Towards a Unified Behavior Tree Framework for Robot Control

Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren

Abstract—Behavior Trees (BTs) are a plan representation and execution tool which has recently started to draw the attention of the robotics community. The available literature lack consistency and mathematical rigor required for control applications. This paper presents a unified formalism for BTs which elevates them to the same standard as other well known plan execution tools. We approach the problem in two steps: first, reviewing the most popular BT literature exposing the aforementioned problems; second, describing our unified BT framework along with equivalence notions between BTs and Controlled Hybrid Dynamical Systems (CHDSs). This paper improves on the existing state of the art as it describes BTs in a more accurate and compact way. We also present an experimental evaluation using a NAO robot and our BT library.

I. INTRODUCTION

Behavior Trees (BTs) are commonly used in simple scenarios [1]–[12] where there is no need to have a solid mathematical foundation encompassing continuous-time dynamics. However, such requirement arises in order to use BTs on more complex applications, e.g., real robots, control systems. This indicates the need to formalize BTs in a more general mathematical framework that is both accurate and compact.

Intuitively, we measure accuracy to be inversely proportional to the degree of misinterpretation a certain statement can be subjected to. Likewise, we measure compactness to be inversely proportional to the amount of definitions required to fully specify an idea. Using these two qualitative indicators we created a framework that surpasses the state of the art.

BTs follow a simpler but rather arbitrary algorithmic definition when compared to Controlled Hybrid Dynamical Systems (CHDSs) [15]–[17]. This causes us to believe that the increased simplicity of BTs comes at the expense of decreased expressiveness. This paper analyzes the correctness of this assertion in detail by providing equivalence notions.

There exist ad-hoc engineering solutions to circumvent the intrinsic limitations of BTs regarding two aspects: nodes are memory-less [5] (do not store the last running node), and BTs execute independently from one another [2] (cooperative tasks are not possible). This paper formalizes and motivates solutions to both problems in an accurate and compact way.

Much uncertainty exist yet regarding the potential of BTs as a suitable representation to replace CHDSs [13]. We address the problem in two complementary ways: from CHDSs to BTs, and vice-versa. This provides important insight about which tasks are representable, under what constraints, and which are the advantages / disadvantages of each paradigm.

The authors are with the Computer Vision and Active Perception Lab., Centre for Autonomous Systems, School of Computer Science and Communication, Royal Institute of Technology (KTH), SE-100 44 Stockholm, Sweden. e-mail: {almc|miccol|ccs|petter}@kth.se

To demonstrate the usability of our work, we implemented an open source BT library for the Robot Operating System (ROS) [19], which allowed us to test the concepts described in this paper on real robotic platforms. We briefly analyze the implementation limitations regarding the different ways of resolving subset intersection (prevalence and hysteresis).

The main contributions of this paper are listed below:

1) A more accurate and compact BT framework.
2) Introduction of the Action and Condition subsets.
3) Formalization and motivation of two node extensions.
4) Equivalence notions between BTs and CHDSs.

This paper is linked to a video¹, which shows the experiments mentioned in Section IX, and the source code of the library is publicly available on www.github.com/almc.

The paper is structured as follows: in Section II we review related work, in Section III we formalize BTs and introduce the subsets, in Sections IV and V we present the Node* and the Decorator* extensions, in Section VI we present a formal definition of CHDSs, in Section VII we study the equivalence between BTs and CHDSs, in Section VIII we present the software structure of our implementation, in Section IX we describe the experimental framework, and in Section X we present the conclusions and future work.

¹YouTube video name: Behavior Trees - NAO Grasping [ROS / C++].
II. RELATED WORK

Most of the current BT research efforts are focused towards finding new efficient ways to implement Artificial Intelligence (AI) for entertainment systems, e.g. specifying Non-Player Characters (NPCs) in video-games. This situation often yields papers that are lengthy qualitative discussions lacking mathematical support and accurate definitions. Ad-hoc solutions, created for particular video-games, are not generalizable to be applied in other research fields. In fact, the very nature of ad-hoc heuristics yields sub-optimal solutions which makes it impossible to do incremental work.

Fortunately, not all papers suffer from these problems but they do differ in several aspects ($§_{11}$ – $§_{11}$). The criteria, by which these papers were evaluated, is specified in Table I, whereas the comparison itself is presented in Table II. We refrain from expanding on the contributions of each paper since our primary goal is to point out the differences as a mean to justify the need for an united BT framework.

