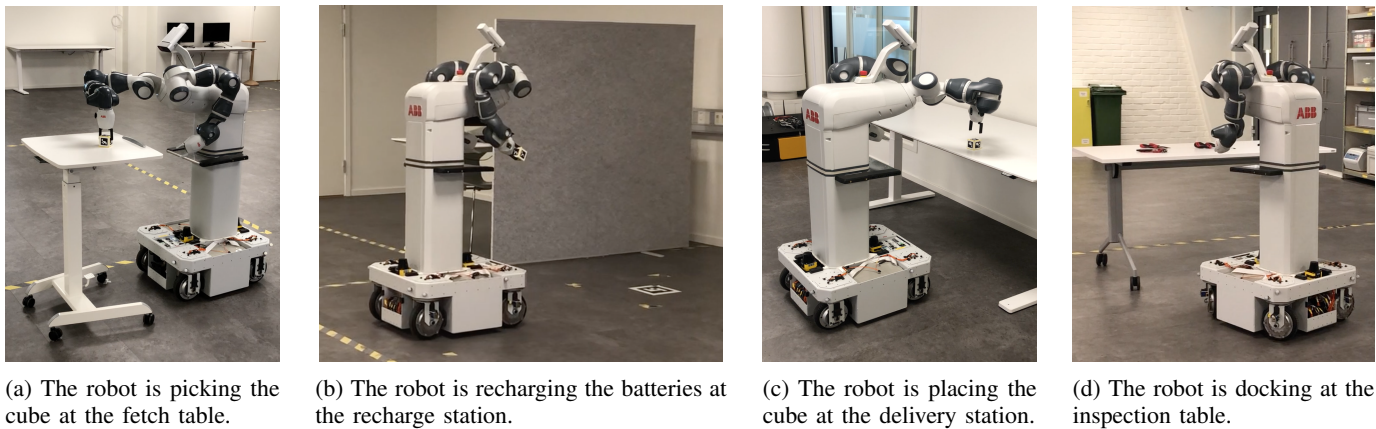
---

[6]https://wasp-sweden.org/industrial-cooperation/research-arenas/wara-robotics/

(a) The robot is picking the cube at the fetch table.

(b) The robot is recharging the batteries at the recharge station.

(c) The robot is placing the cube at the delivery station.

(d) The robot is docking at the inspection table.

Fig. 18: **Experiment 5: Real Case Scenario.** For this experiment the ABB Mobile YuMi Research Platform is used. The Experiment is carried out in the environment of Fig.17.

that switches the execution to a child or another, for instance to the recharge behavior in case the batteries are low. In the case of the FSM instead, it is necessary to include a status check inside the state execution to trigger a transition to the recharge state. Moreover, this type of check has to be done for every other state in the FSM thus making it harder to maintain if its complexity increases. To add a state to the FSM, it is required to provide a the transitions mapping between the return statuses and other states in the FSM. In the case of Algorithm 3, *SUCCESS* triggers a transition to the next state, while *FAILURE* triggers a transition to the SELECTOR state.

As for Experiment 3, the robot has to fetch an item to then deliver it and finally dock. In Fig. 17 we show all these steps. The robot navigates to the *'fetch station'* to pick up the target object. Then, it attempts to navigate to the *'delivery station'* to place it but during the execution the batteries run low. The robot then interrupts current execution to prioritize the recharging. Once the batteries are loaded again, it resumes the task to first deliver the target object and finally dock at the *'inspection table'*. Note that the depleting of the batteries is simulated and recharging them is instantaneous. This choice does not impact the argumentation for this experiment. As in the case of the other experiments, the behavior of the robot does not change if it is controlled by a BT or an FSM. Some excerpts of the task execution are reported in Fig.18, while the full task is shown in the accompanying video, together with the task switching mechanism of BTs and FSMs.

VI. CONCLUSION

In this paper we compared Behavior Trees (BTs) and Finite State Machines (FSMs) in terms of the properties of modularity, reactivity, and readability with a set of examples drawn from robotic applications. We chose common metrics to quantify the differences between the two policy representations, derived upper bounds for the comparison, and highlighted the advantages of using BTs as an alternative to FSMs.

The primary aim of this paper is to compare FSMs and BTs. However, the comparison methodology could also be applied to other graph-based policy representations, such as sequential task plans, Petri Nets, or decision trees [15], [40].

It is important to note that certain metrics, like Graph Edit Distance (GED), are not applicable to non-graph-based representations, while the metrics related to the computational complexity of edit operations can be generalized to other types of representations.

