Towards a Unified Behavior Trees Framework for Robot Control

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

***Abstract*— This paper presents a unified framework for Behavior Trees (BTs), a plan representation and execution tool. The available literature lacks the consistency and mathematical rigor required for robotic and control applications. Therefore, we approach this problem in two steps: first, reviewing the most popular BT literature exposing the aforementioned issues; 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, while providing insight about their actual representation capabilities. Lastly, we demonstrate the applicability of our framework to real systems scheduling open-loop actions in a grasping mission that involves a NAO robot and our BT library.**

I. Introduction

Behavior Trees (BTs) are plan representation tools commonly used in scenarios where there is no need to have a 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 mathematical framework that is both *accurate* and *compact*.

Intuitively, we measure *accuracy* to be inversely proportional to the degree of misinterpretation that 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 indicators we created a framework that surpasses the state of the art.

In pursuit of the *accuracy*, we formulated the *Action* and *Condition* subsets which remove the ambiguities that could arise when referring to the node *Success*, *Running*, or *Failure*. It is relying on these subsets that we could motivate the node extensions, and compare BTs with other plan representations.

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.

After [13], there is still uncertainty regarding the potential of BTs as a suitable representation to replace Controlled Hybrid Dynamical Systems (CHDSs) [15]–[17]. 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 what 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

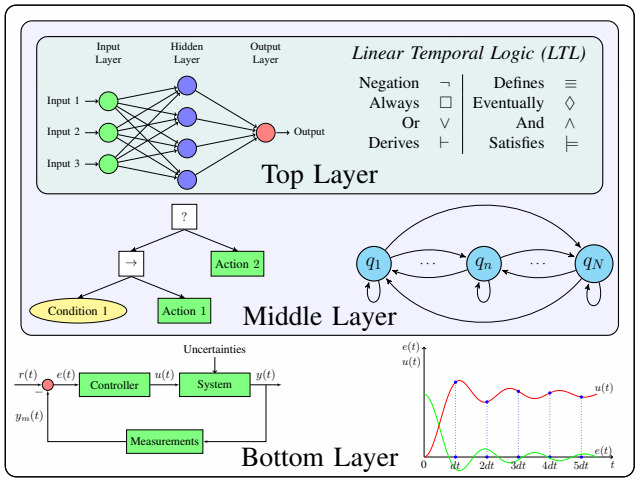


Fig. 1. The *Top*, *Middle*, and *Bottom Layers* of robot architectures. The division is arbitrary: there is no strict boundary between layers.

As represented in Fig. 1, the BTs and CHDSs belong to the *Middle Layer* which provides a mechanism for switching between low-level controllers. The *Top Layer* automatically generates and updates plans, whereas the *Bottom Layer* handles low-level controllers that interact with the hardware.

To demonstrate the usability of our work, we implemented an open source BT library for the Robot Operating System (ROS) [18], 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.

The execution of the task described in Section IX is presented in a video[1], 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*$^\sim$ 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.

[1] YouTube 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 BT frameworks that are game-oriented; even though they contain interesting features, they are not generalizable to be used in other research fields, e.g. robotics.

Fortunately, not all papers suffer from these problems but they do differ in several aspects ($\S_1$ – $\S_{11}$). 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, even though they do contain noteworthy contributions, since our primary goal is to point out the differences between them as a mean to justify the need for an unified BT framework.

TABLE I. Criteria used to compare BT publications ($\S_1$ – $\S_{11}$).

$\S_1$	*planning integration*: whether there exists a mechanism for automated BT generation and maintenance. This corresponds to the *Top Layer*.
$\S_2$	*non-blocking actions*: whether actions stop the flow of the BT until they finish executing. This affects the ability of the BT to react to changes.
$\S_3$	*modularity*: whether BTs are built in such a way that they can be chained together (embedded one inside the other) and still function properly.
$\S_4$	*dynamic tree*: whether BTs can be modified during run-time. This is required for the *Top Layer* implementation, but does not imply it exists.
$\S_5$	*virtual world*: if it was implemented in a simulated environment (game). This considers cases where it has access to the game-state or to an interface.
$\S_6$	*real world*: if it was implemented in the real world (robot). This considers cases where the outcome of the BT manifests outside a computer simulation.
$\S_7$	*multi-agent*: whether the proposed BT structure can handle multiple agents simultaneously. This considers cases involving at least two agents.
$\S_8$	*global variables*: whether the implementation requires a common set of variables (*blackboard*) shared between actions or uses *parameters* instead.
$\S_9$	*BT library*: whether the implementation supports a database of behaviors that can be polled to build BTs. This requires modularity enforced.
$\S_{10}$	*infinite execution*: whether the BTs are meant to finish executing at some point, or they are supposed to run forever. This affects the modularity.
$\S_{11}$	*multiple parents to node*: whether the strict definition of trees is enforced, or a relaxed version (where a node can belong to several parents) is preferred.

