How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees

Michele Colledanchise, *Student Member, IEEE*, and Petter Ögren, *Member, IEEE*

***Abstract*—Behavior trees (BTs) are a way of organizing the switching structure of a hybrid dynamical system (HDS), which was originally introduced in the computer game programming community. In this paper, we analyze how the BT representation increases the modularity of an HDS and how key system properties are preserved over compositions of such systems, in terms of combining two BTs into a larger one. We also show how BTs can be seen as a generalization of sequential behavior compositions, the subsumption architecture, and decisions trees. These three tools are powerful but quite different, and the fact that they are unified in a natural way in BTs might be a reason for their popularity in the gaming community. We conclude the paper by giving a set of examples illustrating how the proposed analysis tools can be applied to robot control BTs.**

***Index Terms*—Behavior trees (BTs), decision trees, finite state machines (FSMs), hybrid dynamical systems (HDSs), modularity, subsumption architecture, sequential behavior compositions.**

I. Introduction

BEHAVIOR trees (BTs) were developed in the computer gaming industry, as a tool to increase *modularity* in the control structures of in-game opponents [1]–[5]. In this billion dollar industry, modularity is a key property to enable reusability of code, incremental design of functionality, and efficient testing of that functionality.

In games, the control structures of in-game opponents are naturally formulated in terms of hybrid dynamical systems (HDSs), i.e., dynamical systems that have a continuous part, such as motion in a virtual environment, and a discrete part, such as decision making, in terms of switching between different continuous controllers. Furthermore, the discrete parts of these HDSs are often modeled as finite state machines (FSMs).

Manuscript received June 30, 2016; accepted October 27, 2016. Date of publication December 19, 2016; date of current version April 3, 2017. This paper was recommended for publication by Associate Editor H. Kress-Gazit and Editor C. Torras upon evaluation of the reviewers' comments. This work was supported by the SARAFun project, partially funded by the EU within H2020 (H2020-ICT-2014/H2020-ICT-2014-1) under Grant 644938.

The authors are with the Center for Autonomous Systems, Department of Computer Vision and Active Perception, KTH—Royal Institute of Technology, Stockholm 114 28, Sweden (e-mail: miccol@kth.se; petter@kth.se).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org

Digital Object Identifier 10.1109/TRO.2016.2633567

However, just as Petri Nets [6] provide an alternative view of FSMs that emphasize *concurrency*, BTs provide an alternative view of FSMs that emphasize *modularity*. How BTs modularize HDS will be discussed in Section IV, but here we note that the core difference is that the *transitions* (one-way control transfers) of the FSM are replaced with *function calls* (two-way control transfers) up and down the tree structure of the BTs.

Following the development in the industry, BTs have now also started to receive attention in academia, see, e.g., [7]–[17].

At Carnegie Mellon University, BTs have been used extensively for robotic manipulation [12], [15]. The fact that modularity is the key reason for using BTs is clear from the following quote: "The main advantage is that individual behaviors can easily be reused in the context of another higher level behavior, without needing to specify how they relate to subsequent behaviors." from [12].

BTs have also been used to enable nonexperts to do robot programming of pick and place operations, due to their "modular, adaptable representation of a robotic task" [17] and proposed as a key component in brain surgery robotics due to the "flexibility, reusability, and simple syntax" [16].

The advantage of BTs as compared to FSMs was also the reason for extending the JADE agent behavior model with BTs in [10], and the benefits of using BTs to control complex multimission UAVs was described in [11].

The modularity and structure of BTs were used to address the formal verification of mission plans in [13] and the execution times of stochastic BTs were analyzed in [14]. BTs have also been studied in machine learning applications [7], [8] and details regarding efficient parameter passing was investigated in [9]. Finally, a Modelica implementation of BTs was presented in [18].

In this paper, we investigate the key property of BTs, *modularity*, by using standard tools from robot control theory. The benefits of modularity become even clearer when key system properties can be shown to be preserved *across compositions* of smaller modules into bigger systems. We will try to capture to what extent this holds for BTs. The key properties we investigate is *efficiency*, in terms of time to successful completion; *safety*, in terms of avoiding particular parts of the state space; and *robustness*, in terms of large regions of attraction, as shown in Fig. 1.

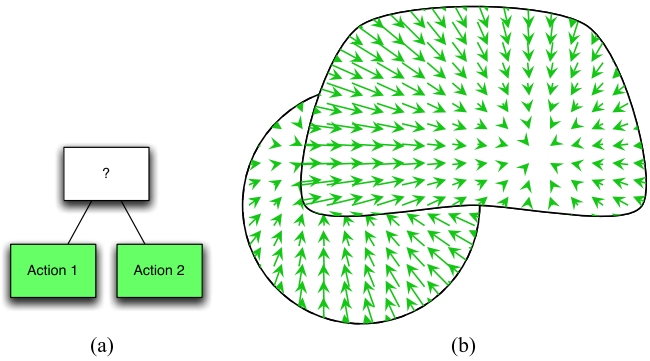

Fig. 1. Minimalist BT composition (a) and the corresponding vector field (b). The second subtree increases the robustness of the composition by increasing the combined region of attraction.

As noted above, the reason BTs are more modular than FSMs is that they use a two-way control transfer, where behavior switching is defined by the context of the parent behavior. To capture this formally, we define a *functional* version of BTs, and use this model to analyze how the key properties mentioned above are transferred across BT compositions.

Performing this analysis, we also show that BTs can be seen as generalizations of three classical concepts from the robot control literature, the *subsumption architecture* [19], *sequential behavior compositions* [20], and *decision trees* [21].

The subsumption architecture [19] is a control structure where a number of controllers are executed in parallel, and higher priority controllers subsume (or suppress), the lower priority ones, whenever needed.

Sequential behavior compositions were introduced in [20] and built upon in, e.g., [22]. The key idea is that the region of attraction of a controller can be increased by combining a set of different controllers, where each controller drives the system state into the region of attraction of another controller, closer to the overall goal state.

Decision trees [21] is a control structure where the controllers are found at the leaves of the tree, and the interior nodes of the tree represent state-dependent predicates, which determine what branches to follow from the root to one of the leaves.

The contributions of this paper is that we formally investigate and capture the modularity of BTs, by introducing a functional representation. This formulation enables us to show the results regarding safety, efficiency, and robustness of modular compositions of BTs. We also explore how BTs generalize three classical concepts from the robot control literature, and the connection between BTs and FSMs. This paper extends the conference paper [23] by adding results on the efficiency of sequence compositions, the analysis of decision trees, a detailed analysis of the relation between BTs and FSMs, and more examples illustrating modularity, and the use of the theoretical results.

The outline of this paper is as follows. In Section II, we review the classical formulation of BTs. Then, in Section III, we introduce a new compact function call formulation of BTs. In Section IV we describe how the BTs modularize hybrid control systems, both conceptually and in terms of how system properties are preserved under module compositions. Then, the way in which BTs generalize a number of existing control structures is investigated in Section V. Finally, a complex example is given in Section VI, and the conclusion is drawn in Section VII.

II. Background: Classical Formulation of BTs

In this section, we will describe BTs in the classical way that can be found in textbooks, such as [4], [5], and papers on game AI such as [1] and [3]. The following section (III) will then provide a functional description of BTs that will be used for our formal analysis.

Let BT be a directed tree, with the usual definition of nodes, edges, root, leaves, children, and parents. In a BT, each node belongs to one of the five categories listed in Table I. Leaf nodes are either *Actions* or *Conditions*, while interior nodes are either *Fallbacks*, *Sequences* or *Parallels*. A minimalistic example BT composed of one Fallback node and two Action nodes can be seen in Fig. 2.

When a BT is executed, the root node is *ticked* with a given frequency, and corresponding timestep Δt. This tick will then progress downwards through the tree, following the rules of the different node types, until it reaches a leaf node. There, some computations are made, often taking both internal states and sensor data into account. If the leaf node is an Action, it might issue some commands to the robot actuators, and it returns either *Success*, *Failure* or *Running* to its parent. The parent node then either returns the same message to its parent, or chooses to tick another child who in turn returns *Success*/*Failure*/*Running*, and so on. We will now describe how this works in more detail. The first node type in Table I is the Fallback.

Fallback[1]: Fallbacks are used when a set of actions represent alternative ways of reaching a similar goal. Thus, Fallbacks will try each of its children, from left to right, and return Success as soon as it has found one child that returns Success. It will return Running as long as the ticked child returns Running and Failure only when all children have failed, as shown in Table I and the pseudocode below.

Looking at the example BT in Fig. 2, the Fallback has two actions, *Enter through Front Door* and *Enter through Back Door*, each with the common purpose of *Enter Building* (the name of the whole BT). The root of the BT is the Fallback, and the Actions are the leaves. According to the pseudocode in Algorithm 1, when the root/Fallback is ticked, it ticks its first child. The Action *Enter through Front Door* then starts executing the corresponding continuous robot controller, and returns *Running*. The Fallback/root also returns *Running*. Then, after the given time step Δt, a new tick is sent from the root, and the whole process is repeated. The return status of the different nodes probably remains the same for a number of time steps. Then, at some point, *Enter through Front Door* does not return *Running* anymore, but instead returns either *Success* if it managed to enter through the door, or *Failure* if it did not manage. In case of Success, the Fallback also returns Success, but in the case of Failure, the Fallback instead starts ticking *Enter through*

[1]Fallbacks are sometimes also called selectors.

TABLE I
FIVE NODE TYPES OF A BT

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

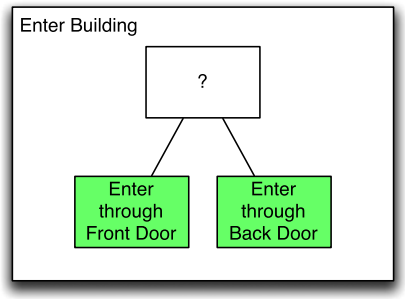

Fig. 2. Fallback is used to create an *Enter Building* BT. The back door option is only tried if the front door option fails. Fallbacks are denoted by a white box with a question mark and actions are denoted by a green box.

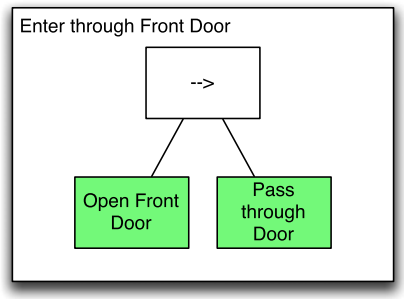

Fig. 3. Sequence is used to create an *Enter Through Front Door* BT. Passing the door is only tried if the opening action succeeds. Sequences are denoted by a white box with an arrow.

Back Door, which probably returns Running for a number of ticks. Finally, when *Enter through Back Door* returns either *Success* or *Failure*, the fallback will return the corresponding thing, as there are no more options to try in case of Failure, and no more options needed in case of Success.

The second node type is Sequence, and a minimalistic BT using a Sequence can be found in Fig. 3.

Sequence: Sequences are used when some actions are meant to be carried out in sequence, and when the success of one action is needed for the execution of the next. Thus, Sequences find and execute the first child that does not return success. A Sequence will return immediately with a status code failure or running when one of its children returns failure or running, see Table I and the pseudocode in Algorithm 2. The children are ticked in order, from left to right.

Looking at the example BT in Fig. 3, the Sequence has two actions, *Open Front Door* and *Pass through Door*. If both succeed, the whole BT, *Enter through Front Door*, will succeed. But if the first action fails, the overall task has failed, and there is no point in trying the second action.

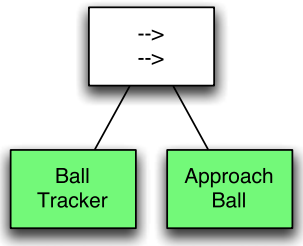

Fig. 4. Two actions Ball Tracker (sensing) and Approach Ball (actuator control) are ticked and executed in parallel. Parallel nodes are denoted by a white box with two arrows.

Remark 1: The definition above corresponds to so-called memoryless Sequences. Most BT implementations also include a Sequence with memory, where a subtree that returned Succeed is never executed again.

The third node type is Parallel, and a minimalistic BT using a parallel node can be found in Fig. 4.