### TABLE I. Criteria used to compare BT publications ($§_{11}$ – $§_{11}$).

<table>
<thead>
<tr>
<th>Criterion</th>
<th>$§_{1}$</th>
<th>$§_{2}$</th>
<th>$§_{3}$</th>
<th>$§_{4}$</th>
<th>$§_{5}$</th>
<th>$§_{6}$</th>
<th>$§_{7}$</th>
<th>$§_{8}$</th>
<th>$§_{9}$</th>
<th>$§_{10}$</th>
<th>$§_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>planning integration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>non-blocking actions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>modularity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dynamic tree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>virtual world</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>multi-agent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>global variables</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BT library</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>infinite execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>multiple parents to node</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table II demonstrates that many papers disagree in crucial aspects such as: action definition $§_{2}$, modularity $§_{3}$, execution policy $§_{10}$, and tree structure $§_{11}$. Our framework, which is mainly based on [13], manages to combine every aspect considered in Table I with the exception of planning integration. To the best of our knowledge, this is the first paper to achieve such integration, thereby bringing BTs closer to robots.

III. BEHAVIOR TREES

This section gives a formal definition of BTs following the guidelines of [3], [8], [13]. A BT is defined as a directed acyclic graph $G(V,E)$ with $|V|$ nodes and $|E|$ edges. We call the outgoing node of a connected pair the parent, and the incoming node the child. We call the child-less nodes leaves, and the unique parent-less node Root. Each node in a BT, with the exception of the Root, is one of six possible types: four non-leaf (control-flow) node types (Selector, Sequence, Parallel and Decorator), and two leaf (execution) node types (Action and Condition). These are summarized in Table III.

As a BT is executed, the Root node generates a tick — enabling signal transferred between nodes — at a fixed frequency $f_{	ext{tick}}$. The tick is then propagated through the branches according to the algorithm defined for each node type. When the tick signal reaches a leaf node, it performs the corresponding Action or Condition. Actions can alter the system configuration, returning one of three possible state values: Success, Failure, or Running. Conditions cannot alter the system configuration, returning one of two possible state values: Success, or Failure. This returned state is then propagated back and forth through the tree, possibly triggering other leaf nodes with their own return states until finally one of these states reaches the Root. The BT waits before sending the new tick to maintain $f_{	ext{tick}}$ constant.

Unlike traditional graph theory trees [14], any node in the BT (except the Root and its only child) can have multiple parents [3]. The reason is to allow sub-trees to be reused without having to copy them. In practice, doing this results in BTs which are functional but cumbersome, for this reason we explicitly advocate for the following workaround: nodes having multiple parents are prohibited, the re-usability of sub-trees is not to be done at the level of control-flow nodes (where the gain is minimal), preferably, it is to be done at the level of execution nodes (where the gain is substantial).

#### A. Node Types

The node types behave according to Algorithms 1–11. The statement $\text{Tick}(\text{child}(i))$ triggers the algorithm that corresponds to its child type, expecting a returned state value. The program execution starts and finishes on Algorithm 4.

### TABLE III. The seven node types of a BT. Abbreviations: Ch $\equiv$ children, $S \equiv$ succeeded, $F \equiv$ failed, $R \equiv$ running. The numbers $S,F \in \mathbb{N}$ are Parallel node parameters, $N \equiv \# \text{ children}$. The subscript $n$ is the Action or Condition index. Lastly, $(S_n,F_n,R_n) \subseteq X_n, X_n(t) \in X_n, U_n(t) \in U_n$ are the subsets, state space and control signals (see Section III-B).