Modularity is achieved when all single components of a software share the same common structure (e.g., the same type and number of inputs and outputs). Modularity is an important feature because it allows to reuse pieces of software with minor modifications. In a BT, a modular policy representation, single nodes as well as subtrees receive a tick signal as input and return a status signal. FSMs are not modular because every state switching is realized with transitions that are state-specific. Therefore, in a BT, any subtree can be independently modified, tested and formally verified with verification architectures like Linear Temporal Logic (LTL). On the other end, editing a state in a FSM requires to take care of all the related transitions.

Reactivity is the ability to take the right action as a consequence of unexpected events or failed actions that might arise at run time. Reactivity allows a robot to be fault-tolerant. FSMs are not fault-tolerant by design, but can achieve fault-tolerant behaviors if for instance all the states are connected to each other. We proposed an alternative design for a fault-tolerant FSM and derived the effort required to modify the standard FSM to the alternative design as a function of the states. Then, we used this design to compare to BTs. With the proposed design FSMs behave like BTs, but BTs are inherently reactive.

In a policy representation, readability is an important feature both at run time and offline. It allows human operators to understand what the robot is doing while solving a task or to analyse the policy to identify fallacies. FSM are more intuitive because of their sequential execution flow. However, as the task complexity increases and the structure grows in size, the high number of transitions jeopardize the tracking of the execution flow. BTs on the other hand, have a higher learning curve because it is necessary to understand the functioning of the control nodes to follow the behavior switching. However, the tree structure allows users to recognise the task hierarchy.

For the experimental part, we conducted the comparison using a set of variations of the **Cleaning Up** task as defined in the RoboCup@Home benchmark, performed by a mobile manipulator. To make the comparison fair, the low level implementation of the skills is shared by both policies. The experiments show that with our proposed design of the FSM, the robot behaves similarly to when controlled by a BT. Since the behavior is similar, the differences between the two representations are purely in the design choice. We showed that the programming effort for such tasks is lower if BTs are preferred to FSMs, especially for large tasks that feature many behaviors and different levels of priority.

As a last remark, from a practical perspective, the choice between FSMs and BTs often depends on the skill level of the person implementing the task-level policy. For individuals with less experience or expertise, FSMs are generally more intuitive and easier to understand and implement compared to BTs, which require a deeper understanding of their structure and execution model. This makes FSMs particularly appealing for novices who need to develop or deploy simple policies quickly.

However, when policies need to be maintained or expanded over time, BTs become the preferable choice. As highlighted in [4], one effective approach is to combine the strengths of both models by using an overarching FSM as a higher-level interface for the user, where each state is implemented as a BT. This hybrid approach leverages the simplicity of FSMs for user interaction while benefiting from the flexibility and scalability of BTs at the implementation level.

ACKNOWLEDGMENTS

REFERENCES

[1] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of Behavior Trees in robotics and AI," *Robotics and Autonomous Systems*, vol. 154, p. 104096, Aug. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889022000513

[2] A. Sidorenko, M. Rezapour, A. Wagner, and M. Ruskowski, "Towards using behavior trees in industrial automation controllers." *Procedia CIRP*, vol. 130, pp. 1234–1243, 2024, 57th CIRP Conference on Manufacturing Systems 2024 (CMS 2024). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827124013908

[3] J. F. Rascón, P. Reverté, X. Ruiz, M. S. Moura, D. Serrano, and C. Rizzo, "Leveraging behavior trees for hybrid autonomous navigation in seasonal agricultural environments," in *2024 7th Iberian Robotics Conference (ROBOT)*, 2024, pp. 1–7.

[4] M. Hallen, M. Iovino, S. Sander-Tavallaey, and C. Smith, "Behavior trees in industrial applications: A case study in underground explosive charging," in *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, 2024, pp. 156–162.

[5] J. Wang, A. Laurenzi, and N. Tsagarakis, "Autonomous behavior planning for humanoid loco-manipulation through grounded language model," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024, pp. 10 856–10 863.

[6] S. Gugliermo, D. C. Domínguez, M. Iannotta, T. Stoyanov, and E. Schaffernicht, "Evaluating behavior trees," *Robotics and Autonomous Systems*, p. 104714, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889024000976

[7] M. Iovino, J. Förster, P. Falco, J. J. Chung, R. Siegwart, and C. Smith, "On the programming effort required to generate Behavior Trees and Finite State Machines for robotic applications," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, May 2023.

[8] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI : An Introduction*. CRC Press, Jul. 2018, kTH. [Online]. Available: https://www.taylorfrancis.com/books/9780429950902

[9] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski, "Behavior trees in action: a study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, pp. 196–209.

[10] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 6793–6800.

[11] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 564–571.