Parallel: A parallel node ticks all its children simultaneously. If M out of the N children return success, then so does the parallel node. If more than $N - M$ return failure, thus rendering success impossible, it returns failure. If none of the conditions above are met, it returns running.

We will now define the two types of leaf nodes.

Action: An Action node performs an action, and returns Success if the action is completed, Failure if it cannot be completed and Running if completion is under way.

Condition: A Condition node determines if a given condition has been met, therefore, success/failure are often interpreted as true/false. Conditions are technically a subset of the Actions, but are given a separate category and graphical symbol to improve readability of the BT and emphasize the fact that they never return running and do not change any internal states/variables of the BT. Examples of Conditions can be found in Fig. 5.

We conclude this section with an illustration of how smaller BTs can be combined into larger ones and a remark on nonreactive BTs.

The BT in Fig. 6 is a straightforward combination of Figs. 2 and 3. If we add the battery power check of Fig. 5, and some additional actions such as *Close Front Door* (in Sequence with Pass through Front Door) and *Smash Back Door* (as a fallback of Open Back Door), we get the BT in Fig. 7.

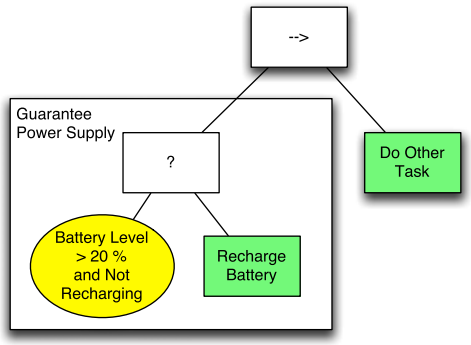

Fig. 5. Condition is used to decide when to recharge the batteries. In each tick of the tree, the battery levels are checked, and the *Do Other Task* Action is stopped whenever the battery level is getting too low.

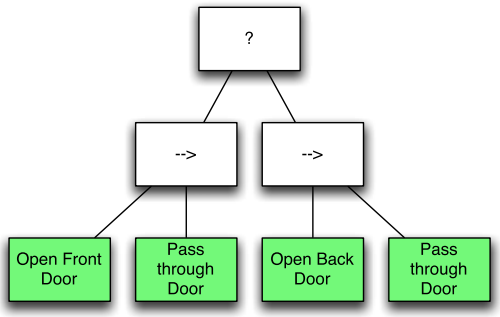

Fig. 6. Two BTs in Figs. 2 and 3 are combined to larger BT. If, e.g., the robot opens the front door, but does not manage to pass through it, it will try the back door.

Remark 2: Some BT implementations do not include the *Running* return status [4]. Instead, they let each action run until it returns *Failure* or *Success*. We denote these BTs nonreactive, since they do not allow actions other than the currently active one to react to changes. This is a significant limitation on nonreactive BTs, which was also noted in [4].

III. New Functional Formulation of BTs

In this section, we present a new functional formulation of the BTs described above. The new formulation is more formal, and will allow us to analyze how properties are preserved over modular compositions of BTs. In the functional version, the *tick* is replaced with a recursive function call that include both the return status, the system dynamics and the system state. The details of the formulation are derived from the pseudocode in Algorithms 1–3.

Definition 1 (Behavior Tree): A BT is a three tuple

$$\mathcal{T}_i = \{f_i, r_i, \Delta t\} \tag{1}$$

where $i \in \mathbb{N}$ is the index of the tree, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ is the right-hand side of an ordinary difference equation, Δt is a time step and $r_i : \mathbb{R}^n \to \{\mathcal{R}, \mathcal{S}, \mathcal{F}\}$ is the return status that can be equal to either *Running* ($\mathcal{R}$), *Success* ($\mathcal{S}$), or *Failure* ($\mathcal{F}$). Let the Running/Activation region (R_i), Success region (S_i), and Failure region (F_i) correspond to a partitioning of the state space,

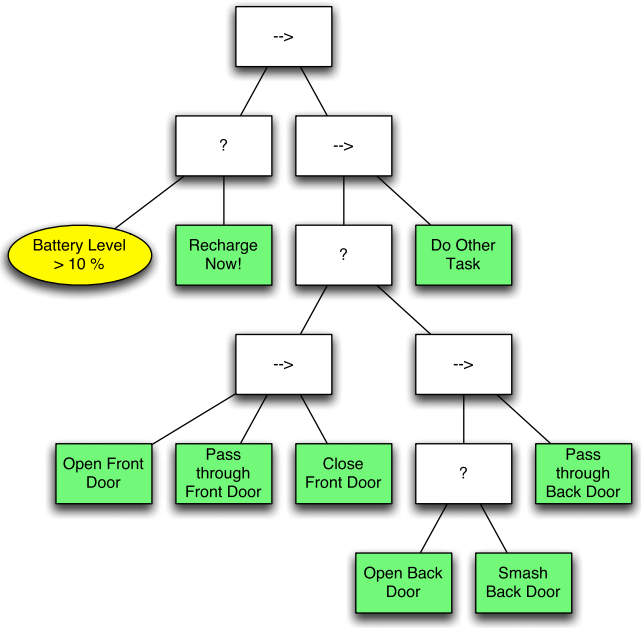

Fig. 7. Combining the BTs above and some additional Actions, we get a flexible BT for entering a building and performing some task.

defined as

$$R_i = \{x : r_i(x) = \mathcal{R}\} \tag{2}$$

$$S_i = \{x : r_i(x) = \mathcal{S}\} \tag{3}$$

$$F_i = \{x : r_i(x) = \mathcal{F}\}. \tag{4}$$

Finally, the execution of a BT $\mathcal{T}_i$ is a standard ordinary difference equation

$$x_{k+t}(t_{k+1}) = f_i(x_k(t_k)) \tag{5}$$

$$t_{k+1} = t_k + \Delta t. \tag{6}$$

The return status r_i will be used when recursively combining BTs, as explained below.

Assumption 1: From now on we will assume that all BTs evolve in the same continuous space $\mathbb{R}^n$ using the same time step Δt_i.

Remark 3: It is often the case, that different BTs, controlling different vehicle subsystems evolving in different state spaces, need to be combined into a single BT. Such cases can be accommodated in the assumption above by letting all systems evolve in a larger state space, that is the Cartesian product of the smaller state spaces.

The five node types of Table I are given functional representations as follows. BTs that satisfy Definition 1 directly, without calling other subtrees, are called Actions and Conditions, with the later ones never returning Running. The three composition nodes, corresponding to Algorithms 1–3 are defined below.

Definition 2 (Sequence compositions of BTs): Two or more BTs can be composed into a more complex BT using a Sequence

Algorithm 1: Pseudocode of a Fallback node with N children.

```
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus = running then
        return running
    else if childStatus = success then
        return success
return failure
```

Algorithm 2: Pseudocode of a Sequence node with N children.

```
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus = running then
        return running
    else if childStatus = failure then
        return failure
return success
```

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

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

operator

$$\mathcal{T}_0 = \text{Sequence}\,(\mathcal{T}_1, \mathcal{T}_2).$$

Then r_0, f_0 are defined as

$$\text{If } x_k \in S_1 \tag{7}$$
$$r_0(x_k) = r_2(x_k) \tag{8}$$
$$f_0(x_k) = f_2(x_k) \tag{9}$$
$$\text{else}$$
$$r_0(x_k) = r_1(x_k) \tag{10}$$
$$f_0(x_k) = f_1(x_k). \tag{11}$$

$\mathcal{T}_1$ and $\mathcal{T}_2$ are called children of $\mathcal{T}_0$. Note that when executing the new BT, $\mathcal{T}_0$ first keeps executing its first child $\mathcal{T}_1$ as long as it returns Running or Failure. The second child is executed only when the first returns Success, and $\mathcal{T}_0$ returns Success only when all children have succeeded, hence the name Sequence.

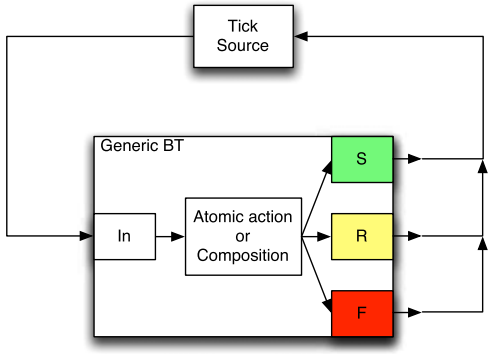

Fig. 8. FSM behaving like a BT, made up of a single normal state, three out transitions Success (S), Running (R), Failure (F), and a Tick source.

For notational convenience, we write

$$\text{Sequence}\,(\mathcal{T}_1, \text{Sequence}\,(\mathcal{T}_2, \mathcal{T}_3)) = \text{Sequence}\,(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3) \tag{12}$$

and similarly for arbitrarily long compositions.

Definition 3 (Fallback compositions of BTs): Two or more BTs can be composed into a more complex BT using a Fallback operator

$$\mathcal{T}_0 = \text{Fallback}\,(\mathcal{T}_1, \mathcal{T}_2).$$

Then r_0, f_0 are defined as

$$\text{If } x_k \in F_1 \tag{13}$$
$$r_0(x_k) = r_2(x_k) \tag{14}$$
$$f_0(x_k) = f_2(x_k) \tag{15}$$
$$\text{else}$$
$$r_0(x_k) = r_1(x_k) \tag{16}$$
$$f_0(x_k) = f_1(x_k). \tag{17}$$

Note that when executing the new BT, $\mathcal{T}_0$ first keeps executing its first child $\mathcal{T}_1$ as long as it returns Running or Success. The second child is executed only when the first returns Failure, and $\mathcal{T}_0$ returns Failure only when all children have tried, but failed, hence the name Fallback.

For notational convenience, we write

$$\text{Fallback}\,(\mathcal{T}_1, \text{Fallback}\,(\mathcal{T}_2, \mathcal{T}_3)) = \text{Fallback}\,(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3) \tag{18}$$

and similarly for arbitrarily long compositions.

Parallel compositions only make sense if the BTs to be composed control separate parts of the state space, thus we make the following assumption.

Assumption 2: Whenever two BTs $\mathcal{T}_1, \mathcal{T}_2$ are composed in parallel, we assume that there is a partition of the state space $x = (x_1, x_2)$ such that $f_1(x) = (f_{11}(x), f_{12}(x))$ implies $f_{12}(x) = 0$ and $f_2(x) = (f_{21}(x), f_{22}(x))$ implies $f_{21}(x) = 0$ (i.e., the two BTs control different parts of the system).

Definition 4 (Parallel compositions of BTs): Two or more BTs can be composed into a more complex BT using a

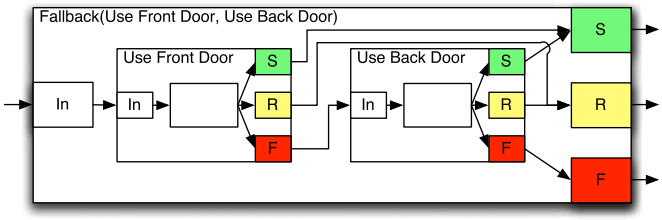

Fig. 9. FSM corresponding to the Fallback BT in Fig. 2. Note how the second state is executed only if the first fails.

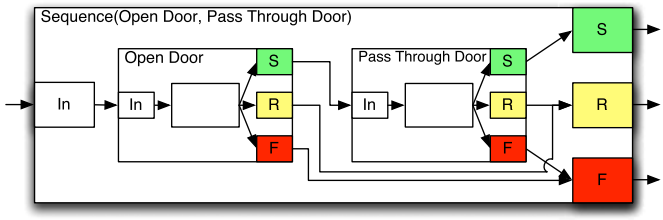

Fig. 10. FSM corresponding to the Sequence BT in Fig. 3. Note how the second state is executed only if the first succeeds.