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Symbol</th>
<th>Succeeds if</th>
<th>Fails if</th>
<th>Runs if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>$\emptyset$</td>
<td>tree $S$</td>
<td>tree $F$</td>
<td>tree $R$</td>
</tr>
<tr>
<td>Selector</td>
<td>$?$</td>
<td>$1 \text{ Ch } S$</td>
<td>$N \text{ Ch } F$</td>
<td>$1 \text{ Ch } R$</td>
</tr>
<tr>
<td>Sequence</td>
<td>$\rightarrow$</td>
<td>$N \text{ Ch } S$</td>
<td>$1 \text{ Ch } F$</td>
<td>$1 \text{ Ch } R$</td>
</tr>
<tr>
<td>Parallel</td>
<td>$\Rightarrow$</td>
<td>$\geq S \text{ Ch } S$</td>
<td>$\geq F \text{ Ch } F$</td>
<td>otherwise</td>
</tr>
<tr>
<td>Decorator</td>
<td>$\diamond$</td>
<td>varies</td>
<td>varies</td>
<td>varies</td>
</tr>
<tr>
<td>Action $n$</td>
<td>$\circ$</td>
<td>$X_n(t) \in S_n$</td>
<td>$X_n(t) \in F_n$</td>
<td>$X_n(t) \in R_n$</td>
</tr>
<tr>
<td>Condition $n$</td>
<td>$\triangledown$</td>
<td>$X_n(t) \in S_n$</td>
<td>$X_n(t) \in F_n$</td>
<td>never</td>
</tr>
</tbody>
</table>

$^2$The nodes which are not ticked are set to a special node state: NotTicked.
Selector. When a Selector node is enabled, it ticks its children sequentially for as long as they return Failure, and until one of them returns Running or Success. If the Selector node does not find a running or succeeding child, it returns Failure, otherwise it returns Running or Success depending on the state of its first non-failing child.

Sequence. When a Sequence node is enabled, it ticks its children sequentially for as long as they return Success, and until one of them returns Running or Failure. If the Sequence node does not find a running or failing child, it returns Success, otherwise it returns Running or Failure depending on the state of its first non-failing child.

Parallel. When a Parallel node is enabled, it ticks all its children sequentially. If the number of succeeding children is \( \geq S \), it returns Success. If the number of failing children is \( \geq F \), it returns Failure. Otherwise, it returns Running.

Decorator. When a Decorator node is enabled, it checks a condition on its internal variables, based on which it could tick or not its only child. It applies decorator-dependent functions to determine the return state of the node.

Action. When an Action node is enabled, it determines the state value to be returned by checking if its current state space configuration \( X_n(t) \) belongs to the Success \( S_n \), Failure \( F_n \) or Running \( R_n \) subsets. On the third case, it also performs a discrete atomic control step \( \gamma_n : X_n \rightarrow U_n \).

Condition. When a Condition node is enabled, it behaves like the Action, without the Running subset and control step.

Algorithm 1: Selector

```plaintext
1 for i ← 1 to N do
2 state ← Tick(child(i))
3 if state = Running then
4 return Running
5 if state = Success then
6 return Success
7 end
8 end
9 return Failure
```

Algorithm 2: Sequence

```plaintext
1 for i ← 1 to N do
2 state ← Tick(child(i))
3 if state = Running then
4 return Running
5 if state = Success then
6 return Success
7 end
8 end
9 return Success
```

Algorithm 3: Parallel

```plaintext
1 for i ← 1 to N do
2 state, ← Tick(child(i))
3 end
4 if nSucc(state) ≥ S then
5 return Success
6 if nFail(state) ≥ F then
7 return Failure
8 else
9 return Running
10 end
```

Algorithm 4: Main Loop

```plaintext
1 initialize(agent)
2 BT.parse(agent)
3 while (active = true) do
4 state ← Tick(Root)
5 sleep(1/fs)
6 BT.delete(agent)
7 end
8 return 0
```

Algorithm 5: Action

```plaintext
1 if X_n(t) ∈ S_n then
2 return Success
3 if X_n(t) ∈ F_n then
4 return Failure
5 if X_n(t) ∈ R_n then
6 U_n(t) ← \( \gamma_n(X_n(t)) \)
7 return Running
8 end
```

Algorithm 6: Condition

```plaintext
1 if X_n(t) ∈ S_n then
2 return Success
3 if X_n(t) ∈ F_n then
4 return Failure
5 end
```

Algorithm 7: Root

```plaintext
1 return Tick(child(0))
```

B. Action Subsets

Action nodes rely on three subsets: \{\( S_n \), \( F_n \), \( R_n \)\}, used in Algorithm 5. Condition nodes rely on two subsets: \{\( S_n \), \( F_n \)\}, used in Algorithm 6. We focus on the Action since the Condition is simpler. These subsets partition\(^3\) the Action’s state space \( X_n \), where \( X_n(t) \) take its values, such that:

\[
S_n \cup F_n \cup R_n \supseteq X_n
\]