TABLE II. Comparison of BT publications. [↻]: reference to this paper.

Ref	$\S_1$	$\S_2$	$\S_3$	$\S_4$	$\S_5$	$\S_6$	$\S_7$	$\S_8$	$\S_9$	$\S_{10}$	$\S_{11}$
[1]	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗	✗
[2]	✗	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗
[3]	✗	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓
[4]	✗	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗
[5]	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✗
[6]	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✗
[7]	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗
[8]	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗
[9]	✗	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗
[10]	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓	✗
[11]	✓	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗
[12]	✗	✓	✗	✓	✓	✗	✓	✓	✗	✓	✗
[13]	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗
[↻]	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table II demonstrates that many papers disagree in crucial aspects such as: $\S_2$, $\S_3$, $\S_{10}$, and $\S_{11}$. Our framework, which builds upon [13], [19] and [20], manages to combine every aspect considered in Table I except *planning integration*. To the best of our knowledge, this is the first paper to achieve such integration. The reader should be aware that BTs are, however, not the only alternative for *Middle Layer* representation, see [21]–[25]. Lastly, we point out that BTs have also been formalized through CSP semantics in [26].

III. Behavior Trees

This section gives a formal description of BTs following the guidelines of [3], [8], [13]. A BT is defined as a directed acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}|$ nodes and $|\mathcal{E}|$ 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.

Unlike traditional graph theory trees [14], any node in the BT (except the *Root* and its only child) can have multiple parents [3]. This allows *sub-trees* to be reused without having to copy them but decreases readability; 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, preferably, it is to be done at the level of *execution* nodes.

The *Root* periodically, with frequency f_{tick}, generates an enabling signal called *tick*, which is propagated through the branches according to the algorithm defined for each node. When the *tick* reaches a *leaf* node, it executes one cycle of the *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 nodes which are not *ticked* are set to a special node state: *NotTicked*. The BT then waits before sending the new *tick* to maintain f_{tick} constant. We remark that *in the implementation* the *tick* frequency f_{tick} is completely unrelated to the controller's frequency f_{control}; they work asynchronously as explained in Section VIII-B.

A. Node Types

The node types behave according to Algorithms 1–11, where the statement `Tick(child(i))` : triggers the algorithm that corresponds to its child node type. The execution begins and ends on Algorithm 4, the symbols $\mathsf{S}, \mathsf{F}, \mathsf{R} \subseteq \mathcal{X}$, $X(t) \in \mathcal{X}$, $U(t) \in \mathcal{U}$ are the *Success/Failure/Running subsets*, *state space*, and *control signals* respectively. For a detailed real-life example using these variables refer to Section III-B.

TABLE III. The seven node types of a BT. `Ch` ≡ *children*, `S` ≡ *succeeded*, `F` ≡ *failed*, `R` ≡ *running*. $N \equiv$ *# children*, $S, F \in \mathbb{N}$ are node parameters.

Node Type	Symb.	Succeeds if	Fails if	Runs if
Root	∅	tree `S`	tree `F`	tree `R`
Selector	?	1 `Ch S`	N `Ch F`	1 `Ch R`
Sequence	→	N `Ch S`	1 `Ch F`	1 `Ch R`
Parallel	⇉	$\geq S$ `Ch S`	$\geq F$ `Ch F`	otherwise
Decorator	◈	varies	varies	varies
Action n	□	$X_n(t) \in \mathsf{S}_n$	$X_n(t) \in \mathsf{F}_n$	$X_n(t) \in \mathsf{R}_n$
Condition n	○	$X_n(t) \in \mathsf{S}_n$	$X_n(t) \in \mathsf{F}_n$	never

Selector. When a *Selector* node is enabled, it *ticks* its children sequentially as long as they continue to 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 as long as they continue to 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-succeeding 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 *functions* ϕ_1 or ϕ_2 to determine the return *state*, see Algorithm 11 for the template.

Action. When an *Action* node, indexed n, 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 *control step* $\gamma_n : \mathcal{X}_n \to \mathcal{U}_n$.