[12] R. Balogh and D. Obdržálek, "Using Finite State Machines in Introductory Robotics," in *Robotics in Education*, W. Lepuschitz, M. Merdan, G. Koppensteiner, R. Balogh, and D. Obdržálek, Eds. Cham: Springer International Publishing, 2019, vol. 829, pp. 85–91, series Title: Advances in Intelligent Systems and Computing. [Online]. Available: http://link.springer.com/10.1007/978-3-319-97085-1_9

[13] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968. [Online]. Available: https://doi.org/10.1145/362929.362947

[14] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[15] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, Apr. 2017, kTH.

[16] O. Biggar, M. Zamani, and I. Shames, "An Expressiveness Hierarchy of Behavior Trees and Related Architectures," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5397–5404, Jul. 2021, conference Name: IEEE Robotics and Automation Letters.

[17] A. Klöckner, "Behavior Trees for UAV Mission Management," *INFORMATIK 2013: Informatik angepasst an Mensch, Organisation und Umwelt*, vol. P-220, pp. 57–68, 2013.

[18] O. Biggar, M. Zamani, and I. Shames, "On modularity in reactive control architectures, with an application to formal verification," *ACM Trans. Cyber-Phys. Syst.*, vol. 6, no. 2, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3511606

[19] M. Olsson, "Behavior Trees for decision-making in autonomous driving," p. 54, 2016.

[20] M. Colledanchise, R. N. Parasuraman, and P. Ogren, "Learning of Behavior Trees for Autonomous Agents," *IEEE Transactions on Games*, pp. 1–1, 2018, kTH.

[21] D. Wuthier, F. Rovida, M. Fumagalli, and V. Krüger, "Productive multitasking for industrial robots," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 12 654–12 661.

[22] J. M. Zutell, D. C. Conner, and P. Schillinger, "Flexible Behavior Trees: In search of the mythical HFSMBTH for Collaborative Autonomy in Robotics," *arXiv:2203.05389 [cs]*, Mar. 2022.

[23] D. Cáceres Domínguez, M. Iannotta, J. A. Stork, E. Schaffernicht, and T. Stoyanov, "A Stack-of-Tasks Approach Combined with Behavior Trees: A New Framework for Robot Control," *IEEE Robotics and Automation Letters*, pp. 1–8, 2022.

[24] J. Bohren and S. Cousins, "The SMACH High-Level Executive [ROS News]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, Dec. 2010, conference Name: IEEE Robotics & Automation Magazine.

[25] M. Iovino, J. Styrud, P. Falco, and C. Smith, "Learning Behavior Trees with Genetic Programming in Unpredictable Environments," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, May 2021, pp. 4591–4597, iSSN: 2577-087X.

[26] K. French, S. Wu, T. Pan, Z. Zhou, and O. C. Jenkins, "Learning behavior trees from demonstration," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 7791–7797.

[27] O. Gustavsson, M. Iovino, J. Styrud, and C. Smith, "Combining Context Awareness and Planning to Learn Behavior Trees from Demonstration,"

This article has been accepted for publication in IEEE Transactions on Automation Science and Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TASE.2025.3610090

20

in *2022 IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*, Aug. 2022.

[28] M. Colledanchise, D. Almeida, and P. Ögren, "Towards Blended Reactive Planning and Acting using Behavior Trees," in *2019 International Conference on Robotics and Automation (ICRA)*, May 2019, pp. 8839–8845, iSSN: 2577-087X.

[29] M. Mayr, F. Ahmad, K. Chatzilygeroudis, L. Nardi, and V. Krueger, "Combining planning, reasoning and reinforcement learning to solve industrial robot tasks," *arXiv preprint arXiv:2212.03570*, 2022.

[30] J. Styrud, M. Iovino, M. Norrlöf, M. Björkman, and C. Smith, "Combining Planning and Learning of Behavior Trees for Robotic Assembly," in *2022 IEEE International Conference on Robotics and Automation (ICRA)*, May 2022.

[31] M. Iovino, J. Styrud, P. Falco, and C. Smith, "A framework for learning behavior trees in collaborative robotic applications," *2023 IEEE International Conference on Automation Science and Engineering (CASE)*, 2023.

[32] P. Ögren, "Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6073–6080, Oct. 2020, conference Name: IEEE Robotics and Automation Letters.

[33] F. Fusaro, E. Lamon, E. D. Momi, and A. Ajoudani, "A Human-Aware Method to Plan Complex Cooperative and Autonomous Tasks using Behavior Trees," in *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*, Jul. 2021, pp. 522–529, iSSN: 2164-0580.

[34] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler, "Parameterizing Behavior Trees," in *Motion in Games*, ser. Lecture Notes in Computer Science, J. M. Allbeck and P. Faloutsos, Eds. Springer Berlin Heidelberg, 2011, pp. 144–155.