As an example consider the modular\(^4\) driving BT shown in Fig. 2, with the corresponding Action subsets portrayed in Fig. 3. These Actions, scheduled by the Sequence node, try to maintain a proper distance from the next vehicle using control algorithms \( \gamma_n(X_n(t)) = U_n(t) \). To illustrate the use of subsets we show two possible event sequences from the perspective of each Action: the first corresponds to the upper branch (s\( _1^e \rightarrow s_2^e \rightarrow s_3^e \rightarrow s_4^e \rightarrow s_5^e \)), and the second corresponds to the lower branch (f\( _1^c \rightarrow f_3^c \rightarrow f_4^c \)). Qualitatively, both scenarios start with two control steps executed by Normal Driver (r\( _1^f \), r\( _2^f \)). In the first case, the execution is handed over to Cruise Driver (r\( _1^c \), r\( _2^c \) reaching Success on the fifth control step (s\( _5^e \)). In the second case, the execution is handed over to Emergency Driver to take care of an unexpected situation during two control steps (r\( _1^e \), r\( _2^e \)).

\(^3\)We could be more restrictive: \( S_n \cap F_n = S_n \cap R_n = F_n \cap R_n = \emptyset \). However, intersecting subsets allow the use of hysteresis on \( X_n \), e.g. the threshold for switching from \( R_n \) to \( S_n \) is not necessarily the same as the threshold for switching from \( S_n \) to \( R_n \), which would happen otherwise.

\(^4\)Modular means that it can be treated as a stand-alone Action by BTs of higher hierarchy seamlessly. Modularity is enforced through the logic of the BT (control-flow nodes and subsets). The Root (not shown in Fig. 2) is removed before appending a sub-BT to another BT for obvious reasons.
IV. The Node* Extension

The BT node algorithms presented so far are insufficient to represent plans where it is necessary to “remember” if an Action / Condition / sub-tree has already succeeded or failed.

A. Node* Motivation

Let us consider a Sequence node with two fully-actuated Actions whose subsets, \( X_1 \) and \( X_2 \), are represented in Fig. 4. Under these assumptions, if \( X_1 \cap X_2 \neq \emptyset \) there is at least one variable controlled by both Actions. Depending on the definition of \( \{S_1,F_1,R_1\} \) and \( \{S_2,F_2,R_2\} \), this could yield unsatisfiable BTs, e.g. where Action 2 will “undo” a goal previously achieved by Action 1, causing an endless loop.

![Fig. 4. Subsets of two Actions demonstrating how cycles appear.](image)

In general, the traditional BT algorithms have the following limitations: in a Sequence node, for an arbitrary j-indexed child Action \( A_j \) to be ticked at time \( t_k \) it needs to happen that \( \{X_1(t_k) \in S_1 \land \ldots \land X_{j-1}(t_k) \in S_{j-1}\} \in \mathbb{R}_+ \). Similarly, in a Selector node, for an arbitrary j-indexed child Action \( A_j \) to be ticked at time \( t_k \) it needs to happen that \( \{X_1(t_k) \in F_1 \land \ldots \land X_{j-1}(t_k) \in F_{j-1}\} \in \mathbb{R}_+ \).

This could be desirable in some cases where we need to guarantee that a certain property holds over a set of Actions, but in other situations it is necessary to “remember” which nodes have already returned Success or Failure, in order to not tick (check) them again on the next iteration. The Parallel node does not have this problem, hence no Parallel* exists.

B. Node* Extended Algorithms

In practice, rather than remembering which children have returned Success or Failure, we use a variable that points to the child that has most recently returned Running. This variable is reset every time the Selector or Sequence returns a terminal state (Success or Failure). See Algorithms 8, 9.

Algorithm 8: Selector*

```
for i ← run-index to N do
    state ← Tick(child(i))
    if state = Running then
        run-index ← i
    return Running
if state = Success then
    return Success
run-index ← 1
return Success
```

Algorithm 9: Sequence*

```
for i ← run-index to N do
    state ← Tick(child(i))
    if state = Running then
        run-index ← i
    return Running
if state = Failure then
    return Failure
run-index ← 1
return Failure
```

V. The Decorator* Extension

The BT node algorithms presented so far are insufficient to represent plans where two or more agents must undertake a common task jointly by synchronizing parts of their BTs.