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

Algorithm 1: Selector

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

Algorithm 2: Sequence

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

Algorithm 3: Parallel

```
for i ← 1 to N do
    state_i ← Tick(child(i))
end
if nSucc(state) ≥ S then
    return Success
if nFail(state) ≥ F then
    return Failure
else
    return Running
end
```

Algorithm 4: Main Loop

```
initialize(agent)
BT.parse(agent)
while (active = true) do
    state ← Tick(Root)
    sleep(1/f_tick)
end
BT.delete(agent)
return 0
```

Algorithm 5: Action

```
if X_n(t) ∈ S_n then
    return Success
if X_n(t) ∈ F_n then
    return Failure
if X_n(t) ∈ R_n then
    U_n(t) ← γ_n(X_n(t))
    return Running
end
```

Algorithm 6: Condition

```
if X_n(t) ∈ S_n then
    return Success
if X_n(t) ∈ F_n then
    return Failure
end
```

Algorithm 7: Root

```
return Tick(child(0))
```

B. Action Subsets

Action nodes rely on three subsets: $\{\mathsf{S}_n, \mathsf{F}_n, \mathsf{R}_n\}$, used in Algorithm 5. *Condition* nodes rely on two subsets: $\{\mathsf{S}_n, \mathsf{F}_n\}$, used in Algorithm 6. We focus on the *Action* since the *Condition* is simpler. These subsets partition the *Action's* state space $\mathcal{X}_n$, where $X_n(t)$ take its values, such that the following properties hold for *Action* n and *Condition* n respectively: $\mathsf{S}_n \cup \mathsf{F}_n \cup \mathsf{R}_n \supseteq \mathcal{X}_n$, and $\mathsf{S}_n \cup \mathsf{F}_n \supseteq \mathcal{X}_n$. We could be more restrictive, i.e. $\mathsf{S}_n \cap \mathsf{F}_n = \mathsf{S}_n \cap \mathsf{R}_n = \mathsf{F}_n \cap \mathsf{R}_n = \varnothing$. However, intersecting subsets allow the use of *hysteresis*, 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.

As an example consider the *modular* driving BT shown in Fig. 2, with the corresponding *Action* subsets portrayed in Fig. 3. A BT being *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* node placement and subsets definitions). The *Root*, purposefully omitted in Fig. 2, is always removed before composing two BTs to avoid having multiple *tick* sources.

These three *Actions*, scheduled by the *Sequence* node, try to maintain a proper distance from the next vehicle using control algorithms $\gamma_n(X_n(t)) = U_n(t)$. Defining the notation $\text{return\_status}^{\text{action\_label}}_{\text{time\_step}}$, we illustrate the use of subsets showing two possible event sequences from the perspective of each driver *Action* $(e|n|c)$: these correspond to the upper/lower branches, i.e. $(s^e_1 \to s^e_5 | r^n_1 \to s^n_5 | f^c_1 \to s^c_5)$, and $(s^e_1 \to r^e_4 | r^n_1 \to f^n_4 | f^c_1 \to f^c_4)$ respectively. Qualitatively, both scenarios start with two control steps (r^n_1, r^n_2) executed by *Normal Driver*. On the upper branch, the execution is handed over to *Cruise Driver* (r^c_3, r^c_4) reaching *Success* on the fifth control step (s^c_5). On the lower branch, the execution is handed over to *Emergency Driver* to take care of an unexpected situation during two control steps (r^e_3, r^e_4).

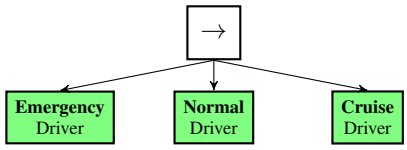


Fig. 2. A generic autonomous driving behavior, represented using a BT.

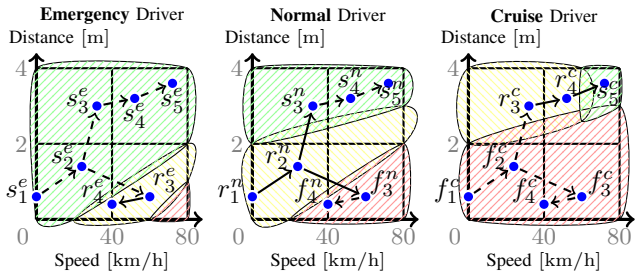


Fig. 3. Succeeding (S, green), Failing (F, red), and Running (R, yellow) *subsets*. The *dots* (blue) represent $X(t) \in \mathcal{X}$ for $t = 1/f_{\text{tick}}, \ldots, 5/f_{\text{tick}}$. The *solid arrows* / *dashed arrows* represent transitions caused by the *Action's* controller / system dynamics or other controllers respectively.