[35] Z. Abu-Aisheh, R. Raveaux, J.-y. Ramel, and P. Martineau, "An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems," in *4th International Conference on Pattern Recognition Applications and Methods 2015*, Lisbon, Portugal, Jan. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01168816

[36] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," p. 5, 2008.

[37] T. Wisspeintner, T. v. d. Zant, L. Iocchi, and S. Schiffer, "RoboCup@Home: Scientific Competition and Benchmarking for Domestic Service Robots," *Interaction Studies*, vol. 10, no. 3, pp. 392–426, Jan. 2009, publisher: John Benjamins. [Online]. Available: https://www.jbe-platform.com/content/journals/10.1075/is.10.3.06wis

[38] M. Breyer, J. J. Chung, L. Ott, R. Siegwart, and J. Nieto, "Volumetric grasping network: Real-time 6 dof grasp detection in clutter," in *Conference on Robot Learning*. PMLR, 2021, pp. 1602–1611.

[39] M. Iovino, "Learning behavior trees for collaborative robotics," Ph.D. dissertation, KTH Royal Institute of Technology, 2023.

[40] M. Colledanchise and P. Ögren, "How behavior trees generalize the teleo-reactive paradigm and and-or-trees," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 424–429.

Pietro Falco received the Ph.D. in Electronic Engineering in 2012 and until February 2015 he held a Post-Doc position at University of Campania Luigi Vanvitelli. From December 2010 to July 2011 he was a Visiting Scholar at Karlsruher Institute of Technology, Karlsruhe. In 2011, he co-founded the company Aeromechs s.r.l. In March 2015, he received a TUM Foundation Fellowship and in February 2016 he received a Marie Curie Individual Fellowship for experienced researchers. In January 2018, he joined ABB Corporate Research in Sweden as a tenured senior research scientist and project manager. In May 2023, he moved to University of Padova as an assistant professor. He is an IEEE senior member and serves as AE for IEEE RAL, and for IEEE Transactions on Medical Robotics and Bionics. His research interests include machine learning for robotics, observation of human manipulation, robotic mobile manipulation, and Bayesian sensor fusion.

Jen Jen Chung (Member, IEEE) received the B.E. degree in aeronautical (space) engineering from the University of Sydney, Camperdown, NSW, Australia, in 2010, and the Ph.D. degree in information-based exploration–exploitation strategies for autonomous soaring platforms from the Australian Centre for Field Robotics, University of Sydney in 2014. She is currently an Associate Professor of Mechatronics with the School of Electrical Engineering and Computer Science, University of Queensland, Brisbane, QLD, Australia. From 2018 to 2022, she was a Senior Researcher with the Autonomous Systems Lab, ETH Zürich, Zürich, Switzerland. From 2014 to 2017, she was a Postdoctoral Scholar with Oregon State University, Corvallis, OR, USA, where she researched multiagent learning methods. Her current research interests include perception, planning, and learning for robotic mobile manipulation, algorithms for robot navigation through human crowds, informative path planning, and adaptive sampling.

Roland Siegwart is professor for autonomous mobile robots at ETH Zurich, founding co-director of the Wyss Zurich and board member of multiple high-tech companies. He was professor a EPFL and Vice President of ETH Zurich. He is among the most cited scientist in robots world-wide, co-founder of more than half a dozen spin-off companies and a strong promoter of innovation and entrepreneurship in Switzerland.

Matteo Iovino received in 2023 a Ph.D. in Computer Science at the Royal Institute of Technology (KTH) in Stockholm (SE). At the time of writing, he was employed by ABB Corporate Research (SE), where he was managing WARA (WASP Research Arena) Robotics. His research interests include control and planning for collaborative mobile manipulation in dynamic industrial environments, with a focus on using learning techniques to synthesize control policies.

Julian Förster received his bachelor's degree in mechanical engineering at ETH Zurich (2015) as well as a master's degree with distinction in mechanical engineering with a specialization in robotics and control systems at ETH Zurich (2018). He received his Ph.D. in 2024 from the Autonomous Systems Lab at ETH Zürich, with research in planning for robotics, scene understanding and machine learning.

Christian Smith received his M.Sc. in Engineering Physics in 2005, and Ph.D. in Computer Science in 2009, both at the Royal Institute of Technology in Stockholm (KTH). He was a post-doctoral researcher at Advanced Telecommunications Research International (ATR) in Kyoto, Japan from 2010 to 2011. He is currently an Associate Professor in Computer Science and the Deputy Head of the Division of Robotics, Perception, and Learning at KTH. He is the secretary of the IEEE Robotics and Automation Society Swedish Chapter, and serves as associate editor for IEEE Robotics and Automation Letters, and several conference editorial boards. His research interests include control and modelling for manipulation and grasping in human-centric environments, such as medical facilities and small-scale manufacturing plants, as well as physical human-robot interaction.