A. Decorator* Motivation

To control multiple agents using BTs, we have two choices: to have one big BT containing the Actions of all the agents, or to have separated BTs running the Actions of each agent independently from one another. The first solution has the advantage that all the agents can be synchronized inside the same structure, however it is clear that for big groups of agents it turns unmanageable. The second solution has the advantage that BTs are much smaller, easier to understand and expand, however it is not obvious how these independent trees can be synchronized to achieve cooperative behaviors.

We discard the first solution as it is unfeasible for our purposes, focusing on the second scenario which can be dealt with using two approaches. First, making new control-flow nodes with the synchronization capability. Second, making a special Decorator node with the sole purpose of providing its sub-tree with the synchronization capability. We favor the second because it allows the multi-agent features to be kept aside from the execution logic, this permits non-cooperative behaviors to become cooperative by merely placing them under the synchronization Decorator defined below.

B. Decorator* Extended Algorithm

When the Decorator* is enabled, it broadcasts the agent’s name (determined contextually) to the other Decorator* nodes of the same cooperative task, indicating them that it is ready to engage as soon as there are enough agents \( n_{req} \) to trigger the sub-tree. In most cases, the tick is received by this node when the other agents are busy performing higher priority tasks of their BTs \( (n_{now} < n_{req}) \), in these cases the Decorator* will return without ticking its sub-tree. Eventually, enough agents will be available to engage in the cooperation \( (n_{now} \geq n_{req}) \), at this point the barrier imposed to the ticks by the synchronization Decorator* will be temporarily removed allowing the sub-tree to be executed. Naturally, this requires a software infrastructure, like ROS, capable of handling message passing, and a mechanism that allows each Decorator* to keep track of which and how many agents have broadcasted their messages. Time stamps are used to ensure that such messages were broadcasted recently enough to be valid. See Algorithms 10, 11.

Algorithm 10: Decorator*

```
1 Broadcast(agent, ID)
2 if n_{now} \geq n_{req} then
3   state ← Tick(child)
4   state ← state_{t}
5 else
6   state ← NotTicked
7 Update(agents, n_{now})
8 return state
```

Algorithm 11: Decorator

```
1 PreFunc(vars)
2 if Condition(vars) then
3   state ← Tick(child)
4 else
5   state ← ϕ_{2}(vars, state_{t})
6 end
7 PostFunc(vars, state)
8 return state
```

---

5 The function \( \mu : \mathcal{U} \to X \) is bijective (one to one correspondence \( \forall u, x \)).

6 The barrier will block again if \( n_{now} < n_{req} \) at any point in time.
VI. CONTROLLED HYBRID DYNAMIC SYSTEMS

Following the definitions of [15]–[17]: a CHDS, shown in Fig. 5, is an indexed collection of Controlled Dynamical Systems (CDS) and a mechanism for switching between them whenever the hybrid state satisfies certain conditions and the control dictates so. More formally, a CHDS $H$ is defined as follows $H = (Q, X_q, U_q, A_q, E_q, T_q, C_q, D_q, S_q)_{q \in Q}$, with:

- $Q$: discrete state space $Q = \{ q_i | i \in \{1, \ldots, |Q| \} \}$
- $X_q$: continuous state space $X_q = \{ x_{j,q} | j \in \{1, \ldots, |X_{chds}| \} \}$
- $U_q$: control signal space $U_q = \{ u_{k,q} | k \in \{1, \ldots, |U_q| \} \}$
- $A_q$: edge label set $A_q = \{ a_{q,j} | q \in N_q \}$
- $N_q$: directed neighborhood of $q (q \in N_q \Rightarrow q \in N_q)$
- $E_q$: edge set, each edge is $E_q = \{ q, q', a_{q,j}, G_{q,j}, J_{q,j} | q \in N_q, q', q \}$ initial and final discrete states ($\{ q, q' \}$ $A_q$ edge label connecting $\{ q, q' \}$ with $a_{q,j} \in A_q$
- $G_{q,j}$: edge guard enabled if $X_q(t) \in G_{q,j} \subseteq X_q$
- $J_{q,j}$: state jump sets $X_q(t) \rightarrow X_q(t) \in J_{q,j} \subseteq X_q \times X_q$
- $T_q$: location invariant of $X_q(t) \in T_q \forall t \in \mathbb{R}, \forall q \in Q$
- $C_q$: control algorithms $U_q(t) = G_q(X_q(t))$
- $D_q$: system dynamics $X_q(t) = \Delta_q(X_q(t), U_q(t))$
- $S_q$: hybrid state $\{ Q(t), X_q(t), A_q(t), U_q(t) \}$
- $S_0$: initial state $\{ Q(0), X_0(0), A_0(0), U_0(0) \}$