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*[2] *Actions* whose subsets, $\mathcal{X}_1$ and $\mathcal{X}_2$, are represented in Fig. 4. Under these assumptions, if $\mathcal{X}_1 \cap \mathcal{X}_2 \neq \varnothing$ there is at least one variable controlled by both *Actions*. Depending on the subset definitions, this could yield unsatisfiable BTs, e.g. *Action 2* executes $r_3^{A_2}$; decreasing $X[1]$ so that $X(t) \notin \mathsf{S}_1$ anymore, *Action 1* then executes $r_4^{A_1}$, $r_2^{A_1}$; causing an endless cycle.

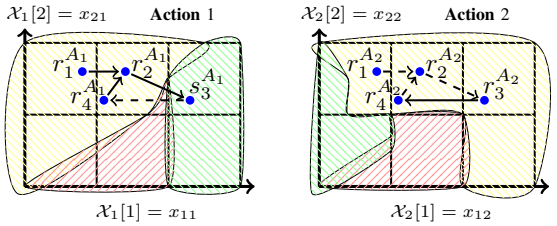


Fig. 4. Subsets of two *Actions* demonstrating how cycles appear. For simplicity we let $\mathcal{X}_1 = \mathcal{X}_2$, but normally it is enough that $\mathcal{X}_1 \cap \mathcal{X}_2 \neq \varnothing$.

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* $\{X_1(t_k) \in \mathsf{S}_1 \wedge \ldots \wedge X_{j-1}(t_k) \in \mathsf{S}_{j-1}\}_{t_k \in \mathbb{R}_+}$. 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* $\{X_1(t_k) \in \mathsf{F}_1 \wedge \ldots \wedge X_{j-1}(t_k) \in \mathsf{F}_{j-1}\}_{t_k \in \mathbb{R}_+}$.

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

Rather than tracking and storing 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
        run-index ← 1
        return Success
    end
end
run-index ← 1
return Failure
```

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
        run-index ← 1
        return Failure
    end
end
run-index ← 1
return Success
```

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

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 ($n_{\text{now}} < n_{\text{req}}$), in these cases the *Decorator~* will return without *ticking* its sub-tree. Eventually, enough agents will be available to engage in the cooperation ($n_{\text{now}} \geq n_{\text{req}}$), at this point the barrier imposed to the *ticks* by the synchronization *Decorator~* will be temporarily[3] 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~

```
Broadcast(agent, ID)
if n_now ≥ n_req then
    state_t ← Tick(child)
    state ← state_t
else
    state ← NotTicked
end
Update(agents, n_now)
return state
```

Algorithm 11: Decorator

```
PreFunc(vars)
if Condition(vars) then
    state_t ← Tick(child)
    state ← ϕ1(vars, state_t)
else
    state ← ϕ2(vars)
end
PostFunc(vars, state)
return state
```

[3]The barrier will block again if $n_{\text{now}} < n_{\text{req}}$ at any point in time.

VI. Controlled Hybrid Dynamical 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 $\mathcal{H}$ is defined[4] as follows $\mathcal{H} = (\mathcal{Q}, \mathcal{X}_q, \mathcal{U}_q, \mathcal{A}_q, \mathcal{E}_q, \mathcal{I}_q, \mathcal{C}_q, \mathcal{D}_q, \mathcal{S}_0)_{q\in\mathcal{Q}}$, with:

- $\mathcal{Q}$ *discrete state space* $\mathcal{Q} = \{q_i \,|\, i \in \{1, \dots, |\mathcal{Q}|\}\}$
- $\mathcal{X}_q$ *continuous state space* $\mathcal{X}_q = \{x_{jq} \,|\, j \in \{1, \dots, |\mathcal{X}_q|\}\}$
- $\mathcal{U}_q$ *control signal space* $\mathcal{U}_q = \{u_{kq} \,|\, k \in \{1, \dots, |\mathcal{U}_q|\}\}$
- $\mathcal{A}_q$ *edge label set* $\mathcal{A}_q = \{a_{q\bar{q}} \,|\, \bar{q} \in \mathcal{N}_q\}$
  - $\mathcal{N}_q$ *directed neighborhood* of q ($\bar{q} \in \mathcal{N}_q \not\Rightarrow q \in \mathcal{N}_{\bar{q}}$)
- $\mathcal{E}_q$ *edge set*, each *edge* is $\mathcal{E}_{q,\bar{q}} = (q, \bar{q}, a_{q\bar{q}}, \mathcal{G}_{q\bar{q}}, \mathcal{J}_{q\bar{q}})_{\bar{q}\in\mathcal{N}_q}$
  - $q, \bar{q}$ *initial* and *final* discrete states $(q, \bar{q}) \in (\mathcal{Q}, \mathcal{N}_q)$
  - $a_{q\bar{q}}$ *edge label* connecting $(q, \bar{q})$ with $a_{q\bar{q}} \in \mathcal{A}_q$
  - $\mathcal{G}_{q\bar{q}}$ *edge guard* enabled if $X_q(t) \in \mathcal{G}_{q\bar{q}} \subseteq \mathcal{X}_q$
  - $\mathcal{J}_{q\bar{q}}$ *state jump* sets $X_q(t) \to X_{\bar{q}}(t) \in \mathcal{J}_{q\bar{q}} \subseteq \mathcal{X}_q \times \mathcal{X}_{\bar{q}}$
- $\mathcal{I}_q$ *location invariant* $X_q(t) \in \mathcal{I}_q \quad \forall t \in \mathbb{R}, \forall q \in \mathcal{Q}$
- $\mathcal{C}_q$ *control algorithms* $U_q(t) = \Gamma_q(X_q(t))$
- $\mathcal{D}_q$ *system dynamics* $\dot{X}_q(t) = \Delta_q(X_q(t), U_q(t))$
- $\mathcal{S}_t$ *hybrid state* $\{Q(t), X_q(t), A_q(t), U_q(t)\}$
- $\mathcal{S}_0$ *initial state* $\{Q(0), X_q(0), A_q(0), U_q(0)\}$