Parallel operator,

$$\mathcal{T}_0 = \text{Parallel}(\mathcal{T}_1, \mathcal{T}_2).$$

Let $x = (x_1, x_2)$ be the partitioning of the state space described in Assumption 2, then $f_0(x) = (f_{11}(x), f_{22}(x))$ and r_0 is defined as

$$\begin{aligned}
&\text{If } M = 1 \\
&r_0(x) = \mathcal{S} \text{ If } r_1(x) = \mathcal{S} \vee r_2(x) = \mathcal{S} && (19)\\
&r_0(x) = \mathcal{F} \text{ If } r_1(x) = \mathcal{F} \wedge r_2(x) = \mathcal{F} && (20)\\
&r_0(x) = \mathcal{R} \text{ else} && (21)\\
&\text{If } M = 2 \\
&r_0(x) = \mathcal{S} \text{ If } r_1(x) = \mathcal{S} \wedge r_2(x) = \mathcal{S} && (22)\\
&r_0(x) = \mathcal{F} \text{ If } r_1(x) = \mathcal{F} \vee r_2(x) = \mathcal{F} && (23)\\
&r_0(x) = \mathcal{R} \text{ else.} && (24)
\end{aligned}$$

IV. How BTs Modularize HDSs

In this section, we will show how BTs modularize the FSMs in HDS. We believe that this modularity is important when designing, testing, and reusing complex task switching structures.

First we show how FSMs can be given the structure of BTs, then we make an informal argument based on a comparison of function calls with GOTO-statements. Then, we will make a formal argument by showing how some system properties are preserved under modular compositions of BTs.

A. Giving an FSM the Structure of a BT

As described above, each BT returns *Success*, *Running* or *Failure*. Imagine we have a state in an FSM that has three transitions, corresponding to these three return statements. Adding a Tick source that collect the return transitions and transfer the execution back into the state, as depicted in Fig. 8, we have a structure that resembles a BT.

We can now compose such FSM states by using both Fallback and Sequence constructs. The FSM corresponding to the Fallback example in Fig. 2 would then look like the one shown in Fig. 9.

Similarly, the FSM corresponding to the Sequence example in Fig. 3 would then look like the one shown in Fig. 10, and a two-level BT, such as the one in Fig. 6 would look like Fig. 11.

A few observations can be made from the above examples. First, it is perfectly possible to design FSMs, and therefore HDSs with a structure taken from BTs. Second, considering that a BT with two levels corresponds to the FSM in Fig. 11, a BT with five levels, such as the one in Fig. 7 would correspond to a somewhat complex FSM.

Third, and more importantly, the *modularity* of the BT construct is illustrated in Figs. 8–11. Fig. 11 might be complex, but that complexity is encapsulated in a box with a single in-transition and three out-transitions, just as the box in Fig. 8.

Fourth, the decision of what to do after a given sub-BT returns is always decided on the parent level of that BT. The sub-BT is ticked, and returns *Success*, *Running* or *Failure* and the parent level decided whether to tick the next child, or return something to its own parent. Thus, the BT ticking and returning of a sub-BT is similar to a *function call* in a piece of source code. A function call in Java, C++ or Python moves execution to another piece of the source code, but then returns the execution to the line right below the function call. What to do next is decided by the piece of code that made the function call, not the function itself. As we will see below, this is quite different from standard FSMs where the decision of what to do next is decided by the state being transitioned to, in a way that resembles the GOTO statement.

B. Function Calls and GOTO Statements

In this section, we will argue that the switching structure provided by BTs supports modularity.

The switching structure of an HDS is given by the transitions of an FSM. These transitions are intuitive, straightforward, and compact. However, they represent control transfers that are so-called *one-way* and thus share the drawbacks that made the GOTO-statement obsolete.

Forty years ago, a control flow statement called GOTO was used extensively in computer programming. Today, this feature has been abandoned by most general purpose programming languages, and the reasons for this was formulated in a famous quote by Edsgar Dijkstra in his paper GOTO *statement considered harmful* [24]: “The GOTO statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program”.

To understand the rationale behind Dijkstas statement, we note that GOTO statements are *one-way control transfers*, where the execution is transferred somewhere in a more or less memoryless fashion. The alternative to one-way control transfers is the *two-way control transfer* embodied in, e.g., function calls.

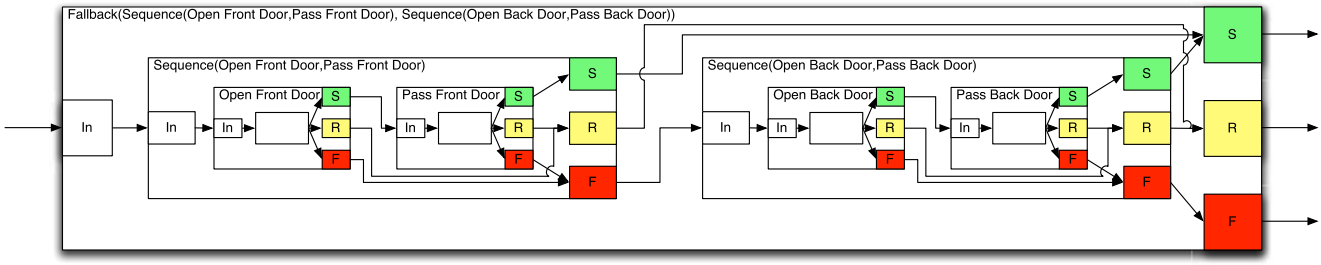

Fig. 11. FSM corresponding to the BT in Fig. 6.

Here, control is transferred back to the place of the function call, together with a result of the computation in the function. Thus, the *implementation* of the function does not depend on *how the results will be used*, and the user of the function does not have to know how it is implemented. On the contrary, in one-way control transfers, the implementation of the functionality must also include instructions of what to do next. This fact couples implementation and usage, and makes modular design less straightforward.

Looking at the state machines in HDSs, we note that the state transitions are indeed one-way control transfers. The called state must also include instructions of what to do next. As above, this fact sometimes makes designing a modular HDS using FSMs quite difficult.

One final, and smaller, drawback of FSMs lies in the graphical representation. The FSM has arrows for possible transitions, but the actual conditions for the transfers has no graphical representation. For BTs, it is clear from the tree structure and node types what a success/failure will mean for the future execution.

Note however, that there are *no* claims that BTs are superior to FSMs from a purely theoretical standpoint. On the contrary, all BTs can most likely be formulated in terms of an FSM, just as most general purpose programming languages are equivalent in the sense of Turing completeness, but still differ in modularity, readability, and reusability of code.

C. How BTs Modularize Efficiency and Robustness

In this section, we will show how some aspects of time efficiency and robustness carry across modular compositions of BTs. This result will then enable us to conclude that if two BTs are "efficient," then their composition will also be "efficient," if the right conditions are satisfied. We also show how the Fallback composition can be used to increase the region of attraction of a BT, thereby making it more robust to uncertainties in the initial configuration.

Note that in this paper, as in [20], by robustness we mean large regions of attraction. We do not investigate, e.g., disturbance rejection, or other forms of robustness.

Many control problems, in particular in robotics, can be formulated in terms of achieving a given goal configuration in a way that is time efficient and robust with respect to the initial configuration. Since all BTs return either Success, Failure or Running, the definitions below will include a finite time, at which Success must be achieved.

In order to formalize the discussion above, we say that *efficiency* can be measured by the size of the time bound τ in Definition 5 and *robustness* can be measured by the size of the region of attraction R' in the same definition.

Definition 5 (Finite Time Successful): A BT is finite time successful (FTS) with region of attraction R', if for all starting points $x(0) \in R' \subset R$, there is a time τ, and a time $\tau'(x(0))$ such that $\tau'(x) \leq \tau$ for all starting points, and $x(t) \in R'$ for all $t \in [0, \tau')$ and $x(t) \in S$ for all $t \geq \tau'$).

As noted in the following lemma, exponential stability implies finite time success, given the right choices of the sets S, F, R.

Lemma 1 (Exponential stability and FTS): A BT for which x_s is a globally exponentially stable equilibrium of the execution (5), and $S \supset \{x : ||x - x_s|| \leq \epsilon\}$, $\epsilon > 0$, $F = \emptyset$, $R = \mathbb{R}^n \setminus S$, is FTS.

Proof: Global exponential stability implies that there exists $a > 0$ such that $||x(k) - x_s|| \leq e^{-\mathrm{ak}}$ for all k. Then, for each ϵ there is a time τ such that $||x(k) - x_s|| \leq e^{-a\tau} < \epsilon$, which implies that there is a $\tau' < \tau$ such that $x(\tau') \in S$ and the BT is FTS. ■

We are now ready to look at how these properties extend across compositions of BTs.

Lemma 2: (Robustness and efficiency of sequence compositions) If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, with $S_1 = R_2' \cup S_2$, then $\mathcal{T}_0 =$ Sequence $(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with $\tau_0 = \tau_1 + \tau_2$, $R_0' = R_1' \cup R_2'$ and $S_0 = S_1 \cap S_2$.

Proof: First we consider the case when $x(0) \in R_1'$. Then, as $\mathcal{T}_1$ is FTS, the state will reach S_1 in a time $k_1 < \tau_1$, without leaving R_1'. Then $\mathcal{T}_2$ starts executing, and will keep the state inside S_1, since $S_1 = R_2' \cup S_2$. $\mathcal{T}_2$ will then bring the state into S_2, in a time $k_2 < \tau_2$, and $\mathcal{T}_0$ will return Success. Thus, we have the combined time $k_1 + k_2 < \tau_1 + \tau_1$.

If $x(0) \in R_2'$, $\mathcal{T}_1$ immediately returns Success, and $\mathcal{T}_2$ starts executing as above. ■

The lemma above is illustrated in Fig. 12, and Example 1 below.

Example 1: Consider the BT in Fig. 3. If we know that *Open Front Door* is FTS and will finish in less than τ_1 seconds, and that *Pass through Door* is FTS and will finish in less than τ_2 seconds. Then, as long as $S_1 = R_2' \cup S_2$, Lemma 2 states that the combined BT in Fig. 3 is also FTS, with an upper bound on the execution time of $\tau_1 + \tau_2$. Note that the condition $S_1 = R_2' \cup S_2$ implies that the action *Pass through Door* will not make the system leave S_1, by, e.g., accidentally colliding with the door and thereby closing it without having passed through it.

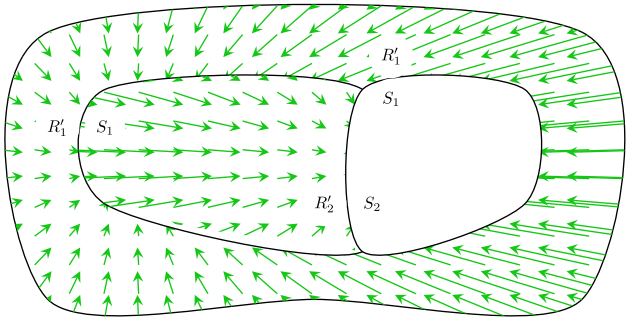

Fig. 12. Sets R_1', S_1, R_2', S_2 of Example 1 and Lemma 2.

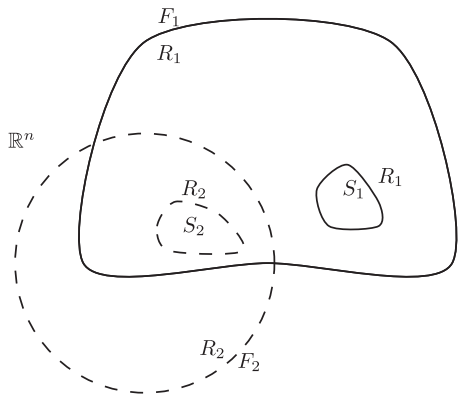

Fig. 13. Sets S_1, F_1, R_1 (solid boundaries) and S_2, F_2, R_2 (dashed boundaries) of Example 2 and Lemma 3.

The result for Fallback compositions is related, but with a slightly different condition on S_i and R_j'.

Lemma 3: (Robustness and efficiency of fallback compositions) If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, with $S_2 \subset R_1'$, then $\mathcal{T}_0 =$ Fallback $(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with $\tau_0 = \tau_1 + \tau_2$, $R_0' = R_1' \cup R_2'$ and $S_0 = S_1$.