Consider a continuous trajectory $(q, \delta_q, X_q(t), U_q(t))$ associated to the discrete state $q$ with a non-negative time $\delta_q$ (duration of the continuous trajectory), a piecewise continuous function $U_q(t) : [0, \delta_q] \rightarrow U_q$, and a continuous piecewise differentiable function $X_q(t) : [0, \delta_q] \rightarrow X_q$, such that $X_q(t) \in T_q \forall t \in (0, \delta_q)$ and $\Delta_q(X_q(t), U_q(t))$ for the piece of discontinuity.

The trajectory (solution / run) of a CHDS is a (possibly infinite) sequence of continuous trajectories chained together: $(q \delta, q', X_{q'}(t), U_{q'}(t)) \rightarrow (q, q', \Delta_q(t), U_q(t))$ such that at the event times where transitions occur $t_0, t_1, \ldots$, defined as: $t_0 = \delta_{q_0}, t_1 = \delta_{q_0} + \delta_{q_1}, t_2 = \delta_{q_0} + \delta_{q_1} + \delta_{q_2}, \ldots$, the following inclusions hold for the discrete transitions $a_j$: $X_{q_j}(t_j) \in G_{q_j} \cup \bar{G}_{q_j}$ and $(X_{q_{j+1}}(t_{j+1})) \in J_{q_{j+1}}$ for all $j = 0, 1, \ldots, \infty$. Where $q'_j$ is the $j$-th state $q$ taking place, to which one associates the symbol $a_j$, representing the value of the discrete "signal" at the $j$-th discrete transition.

VII. EQUIVALENCE

This section represents the representation capabilities of BTs with respect to CHDS. We show explicitly that every CHDS can be represented as a BT but not vice versa.

A. From CHDSs To BTs

To prove that any CHDS has an equivalent BT it suffices to show that the trajectories both systems produce, when confronted with the same environment (for all possible environments and initial conditions), are identical. Naturally, a CHDS has continuous dynamics which are impossible to mimic using a discrete-time structure like a BT. For this reason, we define a continuous-time BT as a regular BT which has: infinite tick frequency, zero tick propagation time, and zero execution time for Actions and Conditions.

Additionally, assuming: the BT initializes $Q(t)$ to the initial state of the CHDS, and the CHDS uses a sequential prioritized jump policy $a_j$. It is straightforward to check that for an infinitesimal time window $dt$, the BT shown in Fig. 6, in continuous-time, produces the same control as the CHDS shown in Fig. 5. More complex jump policies are not covered by this proof but could possibly be reproduced depending on the case. Since this holds true for any infinitesimal time window, it follows that the trajectories are also identical.

Lastly, consider a CHDS where the continuous dynamics are discretized using a finite sampling frequency $f_{CHDS}$, and a BT which satisfies all the properties of continuous-time BTs except for the tick frequency $f_{tick}$ which in this case is finite. Under the assumption that $f_{CHDS} = f_{tick}$, and following a similar reasoning, it follows that the discretized trajectories are equal because they are sampled synchronously.

This BT mimics the corresponding CHDS as follows: there are $|Q|$ branches departing from the first Selector which account for the checks that need to be performed in order to determine which discrete state is currently being executed. The transitions to other discrete states are controlled by the $|N_q|$ branches departing from the second Selector, these include both the CHDS guards $G$ and the jumps $J$. The CHDS-BT equivalence is not unique, to prove it we notice that any re-ordering of the BT nodes in Fig. 6, that preserves the underlying logic, is still an equivalent BT.

Fig. 5. Generic CHDS $H$, showing two connected states $q$ and $q'$.

Fig. 6. Equivalent BT to the generic CHDS presented in Fig. 5.
B. From BTs To CHDSs

Any BT consisting only of a Root, Selectors, Sequences, Actions, and Conditions has an equivalent CHDS representation. The inclusion of Decorators, Selector*, Sequence*, and Parallel nodes precludes the translation because CHDSs do not support such features. To show the equivalence for our constrained scenario we write the CHDS version of a generic Selector, Sequence, and Root. Then, we use the fact that BTs are built using these basic blocks, embedding them one inside the other, to justify making the following assertion: finding an equivalent CHDS representation for a set of BT node types is a sufficient condition to guarantee that any BT built using only those types can be represented using a CHDS.