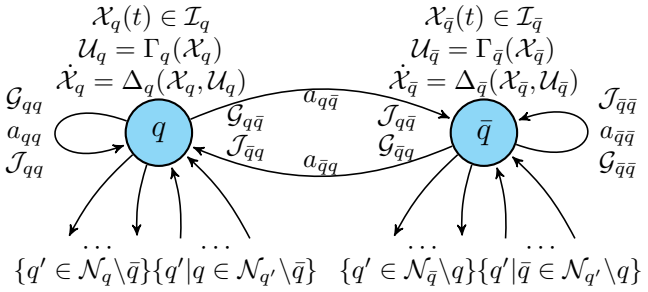


Fig. 5. Generic CHDS $\mathcal{H}$, showing two connected states q and $\bar{q}$.

Consider a *continuous trajectory* $(q, \delta_q, X_q(t), U_q(t))$ associated to the discrete state q with a non-negative time δ_q (duration of the continuous trajectory), a piecewise continuous function $U_q(t) : [0, \delta_q] \to \mathcal{U}_q$, and a continuous piecewise differentiable function $X_q(t) : [0, \delta_q] \to \mathcal{X}_q$, such that $X_q(t) \in \mathcal{I}_q \quad \forall t \in (0, \delta_q)$ and $\Delta(X_q(t), U_q(t)) = \dot{X}_q(t) \quad \forall t \in (0, \delta_q)$ except for the points of discontinuity.

The *trajectory* (*solution / run*) of a CHDS is a (possibly infinite) sequence of *continuous trajectories* chained together: $(q^0, \delta_{q^0}, X_{q^0}(t), U_{q^0}(t)) \overset{a_0}{\to} (q^1, \delta_{q^1}, X_{q^1}(t), U_{q^1}(t)) \overset{a_1}{\to} \cdots$, such that at the *event times* where transitions occur $t_0, t_1, \dots$, defined as: $t_0 = \delta_{q^0}, t_1 = \delta_{q^0} + \delta_{q^1}, t_2 = \delta_{q^0} + \delta_{q^1} + \delta_{q^2}, \dots$, the following inclusions hold for the discrete transitions $\overset{a_j}{\to}$: $X_{q^j}(t_j) \in \mathcal{G}_{q^j q^{j+1}}$ and $(X_{q^j}(t_j), X_{q^{j+1}}(t_j)) \in \mathcal{J}_{q^j q^{j+1}}$ for all $j = 0, 1, \dots, \infty$. Where q^j is the j-th state q taking place, to which one associates the symbol $a_j \equiv a_{q^j q^{j+1}}$, that represents the *jump policy* signal at the j-th state transition.

[4] $\mathcal{I}_q : \mathcal{Q} \to \mathcal{I}_q \subseteq \mathcal{X}_q$, $\mathcal{G}_{q\bar{q}} : \mathcal{Q} \to \mathcal{G}_{q\bar{q}} \subseteq \mathcal{X}_q$, $\mathcal{J}_{q\bar{q}} : \mathcal{Q} \times \mathcal{X}_q \to \mathcal{Q} \times \mathcal{X}_{\bar{q}}$, $\Delta_q : \mathcal{Q} \times \mathcal{X}_q \times \mathcal{U}_q \to \dot{\mathcal{X}}_q$, $\Gamma_q : \mathcal{Q} \times \mathcal{X}_q \to \mathcal{U}_q$, $Q(t) \in \mathcal{Q}$, $X_q(t) \in \mathcal{X}_q$, $A_q(t) \in \mathcal{A}_q$, $U_q(t) \in \mathcal{U}_q$, $\mathcal{S}_t \in \mathcal{Q} \times \mathcal{X}_q \times \mathcal{A}_q \times \mathcal{U}_q$, $\mathcal{S}_0 = \mathcal{S}(t_0)$.