Proof: First we consider the case when $x(0) \in R_1'$. Then, as $\mathcal{T}_1$ is FTS, the state will reach S_1 before $k = \tau_1 < \tau_0$, without leaving R_1'. If $x(0) \in R_2' \setminus R_1'$, $\mathcal{T}_2$ will execute, and the state will progress toward S_2. But as $S_2 \subset R_1'$, $x(k_1) \in R_1'$ at some time $k_1 < \tau_2$. Then, we have the case above, reaching $x(k_2) \in S_1$ in a total time of $k_2 < \tau_1 + k_1 < \tau_1 + \tau_2$. ■

The Lemma above is illustrated in Fig. 13, and Example 2 below.

Remark 4: As can be noted, the necessary conditions in Lemma 2, including $S_1 = R_2' \cup S_2$ might be harder to satisfy than the conditions of Lemma 3, including $S_2 \subset R_1'$. Therefore, Lemma 3 is often preferable from a practical point of view, e.g., using implicit sequences as shown below.

Example 2: This example will illustrate a particular way of using Fallbacks that we call *Implicit sequences*. Consider the BT in Fig. 14. During execution, if the door is closed, then *Pass through Door* will fail and *Open Front Door* will start to execute. Now, right before *Open Front Door* returns Success, the first action *Pass through Door* (with higher priority) will realize that the state of the world has now changed enough to enable a possible success and starts to execute, i.e., return Running instead of Failure. The combined action of this BT will thus make the robot open the door (if necessary) and then pass through if.

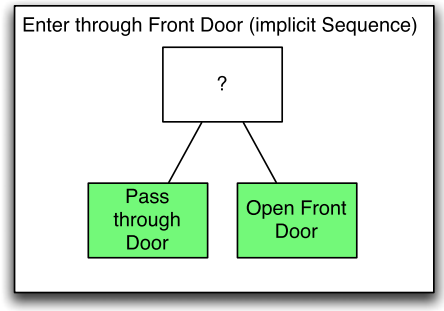

Fig. 14. Implicit sequence created using a Fallback, as described in Example 2 and Lemma 3.

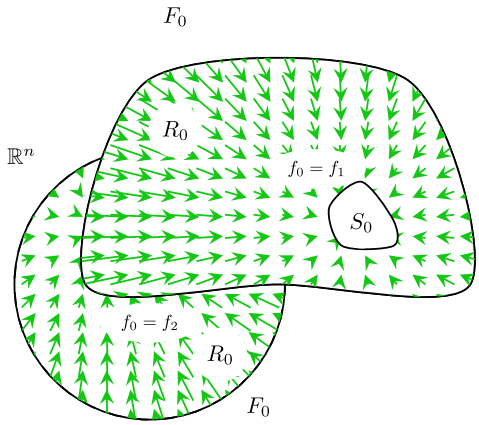

Fig. 15. Sets S_0, F_0, R_0 and the vector field $(f_0(x) - x)$ of Example 2 and Lemma 3.

Thus, even though a Fallback composition is used, the result is sometimes a sequential execution of the children in reverse order (from right to left). Hence the name Implicit sequence.

The above example illustrates how we can increase the robustness of a BT. If we want to be able to handle more diverse situations, such as a closed door, we do not have to make the door passing action more complex, instead we combine it with another BT that can handle the situation and move the system into a part of the state space that the first BT can handle. The sets S_0, F_0, R_0, and f_0 of the combined BT are shown in Fig. 15, together with the vector field $f_0(x) - x$. As can be seen, the combined BT can now move a larger set of initial conditions to the desired region $S_0 = S_1$.

Lemma 4: (Robustness and efficiency of parallel compositions) If $\mathcal{T}_1, \mathcal{T}_2$ are FTS, then $\mathcal{T}_0 =$ Parallel$(\mathcal{T}_1, \mathcal{T}_2)$ is FTS with

$$\text{If } M = 1$$
$$R_0' = \{R_1' \cup R_2'\} \setminus \{S_1 \cup S_2\} \quad (25)$$
$$S_0 = S_1 \cup S_2 \quad (26)$$
$$\tau_0 = \min(\tau_1, \tau_2) \quad (27)$$
$$\text{If } M = 2$$
$$R_0' = \{R_1' \cap R_2'\} \setminus \{S_1 \cap S_2\} \quad (28)$$
$$S_0 = S_1 \cap S_2 \quad (29)$$
$$\tau_0 = \max(\tau_1, \tau_2). \quad (30)$$

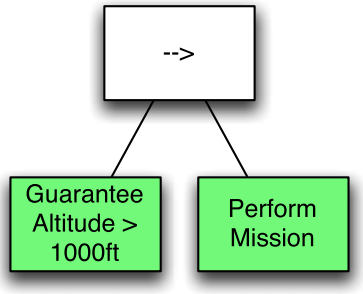

Fig. 16. Safety of the UAV control BT is guaranteed by the first Action.

Proof: The parallel composition executes $\mathcal{T}_1$ and $\mathcal{T}_2$ independently. If $M = 1$ the parallel composition returns success if either $\mathcal{T}_1$ or $\mathcal{T}_2$ returns success, thus $\tau_0 = \min(\tau_1, \tau_2)$. It returns running if either $\mathcal{T}_1$ or $\mathcal{T}_2$ returns running and the other does not return success. If $M = 2$ the parallel composition returns success if and only if both $\mathcal{T}_1$ and $\mathcal{T}_2$ return success, thus $\tau_0 = \max(\tau_1, \tau_2)$. It returns running if either $\mathcal{T}_1$ or $\mathcal{T}_2$ returns running and the other does not return failure. ■

D. How BTs Modularize Safety

Besides being efficient and robust, we also want our robot system to be *safe*, in the sense that it by design never enters a particular part of the state space, that we for simplicity denote the *Obstacle Region*. We make the following definition.

Definition 6 (Safe): A BT is Safe, with respect to the obstacle region $O \subset \mathbb{R}^n$, and the initialization region $I \subset R$, if for all starting points $x(0) \in I$, we have that $x(t) \notin O$, for all $t \geq 0$.

In order to make statements about the safety of composite BTs we also need the following definition.

Definition 7 (Safeguarding): A BT is Safeguarding, with respect to the step length d, the obstacle region $O \subset \mathbb{R}^n$, and the initialization region $I \subset R$, if it is safe, and FTS with region of attraction $R' \supset I$ and a success region S, such that I surrounds S in the following sense:

$$\{x \in X \subset \mathbb{R}^n : \inf_{s \in S} ||x - s|| \leq d\} \subset I \tag{31}$$

where X is the reachable part of the state space $\mathbb{R}^n$.

This implies that the system, under the control of another BT with maximal state space step length d, cannot leave S without entering I, and thus avoiding O, see Lemma 5.

Example 3: To illustrate how safety can be improved by using a Sequence composition, we consider the UAV control BT in Fig. 16. The sets S_i, F_i, R_i are shown in Fig. 17. As $\mathcal{T}_1$ is Guarantee altitude above 1000 ft, its failure region F_1 is a small part of the state space (corresponding to a crash) surrounded by the running region R_1 that is supposed to move the UAV away from the ground, guaranteeing a minimum altitude of 1000 ft. The success region S_1 is large, every state sufficiently distant from F_1. The BT that performs the mission, $\mathcal{T}_2$, has a smaller success region S_2, surrounded by a very large running region R_2, containing a small failure region F_2. The function f_0 is

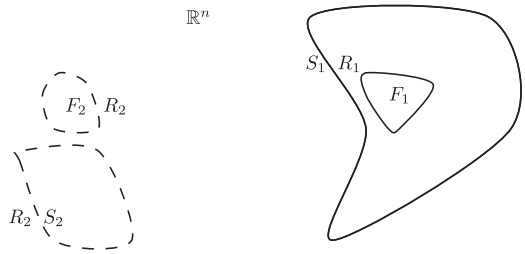

Fig. 17. Sets S_1, F_1, R_1 (solid boundaries) and S_2, F_2, R_2 (dashed boundaries) of Example 3 and Lemma 5.

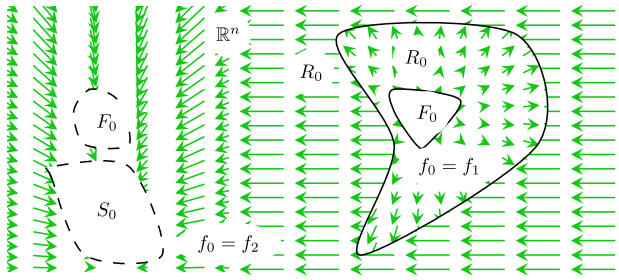

Fig. 18. Sets S_0, F_0, R_0 and the vector field $(f_0(x) - x)$ of Example 3 and Lemma 5.

governed by (9) and (11) and is depicted in form of the vector field $(f_0(x) - x)$ in Fig. 18 .

The discussion above is formalized in Lemma 5.

Lemma 5 (Safety of sequence compositions): If $\mathcal{T}_1$ is safeguarding, with respect to the obstacle O_1 initial region I_1, and margin d, and $\mathcal{T}_2$ is an arbitrary BT with $\max_x ||x - f_2(x)|| < d$, then the composition $\mathcal{T}_0 = \text{Sequence}\,(\mathcal{T}_1, \mathcal{T}_2)$ is Safe with respect to O_1 and I_1.

Proof: $\mathcal{T}_1$ is safeguarding, which implies that $\mathcal{T}_1$ is safe and thus any trajectory starting in I_1 will stay out of O_1 as long as $\mathcal{T}_1$ is executing. But if the trajectory reaches S_1, $\mathcal{T}_2$ will execute until the trajectory leaves S_1. We must now show that the trajectory cannot reach O_1 without first entering I_1. But any trajectory leaving S_1 must immediately enter I_1, as the first state outside S_1 must lie in the set $\{x \in \mathbb{R}^n : \inf_{s \in S_1} ||x - s|| \leq d\} \subset I_1$ due to the fact that for $\mathcal{T}_2$, $||x(k) - x(k+1)|| = ||x(k) - f_2(x(k))|| < d$.

We conclude this section with a discussion about undesired chattering in switching systems.

The issue of undesired chattering, i.e., switching back and forth between different subcontrollers, is always an important concern while designing switched control systems, and BTs are no exception. As is suggested by the right part of Fig. 18, chattering can be a problem when vector fields meet at a switching surface.

Although the efficiency of some compositions can be computed by using Lemmas 2 and 3 above, the efficiency of others can be significantly reduced by chattering, as noted above. Inspired by Filippov and Arscott [25], the following result can give an indication of when chattering is to be expected.

Let R_i and R_j be the running region of $\mathcal{T}_i$ and $\mathcal{T}_j$, respectively. We want to study the behavior of the system when a composition

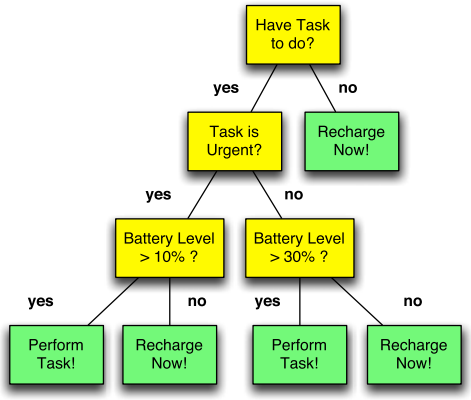

Fig. 19. Decision tree of a robot control system. The decisions are interior nodes, and the actions are leaves.

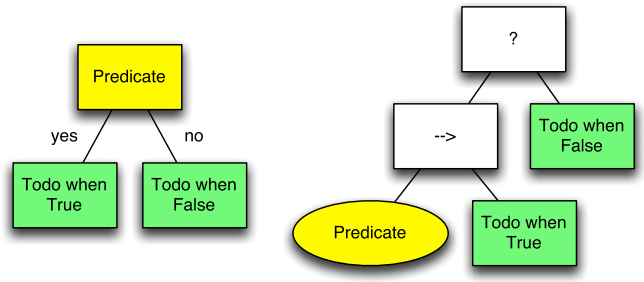

Fig. 20. Basic building blocks of decision trees are "If ...then ...else ..." statements (left), and those can be created in BTs as illustrated above (right).

of $\mathcal{T}_i$ and $\mathcal{T}_j$ is applied. In some cases the execution of a BT will lead to the running region of the other BT and vice-versa. Then, both BTs are alternatively executed and the state trajectory chatters on the boundary between R_i and R_j. We formalize this discussion in the following lemma.