The functionality that is missing from CHDSs to mimic the Selector* and Sequence* nodes is being able to rewire (on run-time) the edges between discrete states, e.g. the start points to a different state. The functionality missing from CHDSs to mimic the Parallel is being able to run multiple discrete states simultaneously, which would involve a multi-valued initial state and possibly a more complex jump policy. Decorators are to be dealt with on a case-by-case basis.

Inside the CHDS discrete states, which correspond to BT Actions / sub-trees (A1 : A_N), we place the corresponding controllers γ_n presented in Algorithm 5. From this point it is straightforward to see that under the input / output mindset, the CHDS of a BT node primitive can be embedded as an Action A_n in a CHDS of higher hierarchy. Performing this procedure recursively turns out to be equivalent to translating the BT to a CHDS starting from the leaves, and following this embedding procedure recursively until the Root is reached.

VIII. ROS IMPLEMENTATION

We propose a ROS implementation of the BT framework, the behavior-trees library, designed abiding by the Google C++ Style Guide [18] with the following compromises:

- Compatibility with existing ROS libraries, making it fast and easy to incorporate into new robotic platforms.
- Simplicity of the code, allowing other programmers to understand, expand, maintain, and reuse the code.
- Efficiency of processing power, providing the complete functionality of BTs, using a low amount of resources.

The implementation consists of three main parts: the Client, where the BT is held, the Server, where the Action and Condition algorithms are held, and the Communication, where the connection between the first two is made. In the following paragraphs we describe how these parts, represented in Fig. 10, work together.

![Diagram showing the client-server communication.](image)

A. Client

This part is a ROS node that contains the BT control-flow nodes, and the clients of the Action and Condition nodes, i.e. the BT logic that does not change between applications. It is composed of three parts: Parser, Interface, and Execution.

a) Parser: Reads the BT specification from a file and generates the node objects dynamically. The nodes are linked with each other according to the tree structure. The parent only stores a pointer to his first child, he refers to his other children using pointers between adjacent brothers.

b) Interface: Is a lightweight OpenGL based interface that displays the BT, and the state of each node in real-time. It allows the user to navigate through the tree, and override the node states for simulation or debugging purposes.

c) Execution: Generates a new tick at the Root of the BT with a fixed frequency. It updates the state of each node by propagating the tick through the branches using the algorithms explained in Section III-A. When the tick reaches a leaf node, an actionlib message is sent to signal the server.

8We ignored Conditions because they are simpler versions of Actions.
B. Server

Every leaf node of the BT is linked to a corresponding ROS server node. These contain the functionality of the application’s actuation and sensing algorithms. Each server is composed of the three parts: Goal, Publisher, and Execution.

a) Goal: Receives the actionlib messages, that are sent from the client side, each time a tick reaches a leaf node. It updates internal variables, such as the “elapsed time since last message was received”, to determine if the control algorithms should be: started, resumed, or stopped.

b) Publisher: Contains functions to send state updates to the client side, so that the BT node states are kept synchronized with the actual state of the controllers. The user must define the succeeding $S_n$, failing $F_n$, and running $R_n$ subsets for each control algorithm to be implemented.

c) Execution: Runs the main loop of the controller on a separate thread to allow asynchronous actionlib message reception. It periodically checks the “elapsed time since last message was received” in order to destroy the thread if no tick has reached the client node for a certain amount of time.

C. Communication

The client and server parts are connected using actionlib, the most widely used ROS library. It functions under a non-supervised goal achieving scheme, where the client sends a goal to the server and waits for it to either succeed or fail.

Clearly, actionlib does not work with the same paradigm as the BTs, but it provides a valuable framework of client-server communication in ROS. We took advantage of this to provide our behavior-trees library with the set of callbacks that allows it to schedule in time the different nodes of a BT. Among these functions\(^9\) we highlight the following:

- **DoneCB**: called upon goal completion.
- **ActiveCB**: called upon goal acceptance.
- **FeedbackCB**: called upon feedback publishing.
- **GoalCB**: called upon new goal reception.
- **PreemptCB**: called upon goal preemption.
- **ExecuteCB**: called periodically if the node is active.