VII. Equivalence

We show that every CHDS using a specific *jump policy* can be represented with a BT, and every BT composed only of certain node types can be represented with a CHDS.

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 that 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. Since this holds true for any infinitesimal time window, it follows that the *trajectories* are also identical. More complex *jump policies* are not covered but could possibly be reproduced depending on the case.

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 $f_{\text{CHDS}} = f_{\text{tick}}$, and following a similar reasoning, it follows that the *discretized trajectories* are equal because they are sampled / executed synchronously.

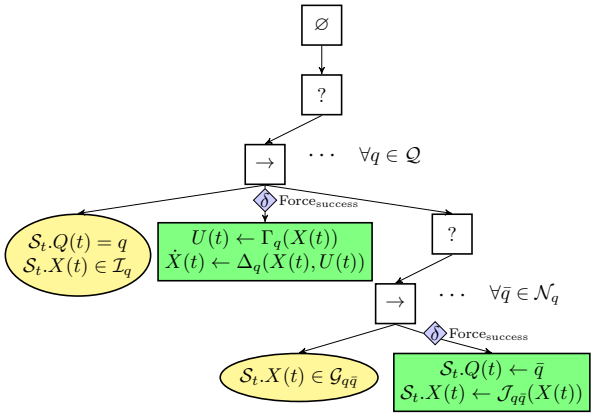


Fig. 6. Equivalent BT to the generic CHDS presented in Fig. 5.

This BT *mimics* the corresponding CHDS as follows: there are $|\mathcal{Q}|$ 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 covered by the $|\mathcal{N}_q|$ branches departing from the *second Selector*, these include both the CHDS guards $\mathcal{G}$ and the jumps $\mathcal{J}$. 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 and *jump policy*, is still a valid BT.

B. From BTs To CHDSs

The inclusion of *Decorator*, *Selector**, *Sequence**, and *Parallel* nodes precludes the translation because CHDSs do not support certain features that those nodes bring about. The functionality missing from CHDSs to mimic the *Selector** and *Sequence** nodes is being able to rewire (on run-time) the edges between discrete states; this involves dynamic edges. The functionality missing from CHDSs to mimic the *Parallel* node is being able to execute multiple discrete states simultaneously; this involves multi-valued initial states. *Decorators* are to be dealt with on a case-by-case basis.

Any BT consisting only of a *Root*, *Selectors*, *Sequences*, *Actions*, and *Conditions* has an equivalent CHDS representation. To show the equivalence we write the CHDS version of a generic *Selector* / *Sequence* (with N children), and a *Root*. Then, we use the fact that BTs are constructed using these basic building blocks (*node primitives*), embedding them one inside the other, to justify making the following assertion: finding an equivalent CHDS representation for a set of BT *node primitives* is a sufficient condition to guarantee that any BT built using only this set can be represented with a CHDS.

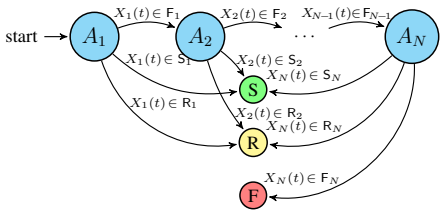


Fig. 7. *Selector* node with N *Actions* / *sub-trees* represented as a CHDS.

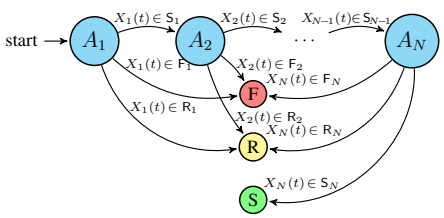


Fig. 8. *Sequence* node with N *Actions* / *sub-trees* represented as a CHDS.

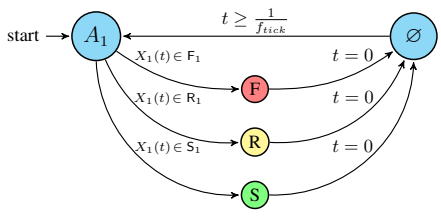


Fig. 9. *Root* node with 1 *Action* / *sub-tree* represented as a CHDS.