Lemma 6: Given a composition $\mathcal{T}_0 = \text{Sequence}\,(\mathcal{T}_1, \mathcal{T}_2)$, where f_i depend on Δt such that $||f_i(x) - x|| \to 0$ when $\Delta t \to 0$. Let $s : \mathbb{R}^n \to \mathbb{R}$ be such that $s(x) = 0$ if $x \in \delta S_1 \cap R_2$, $s(x) < 0$ if $x \in \text{interior}\,(S_1) \cap R_2$, $s(x) > 0$ if $x \in \text{interior}\,(\mathbb{R}^n \setminus S_1) \cap R_2$, and let

$$\lambda_i(x) = \left(\frac{\partial s}{\partial x}\right)^T (f_i(x) - x).$$

Then, $x \in \delta S_1$ is chatter free, i.e., avoids switching between $\mathcal{T}_1$ and $\mathcal{T}_2$ at every timestep, for small enough Δt, if $\lambda_1(x) < 0$ or $\lambda_2(x) > 0$.

Proof: When the condition holds, the vector field is pointing outwards on at least one side of the switching boundary.

Note that this condition is not satisfied on the right-hand side of Fig. 18. This concludes our analysis of BT compositions.

V. How BTs Generalize Decision Trees, the Subsumption Architecture, and Sequential Behavior Compositions

In this section, we will describe decision trees, the subsumption architecture, and sequential behavior compositions, and see how each of these architectures can be seen as a special case of BTs.

A. How BTs Generalize Decision Trees

Decision trees are tree structures that aggregate a number of *If clauses*, that leads to a given decision or prediction. Each leaf of the tree represents a particular decision, prediction, conclusion, or action to be carried out, and each nonleaf represent a predicate to be checked.

A typical decision tree is shown in Fig. 19. The predicates, evaluating to True/False are found in the interior nodes of the Tree, while the Actions/Conclusions are found at the leaves. Without loss of generality we consider binary decision trees, the extension to multiple choice nodes is straightforward.

In the decision tree of Fig. 19, the robot has to decide whether to perform a given task or recharge its batteries. This decision is taken based upon the urgency of the task and the current battery level. The following Lemma shows how to create an equivalent BT from a given decision tree.

Lemma 7: A decision tree, can be recursively described as

$$DT_i = \begin{cases} DT_{i1}, & \text{if predicate } P_i \text{ is true} \\ DT_{i2}, & \text{if predicate } P_i \text{ is false} \end{cases} \tag{32}$$

where DT_{i1}, DT_{i2} are either atomic actions, or sub-DTs with identical structure. Given such a DT_i, we can create an equivalent BT by setting

$$\mathcal{T}_i = \text{Fallback}\,(\text{Sequence}\,(P_i, \mathcal{T}_{i1}), \mathcal{T}_{i2}) \tag{33}$$

for nonatomic actions, $\mathcal{T}_i = DT_i$ for atomic actions and requiring all actions to return Running all the time.

The original decision tree and the new BT are equivalent in the sense that the same values for P_i will always lead to the same atomic action being executed. The lemma is illustrated in Fig. 20.

Proof: Informally, first we note that by requiring all actions to return Running, we basically disable the feedback functionality that is built into the BT. Instead whatever action that is ticked will be the one that executes, just as the decision tree. Second the result is a direct consequence of the fact that the predicates of the decision trees are essentially "If ...then ...else ..." statements, which can be captured by BTs as shown in Fig. 20. More formally, the BT equivalent of the decision tree is given by

$$\mathcal{T}_i = \text{Fallback}\,(\text{Sequence}\,(P_i, \mathcal{T}_{i1}), \mathcal{T}_{i2}).$$

For the atomic actions always returning running we have $r_i = R$, for the actions being predicates we have that $r_i = P_i$. This, together with Definitions 2 and 3 gives that

$$f_i(x) = \begin{cases} f_{i1}, & \text{if predicate } P_i \text{ is true} \\ f_{i2}, & \text{if predicate } P_i \text{ is false} \end{cases} \tag{34}$$

which is equivalent to (32) ■

Note that this observation opens up the possibilities of using the extensive literature on learning decision trees from human operators, see, e.g., [21], to create BTs. These learned BTs can

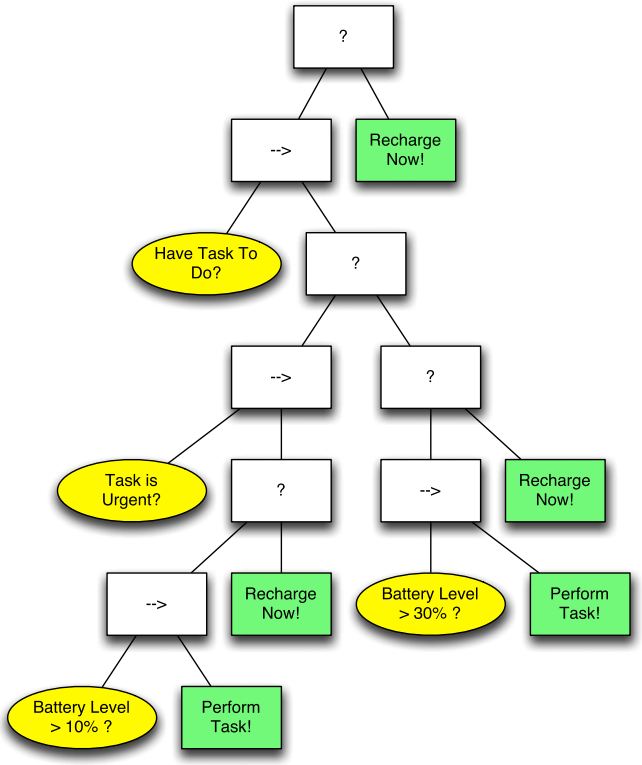

Fig. 21. BT that is equivalent to the decision tree in Fig. 19. A more compact version of the same tree can be found in Fig. 22.

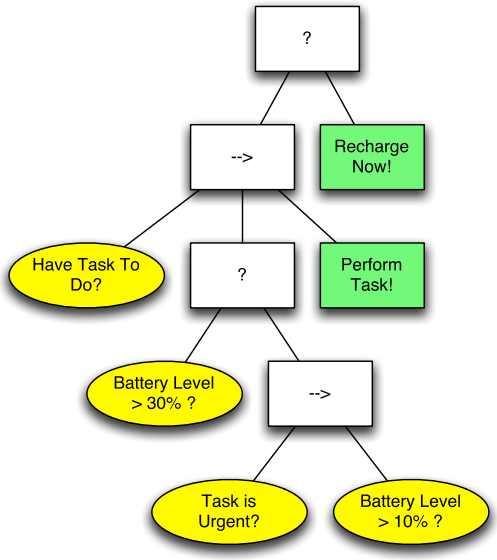

Fig. 22. More compact formulation of the BT in Fig. 21.

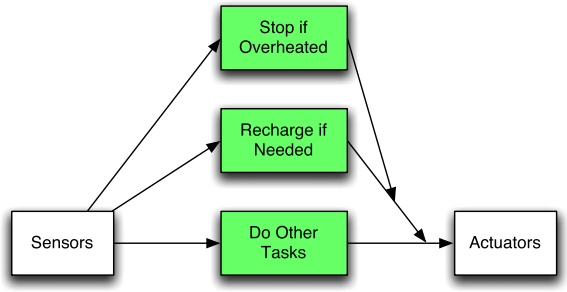

Fig. 23. Subsumption architecture. A higher level behavior can subsume (or suppress) a lower level one.

then be extended with safety or robustness features, as described in Section IV.

We finish this section with an example of how BTs generalize decision trees. Consider the decision tree in Fig. 19. Applying Lemma 7 we get the equivalent BT of Fig. 21. However, the direct mapping does not always take full advantage of the features of BTs. Thus a more compact, and still equivalent, BT can be found in Fig. 22, where again, we assume that all actions always return *Running*.

B. How BTs Generalize the Subsumption Architecture

In this section, we will see how the subsumption architecture, proposed by Brooks [19], can be realized by using a Fallback composition. The basic idea proposed in [19] was to have a number of controllers' setup in parallel and each controller was allowed to output both actuator commands, and a binary value, signaling if it wanted to control the robot or not. The controllers were then ordered according to some priority, and the highest priority controller, out of the ones signaling for action, was allowed to control the robot. Thus, a higher level controller was able to *subsume* the actions of a lower level one.

An example of a subsumption architecture can be found in Fig. 23. Here, the basic level controller *Do Other Tasks* is assumed to be controlling the robot for most of the time. However, when the battery level is low enough, the *Recharge if Needed* controller will signal that it needs to command the robot, subsume the lower level controller, and guide the robot toward the recharging station. Similarly, if there is risk for overheating, the top level controller *Stop if Overheated* will subsume both of the lower level ones, and stop the robot until it has cooled down.

Lemma 8: Given a subsumption architecture, we can create an equivalent BT by arranging the controllers as actions under a Fallback composition, in order from higher to lower priority. Furthermore, we let the return status of the actions be Failure if they do not need to execute, and Running if they do. They never return Success. Formally, a subsumption architecture composition $S_i(x) = \text{Sub}\left(S_{i1}(x), S_{i2}(x)\right)$ can be defined by

$$S_i(x) = \begin{cases} S_{i1}(x), & \text{if } S_{i1} \text{ needs to execute} \\ S_{i2}(x), & \text{else.} \end{cases} \tag{35}$$

Then, we write an equivalent BT as

$$\mathcal{T}_i = \text{Fallback}\left(\mathcal{T}_{i1}, \mathcal{T}_{i2}\right) \tag{36}$$

where $\mathcal{T}_{\text{ij}}$ is defined by $f_{\text{ij}}(x) = S_{\text{ij}}(x)$ and

$$r_{\text{ij}}(x) = \begin{cases} \mathcal{R}, & \text{if } S_{\text{ij}} \text{ needs to execute} \\ \mathcal{F}, & \text{else.} \end{cases} \tag{37}$$

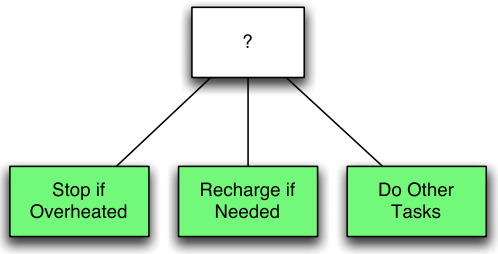

Fig. 24. BT version of the subsumption example in Fig. 23.

TABLE II
POSSIBLE OUTCOMES OF SUBSUMPTION-BT EXAMPLE

Stop if over heated	Recharge if Needed	Do Other Tasks	Executed Action
Running	Running	Running	Stop ...
Running	Running	Failure	Stop ...
Running	Failure	Running	Stop ...
Running	Failure	Failure	Stop ...
Failure	Running	Running	Recharge ...
Failure	Running	Failure	Recharge ...
Failure	Failure	Running	Do other ...
Failure	Failure	Failure	–

Proof: By the above arrangement, and Definition 3 we have that

$$f_i(x) = \begin{cases} f_{i1}(x), & \text{if } S_{i1} \text{ needs to execute} \\ f_{i2}(x), & \text{else} \end{cases} \tag{38}$$

which is equivalent to (35). In other words, actions will be checked in order of priority, until one that returns running is found. ■

A BT version of the example in Fig. 23 can be found in Fig. 24. The fact that the two control structures are equivalent is illustrated in Table II, where the executing action of all 2^3 possible return status combinations are listed. Note that no action is executed if all actions return Failure.

C. How BTs Generalize Sequential Behavior Compositions

In this section, we will see how the Fallback composition, and Lemma 3, can also be used to implement the sequential behavior compositions proposed in [20].

The basic idea proposed by Burridge *et al.* [20] is to extend the region of attraction by using a family of controllers, where the asymptotically stable equilibrium of each controller was either the goal state, or inside the region of attraction of another controller, positioned earlier in the sequence.