D. Implementation Limitations

There are two main limitations: the first cannot be avoided due to the nature of BTs, the second has a small workaround:

1) BTs operate by calling a function from inside another function in a recursive manner following the Algorithms 1–11. Computationally, this could produce stack overflow for huge trees, even for implementations like ours that separate the control algorithms from the execution logic using clients and servers.

2) For each tick that is sent to traverse the BT, a large number of checks has to be performed over the state spaces of the Actions in the tree. Our implementation overcomes this problem by performing both calculations asynchronously, thereby preferring to get a delayed state update than blocking the tick flow.

\(^9\)The callbacks DoneCB, ActiveCB, and FeedbackCB belong to the client. The callbacks GoalCB, PreemptCB, and ExecuteCB belong to the server.

IX. Experiments

To show the potential of BTs and the usability of our library, we implemented in ROS a grasping task using a NAO humanoid robot from Aldebaran Robotics, see Fig. 11.

A. The Experimental Setup

We define a scenario where the robot stands up, walks towards a table, attempts 3 different grasps on an object until one of them succeeds or all three fail. If a grasp succeeds: the robot informs the user, releases the object, turns 180 degrees, and returns to the starting position. If all grasps fail, the robot informs the user, but does not return to the starting position. In both cases, whether the grasp was successful or not, the robot sits down and disables its motors.

To improve the safety of the robot behavior, we include fallback handling for motor temperature and falls. This means that at any point in the execution of the program, if the robot detects either of these conditions, it automatically disables the current node and enables the proper one to handle the situation. For instance, if the robot detects a high motor temperature, it sits down and disables the motors in order to prevent overheating. Additionally, if the robot detects a fall, it attempts to stand up before continuing.

B. Behavior Tree Representation

The BT of this task is shown in Fig. 12, and features two characteristics that, in general, make BTs powerful tools:

- **Flexibility**: It is easy to extend the behavior by adding or removing nodes without modifying the structure of the tree. For instance, to include detection and proper handling of low battery levels, it suffices to add the dashed Condition in Fig. 12. In contrast, adding or removing a node from a CHDS, could potentially involve wiring $2N + 1$ arcs ($N/N$ arcs going to l departing from the node + 1 self-loop).

- **Modularity**: It is possible to encapsulate behaviors as sub-trees, in order to append them to a tree with a higher hierarchy. To do this, the sub-tree to be appended needs to be modular, which informally means it never returns Success or Failure until it has actually finished executing its goal.

![Fig. 12. BT representation of the grasping task implemented on the NAO.](image-url)
X. Conclusions

We presented an algorithmic BT framework which is substantially more accurate and compact than previous descriptions. We provided equivalence notions between BTs and CHDSs which gave us insight about the representation capabilities of each framework: using BTs we lose the ability to be in certain state, but we achieve modularity, i.e. the ability to treat BTs as hierarchical structures which can be chained together. A BT is modular if its control-flow nodes and Action / Condition node subsets are laid out in such a way that the goal represented succeeds or fails in finite time, and its Root receives Running for all intermediate states.

We introduced the Action and Condition subsets, which allowed us to formalize and motivate two extensions to the basic BT functionality, thereby avoiding the use of ad-hoc solutions. The experimental setup and theoretical analysis allowed us to conclude that BTs can replace CHDSs without losing mathematical accuracy, descriptive power or human readability. Moreover, BTs have a larger set of representable plans because nodes such as the Parallel, Selector*, or Sequence* do not have a corresponding CHDS representation.

While not explicitly demonstrated here, we believe that the flexibility and modularity of BTs, make them a good candidate for representing Middle Layer plans that are created and maintained automatically by high-level AI algorithms. In future work, we plan to address this by studying how BTs can be generated parting from a given Linear-Temporal-Logic (LTL) formula. Lastly, we plan to further analyze racing conditions that could emerge as a result of an inappropriate use of the Parallel node at the level of control signals.

Acknowledgment

This work has been supported by the Swedish Research Council (VR) and the European Union Project RECONFIG, www.reconfig.eu (FP7-ICT-2011-9, Project Number: 600825), the authors gratefully acknowledge the support.

We also thank: Isak Tjernberg for sharing his experience with the NAO platform, Xavi Gratal for providing his programming expertise, Francisco Viña for sharing his ROS knowledge. Lastly we thank: Alessandro Pierpont, Johannes Stork, and Jana Tumova for their useful critical reviews.

References