In these three CHDSs: Fig. 7, Fig. 8, and Fig. 9, the *start* is to be thought of as an *input* (where the *tick* comes from), and the three small colored circles labeled 'S', 'R', and 'F' are to be thought of as *outputs* (where the *tick* is returned), none of which are discrete states, they are symbolic and merely used to wire arcs when composing the CHDS of the BT.

Inside the CHDS discrete states, which correspond to BT *Actions* / *sub-trees*[5] $(A_1 : A_N)$, 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* 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 *actionlib* ROS library provides the connection between the first two. In the following paragraphs we describe how these parts, represented in Fig. 10, work together.

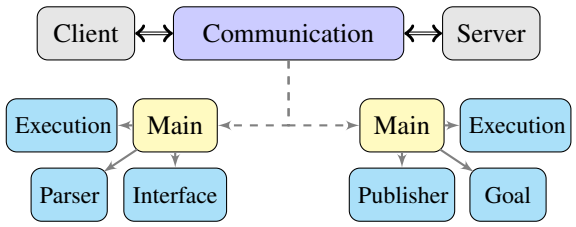


Fig. 10. Diagram showing the client-server communication.

A. Client

This part is a ROS module 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 and refers to the other children using pointers between adjacent brothers.

b) Interface: Displays the BT with the node *states* in real-time using a lightweight *OpenGL* based interface. 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*.

[5]We ignored *Conditions* because they are simpler versions of *Actions*.

B. Server

Every *leaf* node of the BT is linked to a corresponding ROS *server* module. These contain the functionality of the application's actuation and sensing algorithms running at the control frequency f_{control} referenced earlier. 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[6] 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.

[6]The callbacks *DoneCB*, *ActiveCB*, and *FeedbackCB* belong to the *client*. The callbacks *GoalCB*, *PreemptCB*, and *ExecuteCB* belong to the *server*.

IX. System Demonstration

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. 12.

A. The Mission

We define a scenario where the robot stands up, walks towards a table, and attempts three 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. 11, and features two characteristics that, in general, make BTs powerful tools:

a) 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. 11. In contrast, adding or removing a node from a CHDS, could potentially involve wiring $2N+1$ arcs (N/N arcs *going to* / *departing from* the node $+$ 1 self-loop).

b) 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*.

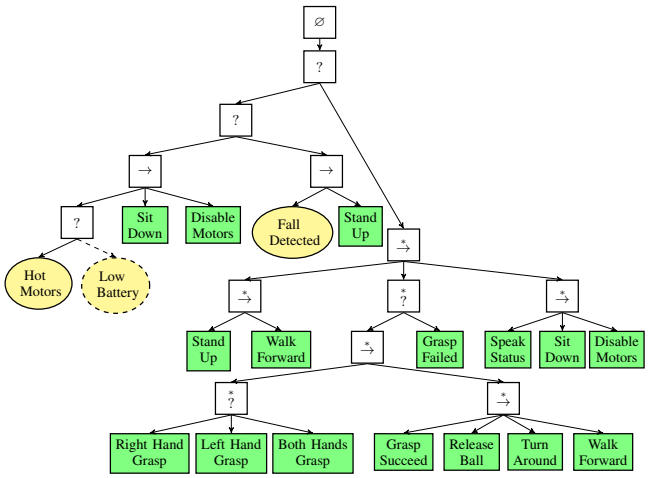
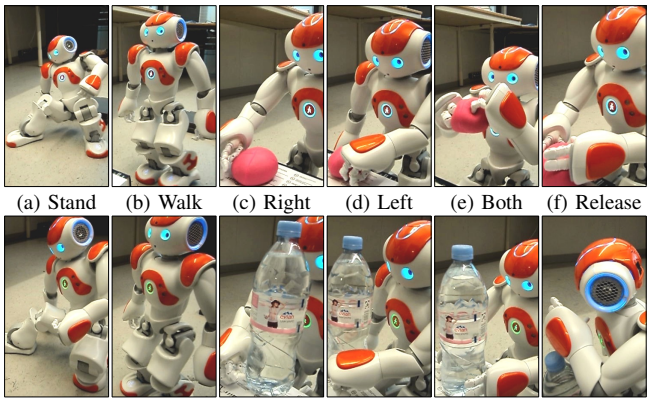


Fig. 11. BT representation of the grasping task implemented on the NAO. We entirely disregard the automatic plan generation, i.e. *Top Layer* of Fig. 1.



(a) Stand (b) Walk (c) Right (d) Left (e) Both (f) Release

(g) Stand (h) Walk (i) Right (j) Left (k) Both (l) Failed

Fig. 12. Demonstration 1: *successful ball grasp* depicted in 12(a)–12(f). Demonstration 2: *failed bottle grasp* depicted in 12(g)–12(l).