We will now describe the construction of [20] in some detail, and then see how this concept is captured in the BT framework. Given a family of controllers $U = \{\Phi_i\}$, we say that Φ_i *prepares* Φ_j if the goal $G(\Phi_i)$ is inside the domain $D(\Phi_j)$. Assume the overall goal is located at $G(\Phi_1)$. A set of execution regions $C(\Phi_i)$ for each controller was then calculated according to the following scheme:

1) Let a queue contain Φ_1. Let $C(\Phi_1) = D(\Phi_1)$, $N = 1$, $D_1 = D(\Phi_1)$.
2) Remove the first element of the queue and append all controllers that *prepare* it to the back of the queue.
3) Remove all elements in the queue that already has a defined $C(\Phi_i)$.
4) Let Φ_j be the first element in the queue. Let $C(\Phi_j) = D(\Phi_j) \setminus D_N$, $D_{N+1} = D_N \cup D(\Phi_j)$, and $N \leftarrow N + 1$.
5) Repeat steps 2–4 until the queue is empty.

The combined controller is then executed by finding j such that $x \in C(\Phi_j)$ and then by invoking controller Φ_j.

Looking at the design of the Fallback operator in BTs, it turns out that it does exactly the job of the Burridge algorithm above, as long as the subtrees of the Fallback are ordered in the same fashion as the queue above. We formalize this in Lemma 9.

Lemma 9: Given a set of controllers $U = \{\Phi_i\}$ we define the corresponding regions $S_i = G(\Phi_i), R'_i = D(\Phi_i), F_i = \text{Complement}(D(\Phi_i))$, and consider the controllers as atomic BTs, $\mathcal{T}_i = \Phi_i$. Assume S_1 is the overall goal region. Iteratively create a larger BT $\mathcal{T}_L$ as

1) Let $\mathcal{T}_L = \mathcal{T}_1$.
2) Find a BT $\mathcal{T}_* \in U$ such that $S_* \subset R'_L$
3) Let $\mathcal{T}_L \leftarrow \text{Fallback}\,(\mathcal{T}_L, \mathcal{T}_*)$
4) Let $U \leftarrow U \setminus \mathcal{T}_*$
5) Repeat steps 2, 3 and 4 until U is empty.

If all $\mathcal{T}_i$ are FTS, then so is $\mathcal{T}_L$.

Proof: The statement is a direct consequence of iteratively applying Lemma 3. ■

Thus, we see that BTs generalize the sequential behavior compositions of [20], with the execution region computations and controller switching replaced with the Fallback composition, as long as the ordering is given by Lemma 9.

VI. EXAMPLES

In this section, we will give three examples of how BTs can be used in robotics. The first example illustrates how the functional representation of Section III can be used to guarantee safety in terms of avoiding empty batteries. The second example illustrates how the functional representation can be used to increase robustness, in terms of increasing the region of attraction for a robot executing a task. Then, the third example illustrates the modularity of a larger BT by combining the two smaller examples with additional subtrees that add some additional robot capabilities.

All BTs were implemented by using our publicly available ROS BT package.[2] To illustrate the modularity, the leaf nodes are a mix of behaviors from the NAO Software Development Kit, such as *Stand Up*, *Sit Down*, and *Lie Down* and behaviors we developed ourselves, such as *Approach Ball*, *Grasp Ball*, and *Throw Ball*, see below.

Example 4 (Safety): To illustrate Lemma 5 we choose the BT of Fig. 25, which is actually a compact version of the BT of Fig. 5. The idea is that the first BT in the sequence is to guarantee that the combination does not run out of battery, under

[2]Library available at http://wiki.ros.org/behavior_tree.

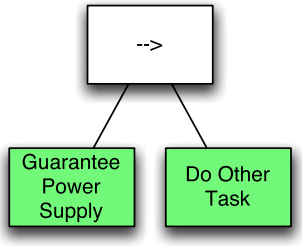

Fig. 25. BT where the first action guarantees that the combination does not run out of battery.

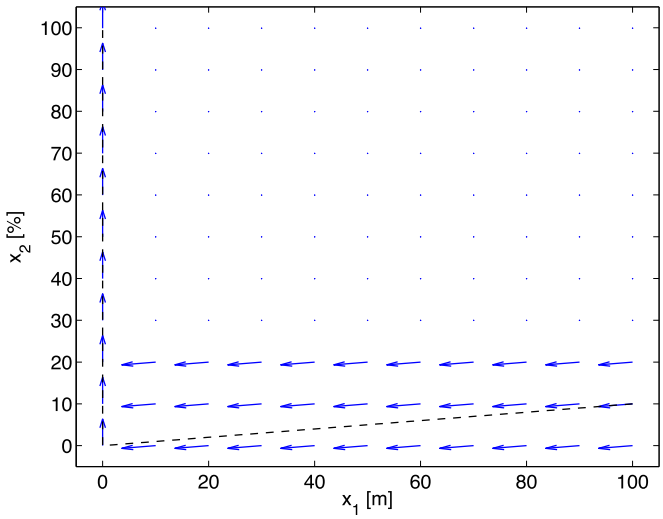

Fig. 26. *Guarantee Power Supply* Action.

very general assumptions about what is going on in the second BT.

First we describe the sets S_i, F_i, R_i and the corresponding vector fields of the functional representation. Then, we apply Lemma 5 to see that the combination does indeed guarantee against running out of batteries.

Let $\mathcal{T}_1$ be *Guarantee Power Supply* and $\mathcal{T}_2$ be *Do other tasks*. Let furthermore $x_k = (x_{1k}, x_{2k}) \in \mathbb{R}^2$, where $x_{1k} \in [0, 100]$ is the distance from the current position. to the recharging station and $x_{2k} \in [0, 100]$ is the battery level. For this example $\Delta t = 10\,\text{s}$.

For *Guarantee Power Supply*, $\mathcal{T}_1$, we have that

$$S_1 = \{x : 100 \leq x_2 \text{ or } (0.1 \leq x_1, 20 < x_2)\} \tag{39}$$

$$R_1 = \{x : x_2 \leq 20 \text{ or } (x_2 < 100 \text{ and } x_1 < 0.1)\} \tag{40}$$

$$F_1 = \emptyset \tag{41}$$

$$f_1(x) = \begin{pmatrix} x_1 \\ x_2 + 1 \end{pmatrix} \text{ if } x_1 < 0.1, x_2 < 100 \tag{42}$$

$$= \begin{pmatrix} x_1 - 1 \\ x_2 - 0.1 \end{pmatrix} \text{ else} \tag{43}$$

that is when running, the robot moves to $x_1 < 0.1$ and recharges. While moving, the battery level decreases and while charging the battery level increases. If at the recharge position, it returns Success only after reaching $x_2 \geq 100$. Outside of the recharge area, it returns Success as long as the battery level is above 20. A phase portrait of $f_1(x) - x$ is shown in Fig. 26

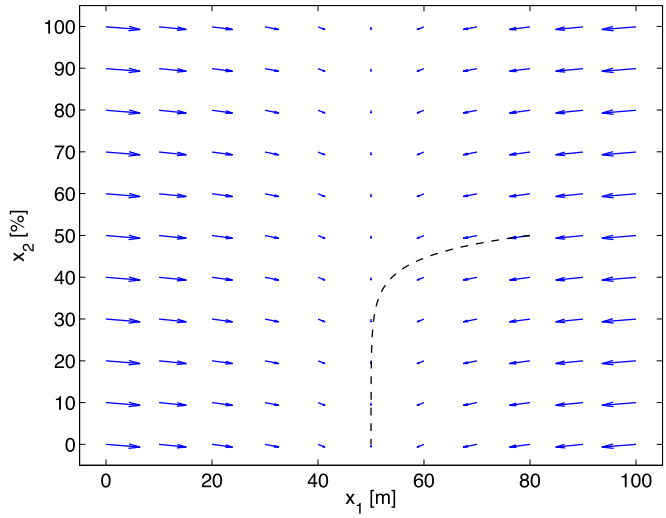

Fig. 27. *Do Other Task* Action.

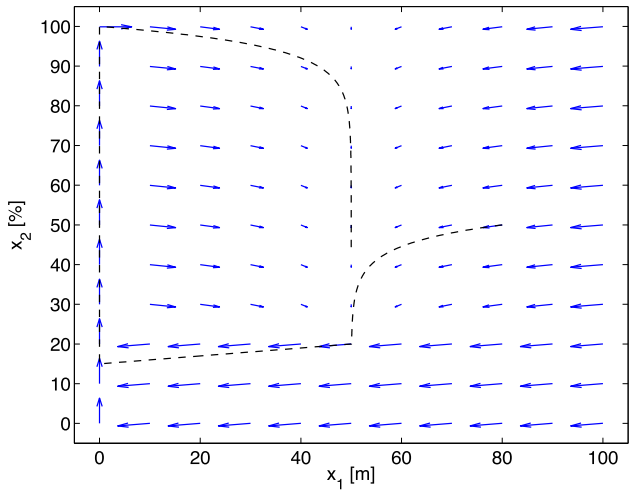

Fig. 28. Phase portrait of $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$. Note that $\mathcal{T}_1$ guarantees that the combination does not run out of battery. The dashed line is a simulated execution, starting at $(80, 50)$.

For *Do Other Task*, $\mathcal{T}_2$, we have that

$$S_2 = \emptyset \tag{44}$$

$$R_2 = \mathbb{R}^2 \tag{45}$$

$$F_2 = \emptyset \tag{46}$$

$$f_2(x) = \begin{pmatrix} x_1 + (50 - x_1)/50 \\ x_2 - 0.1 \end{pmatrix} \tag{47}$$

that is when running, the robot moves toward $x_1 = 50$ and does some important task, while the battery level keeps on decreasing. A phase portrait of $f_2(x) - x$ is shown in Fig. 27.

Given $\mathcal{T}_1$ and $\mathcal{T}_2$, the composition $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2)$ is created to improve the safety of $\mathcal{T}_2$, as described below.

Informally, we can look at the phase portrait in Fig. 28 to get a feeling for what is going on. The obstacle to be avoided is the Empty Battery state $O = \{x : x_2 = 0\}$, and $\mathcal{T}_0$ makes

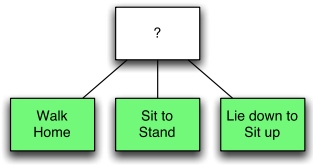

Fig. 29. Combination $\mathcal{T}_3 = \text{Fallback}(\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6)$ increases robustness by increasing the region of attraction.

sure that this state is never reached, since the *Guarantee Power Supply* action starts executing as soon as *Do Other Task* brings the battery level below 20%. The remaining battery level is also enough for the robot to move back to the recharging station, given that the robot position is limited by the reachable space, i.e., $x_{1k} \in [0, 100]$.

Formally, we state the following Lemma.

Lemma 10: Let the obstacle region be $O = \{x : x_2 = 0\}$ and the initialization region be $I = \{x : x_1 \in [0, 100], x_2 \geq 15\}$.

Furthermore, let $\mathcal{T}_1$ be given by (39)–(43) and $\mathcal{T}_2$ be an arbitrary BT satisfying $\max_x ||x - f_2(x)|| < d = 5$, then $\mathcal{T}_0$ = Sequence($\mathcal{T}_1, \mathcal{T}_2$) is Safe with respect to I and O, i.e., if $x(0) \in I$, then $x(t) \notin O$, for all $t > 0$.

Proof: First we see that $\mathcal{T}_1$ is Safe with respect to O and I. Then, we notice that $\mathcal{T}_1$ is Safeguarding with margin $d = 10$ for the reachable set $X = \{x : x_1 \in [0, 100], x_2 \in [0, 100]\}$. Finally, we conclude that $\mathcal{T}_0$ is Safe, according to Lemma 5. ■

Note that if we did not constraint the robot to move in some reachable set $X = \{x : x_1 \in [0, 100], x_2 \in [0, 100]\}$, it would be able to move so far away from the recharging station that the battery would not be sufficient to bring it back again before reaching $x_2 = 0$.

Example 5 (Robustness and efficiency): To illustrate Lemma 3 we look at the BT of Fig. 29 controlling an NAO robot. The BT has three actions *Walk Home*, which is first tried, if that fails (the robot cannot walk if it is not standing up) it tries the action *Sit to Stand*, and if that fails, it tries *Lie down to Sit Up*. Thus, each fallback action brings the system into the running region of the action to its left, e.g., the result of *Sit to Stand* is to enable the execution of *Walk Home*.