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 advantages of each framework: using BTs we lose the ability to *be* in a certain state, but we achieve *modularity*. A BT is modular if its *control-flow* nodes and the 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 robotic platform tests and theoretical analysis allowed us to conclude that BTs can replace certain CHDSs without losing 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. We acknowledge that the current demonstration merely shows open-loop action scheduling through BTs, and we plan to address this by upgrading the system to encompass closed-loop controllers. Lastly, we plan to analyze the relation between our framework parameters and its functionality.

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, and *Francisco Viña* for sharing his ROS knowledge. Lastly we thank: *Danica Kragic*, *Alessandro Pieropan*, *Johannes Stork*, *Jana Tumova*, and *Yuquan Wang* for the useful critical reviews provided for this paper.

References

[1] R. Palma, P. A. González-Calero, M. A. Gómez-Martín, and P. P. Gómez-Martín, "Extending Case-Based Planning with Behavior Trees," in *FLAIRS Conference*. AAAI Press, 2011.

[2] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," in *Applications of Evolutionary Computation*. Springer, 2010, vol. 6024, pp. 100–110.

[3] A. Johansson and P. Dell'Acqua, "Emotional Behavior Trees," in *Computational Intelligence and Games, 2012 IEEE Conference on*. IEEE, 2012, pp. 355–362.

[4] J. Tremblay, "Understanding and Evaluating Behaviour Trees," McGill University, Modelling, Simulation and Design Lab, Tech. Rep., 2012.

[5] S. Delmer, "Behavior Trees for Hierarchical RTS AI," Diploma Thesis, Southern Methodist University (SMU), 2012.

[6] G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo, and P. A. González-Calero, "Query-Enabled Behavior Trees," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, no. 4, pp. 298–308, 2009.

[7] N. Yatapanage, K. Winter, and S. Zafar, "Slicing Behavior Tree Models for Verification," in *Theoretical Computer Science*, ser. IFIP Advances in Information and Communication Technology. Springer, 2010, vol. 323, pp. 125–139.

[8] 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. Springer, 2011, vol. 7060, pp. 144–155.

[9] R. Colvin, L. Grunske, and K. Winter, "Probabilistic Timed Behavior Trees," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4591, pp. 156–175.

[10] M. Cutumisu and D. Szafron, "An Architecture for Game Behavior AI: Behavior Multi-Queues," in *AIIDE*. The AAAI Press, 2009.

[11] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution," in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6624, pp. 123–132.

[12] D. Becroft, J. Bassett, A. Mejía, C. Rich, and C. L. Sidner, "AIPaint: A Sketch-Based Behavior Tree Authoring Tool," in *AIIDE*, 2011.

[13] P. Ögren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in *AIAA Guidance, Navigation, and Control Conference*. AIAA, 2012.

[14] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Macmillan London, 1976, vol. 290.

[15] A. J. van der Schaft and J. M. Schumacher, *An Introduction to Hybrid Dynamical Systems*. Springer, 2000, vol. 251.

[16] M. S. Branicky, "Introduction to Hybrid Systems," in *Handbook of Networked and Embedded Control Systems*, ser. Control Engineering. Springer, 2005, pp. 91–116.

[17] M. Branicky, "Studies in Hybrid Systems: Modeling, Analysis, and Control," Ph.D. dissertation, Cambridge, MA, USA, 1995.

[18] Willow Garage. (2012) Robot Operating System (ROS). Groovy. [Online]. Available: http://www.ros.org/wiki/

[19] M. Colledanchise, A. Marzinotto, and P. Ögren, "Performance Analysis of Stochastic Behavior Trees," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, June 2014.

[20] J. A. D. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, M. Pivtoraiko, J.-S. Valois, and R. Zhu, "An Integrated System for Autonomous Robotics Manipulation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 2955–2962.

[21] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. 2, 1986.

[22] R. C. Arkin, *An Behavior-based Robotics*. Cambridge, MA, USA: MIT Press, 1998.

[23] J. Rosenblatt, "DAMN: A Distributed Architecture for Mobile Navigation," Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[24] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mosenlechner, W. Meeussen, and S. Holzer, "Towards Autonomous Robotic Butlers: Lessons Learned with the PR2," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 5568–5575.

[25] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. Kemp, "ROS Commander: Flexible Behavior Creation for Home Robots," in *ICRA*, 2013.

[26] R. J. Colvin and I. J. Hayes, "A semantics for Behavior Trees using CSP with specification commands," *Science of Computer Programming*, vol. 76, no. 10, pp. 891–914, 2011.