Let $x_k = (x_{1k}, x_{2k}) \in \mathbb{R}^2$, where $x_{1k} \in [0, 0.5]$ is the horizontal position of the robot head and $x_{2k} \in [0, 0.55]$ is vertical position (height above the floor) of the robot head. The objective of the robot is to get to the destination at $(0, 0.48)$.

First we describe the sets S_i, F_i, R_i and the corresponding vector fields of the functional representation. Then, we apply Lemma 3 to see that the combination does indeed improve robustness. For this example $\Delta t = 1$ s.

For *Walk Home*, $\mathcal{T}_4$, we have that

$$S_4 = \{x : x_1 \leq 0\} \tag{48}$$

$$R_4 = \{x : x_1 \neq 0, x_2 \geq 0.48\} \tag{49}$$

$$F_4 = \{x : x_1 \neq 0, x_2 < 0.48\} \tag{50}$$

$$f_4(x) = \begin{pmatrix} x_1 - 0.1 \\ x_2 \end{pmatrix} \tag{51}$$

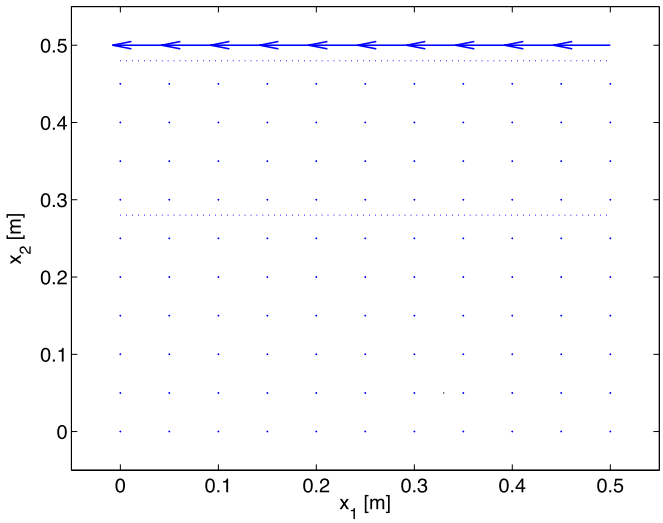

Fig. 30. Action *Walk Home*, keeps the head around $x_2 = 0.5$ and moves it toward the destination $x_1 = 0$.

that is it runs as long as the vertical position of the robot head, x_2, is at least 0.48 m above the floor, and moves toward the origin with a speed of 0.1 m/s. If the robot is not standing up $x_2 < 0.48$ m it returns Failure. A phase portrait of $f_4(x) - x$ is shown in Fig. 30. Note that $\mathcal{T}_4$ is FTS with the completion time bound $\tau_4 = 0.5/0.1 = 10$ and region of attraction $R'_4 = R_4$.

For *Sit to Stand*, $\mathcal{T}_5$, we have that

$$S_5 = \{x : 0.48 \leq x_2\} \tag{52}$$

$$R_5 = \{x : 0.3 \leq x_2 < 0.48\} \tag{53}$$

$$F_5 = \{x : x_2 < 0.3\} \tag{54}$$

$$f_5(x) = \begin{pmatrix} x_1 \\ x_2 + 0.05 \end{pmatrix} \tag{55}$$

that is it runs as long as the vertical position of the robot head, x_2, is in between 0.3 and 0.48 m above the floor. If $0.48 \leq x_2$ the robot is standing up, and it returns Success. If $x_2 \leq 0.3$ the robot is lying down, and it returns Failure. A phase portrait of $f_5(x) - x$ is shown in Fig. 31. Note that $\mathcal{T}_5$ is FTS with the completion time bound $\tau_5 = \text{ceil}\,(0.18/0.05) = \text{ceil}\,(3.6) = 4$ and region of attraction $R'_5 = R_5$.

For *Lie down to Sit Up*, $\mathcal{T}_6$, we have that

$$S_6 = \{x : 0.3 \leq x_2\} \tag{56}$$

$$R_6 = \{x : 0 \leq x_2 < 0.3\} \tag{57}$$

$$F_6 = \emptyset \tag{58}$$

$$f_6(x) = \begin{pmatrix} x_1 \\ x_2 + 0.03 \end{pmatrix} \tag{59}$$

that is it runs as long as the vertical position of the robot head, x_2, is below 0.3 m above the floor. If $0.3 \leq x_2$ the robot is sitting up (or standing up), and it returns Success. If $x_2 < 0.3$ the robot is lying down, and it returns Running. A phase portrait of $f_6(x) - x$ is shown in Fig. 32. Note that $\mathcal{T}_6$ is FTS with

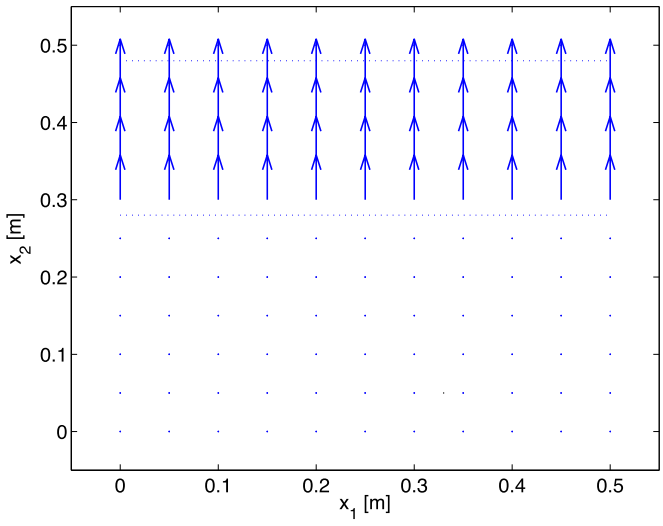

Fig. 31. Action *Sit to Stand* moves the head upwards in the vertical direction toward standing.

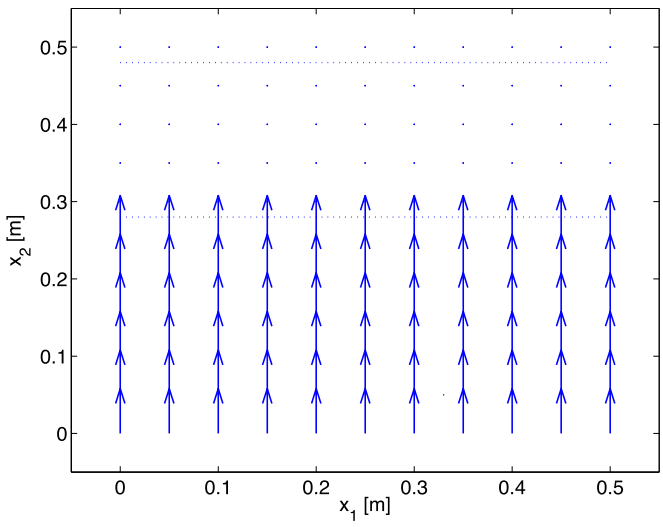

Fig. 32. Action *Lie down to Sit Up* moves the head upwards in the vertical direction toward sitting.

the completion time bound $\tau_6 = 0.3/0.03 = 10$ and region of attraction $R'_6 = R_6$.

Informally, we can look at the phase portrait in Fig. 33 to get a feeling for what is going on. As can be seen the fallbacks make sure that the robot gets on its feet and walks back, independently of where it started in $\{x : 0 < x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$.

Formally, we can use Lemma 3 to compute robustness in terms of the region of attraction R'_3, and efficiency in terms of bounds on completion time τ_3. The results are described in the following Lemma.

Lemma 11: Given $\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6$ defined in (48)–(59).

The combined BT $\mathcal{T}_3 = \text{Fallback}\,(\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6)$ is FTS, with region of attraction $R'_3 = \{x : 0 < x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$, completion time bound $\tau_3 = 24$.

Proof: We note that $\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6$ are FTS with $\tau_4 = 10$, $\tau_5 = 4$, $\tau_6 = 10$ and regions of attractions equal to the running regions $R'_i = R_i$. Thus, we have that $S_6 \subset R_5 = R'_5$ and $S_5 \subset R_4 = R'_4$. Applying Lemma 3 twice now gives the desired results, $R'_3 = R'_4 \cup R'_5 \cup R'_6 = \{x : 0 \leq x_1 \leq 0.5, 0 \leq x_2 \leq 0.55\}$ and $\tau_3 = \tau_4 + \tau_5 + \tau_6 = 10 + 4 + 10 = 24$.

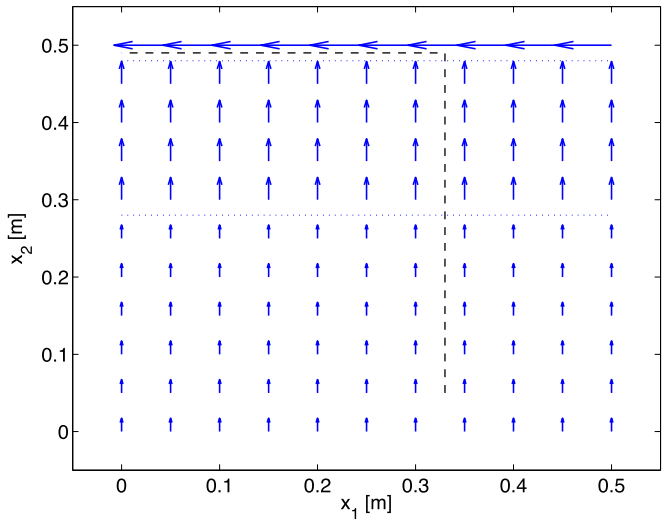

Fig. 33. Combination Fallback$(\mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6)$ first gets up, and then walks home.

We will use a larger BT below to illustrate modularity, as well as the applicability of the proposed analysis tools to more complex problems.

Example 6 (Big BT): The BT in Fig. 34 was designed for controlling an NAO humanoid robot in an interactive capability demo, and includes the BTs of Figs. 25 and 29 as subtrees, as discussed below.

The top left part of the tree includes some exception handling, in terms of battery management, and backing up and complaining in case the toe bumpers are pressed. The top right part of the tree is a parallel node, listening for new user commands, along with a request for such commands if none are given and an execution of the corresponding activities if a command has been received.

The subtree *Perform Activities* is composed of checking of what activity to do, and execution of the corresponding command. Since the activities are mutually exclusive, we let the Current Activity hold only the latest command and no ambiguities of control commands will occur.

The subtree *Play Ball Game* runs the ball tracker, in parallel with moving closer to the ball, grasping it, and throwing it.

As can be seen, the design is quite modular. An HDS implementation of the same functionality would need an extensive amount of transition arrows going in between the different actions.

We will now apply the analysis tools of the paper to the example, initially assuming that all atomic actions are FTS, as described in Definition 5.

Comparing Figs. 25 and 34 we see that they are identical, if we let *Do Other Task* correspond to the whole right part of the larger BT. Thus, according to Lemma 10, the complete BT is safe, i.e., it will not run out of batteries, as long as the reachable state space is bounded by 100 distance units from the

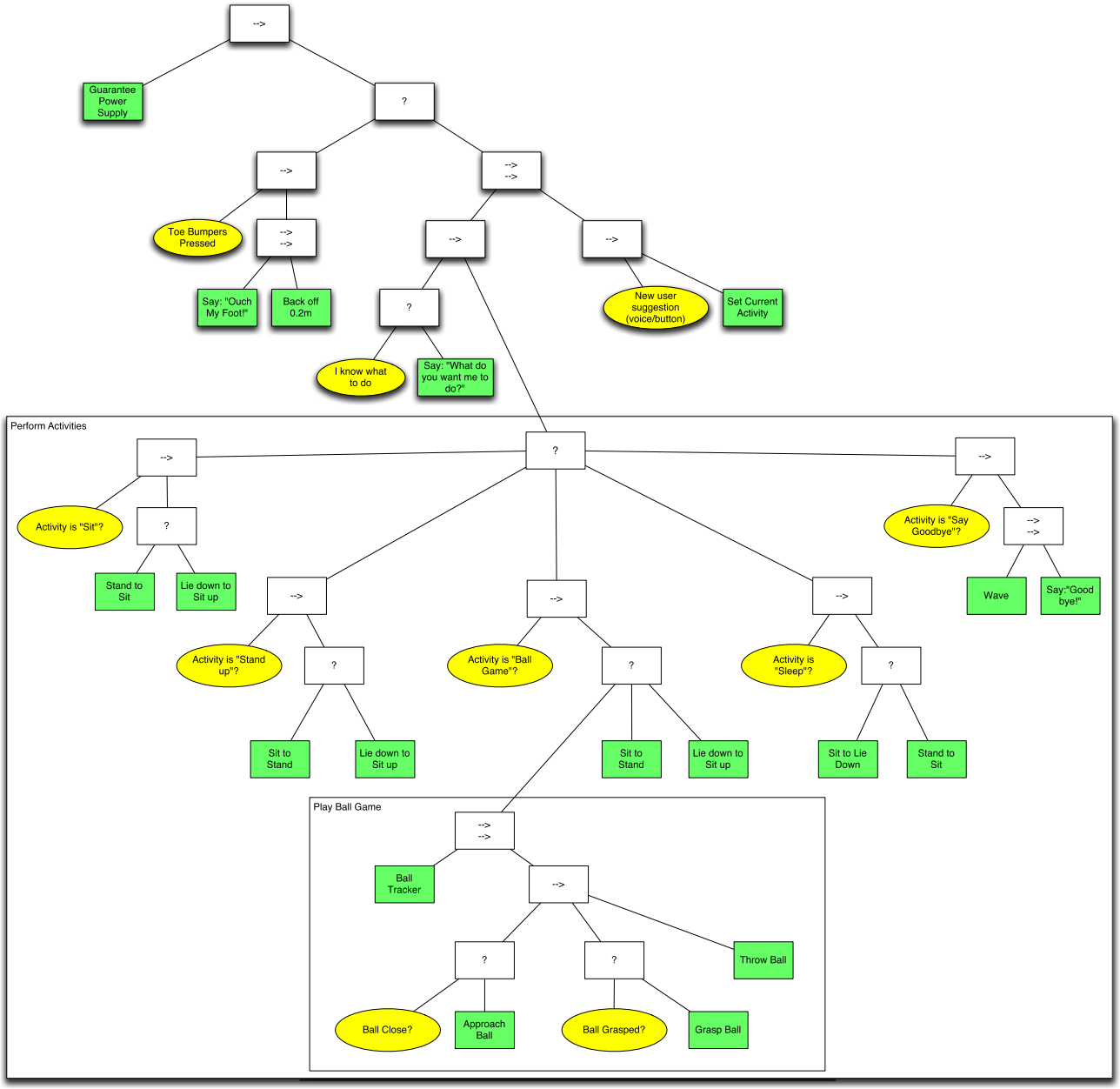

Fig. 34. BT that combines some capabilities of the NAO robot in an interactive and modular way. Note how atomic actions can easily be replaced by more complex sub-BTs.

recharging station and the time steps are small enough so that $\max_x ||x - f_2(x)|| < d = 5$, i.e., the battery does not decrease more than 5% in a single time step.

The design of the right subtree in *Play Ball Game* is made to satisfy Lemma 2, with the condition $S_1 = R_2' \cup S_2$. Let $\mathcal{T}_1 = \text{Fallback}(\textit{Ball Close?}, \textit{Approach Ball})$, $\mathcal{T}_2 = \text{Fallback}(\textit{Ball Grasped?}, \textit{Grasp Ball})$, $\mathcal{T}_3 = \textit{Throw Ball}$. Note that the use of condition action pairs makes the success regions explicit. Thus $S_1 = R_2' \cup S_2$, i.e., *Ball Close* is designed to describe the Region of Attraction of *Grasp Ball*, and $S_2 = R_3' \cup S_3$, i.e., *Ball Grasped* is designed to describe the Region of Attraction of *Throw Ball*. Finally, applying Lemma 2 twice we conclude that the right part of *Play Ball Game* is FTS with completion time bound $\tau_1 + \tau_2 + \tau_3$, region of attraction $R_1' \cup R_2' \cup R_3'$ and success region $S_1 \cap S_2 \cap S_3$.

The parallel composition at the top of *Play Ball Game* combines *Ball Tracker* which always returns Running, with the subtree discussed above. The parallel node has $M = 1$, i.e., it only needs the Success of one child to return Success. Thus, it is clear from Definition 4 that the whole BT *Play Ball Game* has the same properties regarding FTS as the right subtree.

Finally, we note that *Play Ball Game* fails if the robot is not standing up. Therefore, we improve the robustness of that subtree in a way similar to Example 5 in Fig. 29. Thus we create the composition Fallback(Play Ball Game, $\mathcal{T}_5$, $\mathcal{T}_6$), with $\mathcal{T}_5 =$ Sit to Stand, $\mathcal{T}_6 =$ Lie Down to Sit Up.

Assuming that that high dimensional dynamics of *Play Ball Game* is somehow captured in the x_1 dimension we can apply an argument similar to Lemma 11 to conclude that the combined BT is indeed also FTS with completion time bound $\tau_1 + \tau_2 + \tau_3 + \tau_5 + \tau_6$, region of attraction $R'_1 \cup R'_2 \cup R'_3 \cup R'_5 \cup R'_6$ and success region $S_1 \cap S_2 \cap S_3$.

The rest of the BT concerns user interaction and is thus not suitable for doing performance analysis.

Note that the assumption on all atomic actions being FTS is fairly strong. For example, the NAO grasping capabilities are somewhat unreliable. But we believe that a deterministic analysis such as this one is still useful for making good design choices. An analysis by using a stochastic approach, modeling the probabilities of success and failure, is also conceivable, but outside the scope of this paper.

VII. Conclusion

In this paper, we have provided a theoretical description of how properties such as efficiency, robustness, and safety are preserved in modular compositions of BTs, enabled by a new functional formulation of BTs.

It was shown that under certain circumstances, the composition of a particular class of BTs with a very general class can still be guaranteed to be safe. A result that is potentially useful in areas where there is a need to both guarantee key properties of a piece of software, and continuously adding functionality to that same software. Regarding robustness, it was shown how the region of attraction of a controller can be extended by the compositions of controllers.

The proposed analysis tools were illustrated by using two smaller and one more complex example, where safety and robustness of different action combinations were analyzed.

We have also shown how BTs generalize the important, but quite different concepts of decision trees, the subsumption architecture, and sequential behavioral compositions. As decision trees are an important tool in machine learning, this opens up the possibilities of learning BTs, while the results for the subsumption architecture and sequential behavior compositions are more useful for designing and analyzing robust robot controllers.

All examples were implemented by using our publicly available ROS BT implementation,[3] and a combination of atomic actions created by others and ourselves.

To conclude, we believe that the strength of BTs lie in their modularity, and that BTs can complement FSM in robotic software development, much like one programming language can complement another.

Acknowledgment

The authors would like to thank Prof. M. Egerstedt for his valuable input into this paper.

[3]Library available at http://wiki.ros.org/behavior_tree.

References

[1] D. Isla, "Handling complexity in the Halo 2 AI," in *Proc. Game Developers Conf.*, 2005.

[2] A. Champandard, "Understanding behavior trees." 2007. [Online]. Available: AiGameDev.com.

[3] D. Isla, "Halo 3-building a better battle," in *Proc. Game Developers Conf.*, 2008.

[4] I. Millington and J. Funge, *Artificial Intelligence for Games*. Boca Raton, FL, USA: CRC Press, 2009.

[5] S. Rabin, *The behavior tree starter kit, in Game AI Pro*. Boca Raton, FL, USA: CRC Press, 2014, ch. 6, pp. 73–91.

[6] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[7] C. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game DEFCON," in *Applications of Evolutionary Computation*. Berlin, Germany: Springer, 2010, pp. 100–110.

[8] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," *IEEE Trans. Comput. Intell. AI Games*. [Online.] Available: http://ieeexplore.ieee.org/document/7435292/

[9] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler, "Parameterizing behavior trees," in *Motion in Games*. Berlin, Germany: Springer, 2011, pp. 144–155.

[10] I. Bojic, T. Lipic, M. Kusek, and G. Jezic, "Extending the JADE agent behaviour model with JBehaviourtrees framework," in *Agent and Multi-Agent Systems: Technologies and Applications*. Berlin, Germany: Springer, 2011, pp. 159–168.

[11] P. Ögren, "Increasing modularity of UAV control systems using computer game behavior trees," in *Proc. AIAA Guid., Navig. Control Conf.*, Minneapolis, MN, USA, 2012-4458.

[12] J. A. D. Bagnell *et al.*, "An integrated system for autonomous robotics manipulation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2012, pp. 2955–2962.

[13] A. Klökner, "Interfacing behavior trees with the world using description logic," in *Proc. AIAA Conf. Guid. Navig. Control*, Boston, MA, USA, 2013-4636.

[14] M. Colledanchise, A. Marzinotto, and P. Ögren, "Performance analysis of stochastic behavior trees," in *Proc. IEEE Int. Conf. Robot. Autom.*, Jun. 2014, pp. 3265–3272.

[15] T. Galluzzo, M. Kazemi, and J.-S. Valois, "Bart—behavior architecture for robotic tasks," Tech. Rep., 2013. [Online]. Available: https://code.google.com/archive/p/bart

[16] D. Hu, Y. Gong, B. Hannaford, and E. J. Seibel, "Semi-autonomous simulated brain tumor ablation with Raven II surgical robot using behavior tree," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2015, pp. 3868–3875.

[17] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, "A framework for end-user instruction of a robot assistant for manufacturing," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2015, pp. 6167–6174.

[18] A. Klöckner, F. van der Linden, and D. Zimmer, "The modelica behaviortrees library: Mission planning in continuous-time for unmanned aircraft," in *Proc. 10th Int. Modelica Conf.*, 2014, pp. 727–736.

[19] R. Brooks, "A robust layered control system for a mobile robot," *IEEE J. Robot. Autom.*, vol. 2, no. 1, pp. 14–23, Mar. 1986.

[20] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, "Sequential composition of dynamically dexterous robot behaviors," *Int. J. Robot. Res.*, vol. 18, no. 6, pp. 534–555, 1999.

[21] C. Sammut, S. Hurst, D. Kedzier, and D. Michie, "Learning to fly," in *Imitation in Animals and Artifacts*. Cambridge, MA, USA: MIT Press, 2002, pp. 171–190.

[22] J. Le Ny and G. J. Pappas, "Sequential composition of robust controller specifications," in *IEEE Int. Conf. Robot. Autom.*, 2012, pp. 5190–5195.

[23] M. Colledanchise and P. Ögren, "How behavior trees modularize robustness and safety in hybrid systems," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Jun. 2014, pp. 1482–1488.

[24] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, pp. 147–148, Mar. 1968. [Online]. Available: http://doi.acm.org/10.1145/362929.362947

[25] A. Filippov and F. Arscott, *Differential Equations With Discontinuous Righthand Sides: Control Systems (Mathematics and its Applications Series)*. Norwell, MA, USA: Kluwer, 1988. [Online]. Available: http://books.google.se/books?id=KBDyZSwpQpQC

Michele Colledanchise (S'14) received the B.S. and M.S. degrees in automatic control from Polytechnic University of Marche, Ancona, Italy, in 2010 and 2012, respectively. He is currently working toward the Ph.D. degree in robotics with Royal Institute of Technology (KTH), Stockholm, Sweden.

His research interests include control systems, control architectures, and automated planning, with a strong focus in robotic applications.

Petter Ögren (M'04) was born in Stockholm, Sweden, in 1974. He received the M.S. degree in engineering physics and the Ph.D. degree in applied mathematics from Royal Institute of Technology (KTH), Stockholm, Sweden, in 1998 and 2003, respectively.

In the fall of 2001, he visited the Mechanical Engineering Department, Princeton University, Princeton, NJ, USA. From 2003 to 2012, he was a Senior Scientist and Deputy Research Director in Autonomous Systems in the Swedish Defence Research Agency (FOI). He is currently an Associate Professor in the Computer Vision and Active Perception Laboratory, KTH. His research interests include robot control system architectures, teleoperation, navigation and obstacle avoidance, and multiagent coordination.