Behavior Trees in Robot Control Systems

Petter Ögren,¹ and Christopher I. Sprague¹

¹KTH Royal Institute of Technology, Stockholm, Sweden, SE-10044; email: petter,sprague@kth.se

Keywords
behavior trees, modularity, hierarchical modularity, transparency, robustness, autonomous system, feedback, task switching

Abstract
In this paper we will give a control theoretic perspective on the research area of behavior trees in robotics. The key idea underlying behavior trees is to make use of modularity, hierarchies and feedback, in order to handle the complexity of a versatile robot control system. Modularity is a well-known tool to handle software complexity by enabling development, debugging and extension of separate modules without having detailed knowledge of the entire system. A hierarchy of such modules is natural, since robot tasks can often be decomposed into a hierarchy of sub-tasks. Finally, feedback control is a fundamental tool for handling uncertainties and disturbances in any low level control system, but in order to enable feedback control on the higher level, where one module decides what submodule to execute, information regarding progress and applicability of each submodule needs to be shared in the module interfaces. We will describe how these three concepts come to use in theoretical analysis, practical design, as well as extensions and combinations with other ideas from control theory and robotics.
1. Introduction

In this section we will describe why modularity, hierarchical structure, and feedback are useful in robot control systems, and how these three concepts are combined in a control structure called behavior trees (BTs).

The rapid development of robotic hardware and software has enabled the use of robots to expand beyond structured factory environments, into our homes, streets and diverse workplaces. In these new settings, robots often need a wide range of capabilities, including the possibility to add even more features by online software updates. It is well known that adding features to software increases complexity, which in turn increases the cost of development (1). It is also well known that modularity is a key principle that can be used to reduce complexity. By dividing a system into modules with well defined interfaces and functionality, each module can be developed, tested, and extended without having detailed knowledge about the rest of the system. Thus, there is reason to believe that modularity, in terms of well defined interfaces and functionality, is an important property also for a robot control system.

A natural extension of modularity is hierarchical modularity (1), where modules can contain sub-modules and so on. The rationale for such a structure is the simple observation that when a system grows, a single layer of modules either results in a very large number of modules, or in modules that are themselves very large. Thus, the benefit of modularity is strengthened if the modules can contain submodules in a hierarchical fashion. There is an additional reason why hierarchical modularity makes sense in robot control systems, and this is the fact that many robot tasks can naturally be divided into subtasks in a hierarchical way, an observation that is underlying e.g. hierarchical task networks (2,3). For example, fetching an item might involve moving to a cupboard and opening it, which might in turn involve grasping a handle, and so on.

To make the control system modular, we will make the actual control policy, the mapping from state to action, modular. In many applications within robotics and control there is a need to compose a control policy out of a set of subpolicies. In an autonomous car there might be subpolicies for parking, overtaking, lane keeping, handling intersections etc, and in a mobile manipulator there might
Figure 1: A mobile manipulator BT. The top level goals can be found in the top row: Make sure in safe area (1), Make sure in range of charger (6), Make sure object at goal (9) and Make sure robot at charger (37), in order of priority. If, at some time instant the action, Move to object (26), is executing, a human operator can easily understand why this action was chosen by reading every double stroked module on the branch towards the root: Move to object (26), in order to Make sure robot near object (22), (to) Make sure object in gripper (12), (to) Make sure object at goal(9). The meaning of the double strokes will be explained in detail in Section 6.

be subpolicies for grasping, docking with a recharger, moving from A to B etc.

Feedback is perhaps the most important principle in control theory, and the property that separates open loop control from closed loop control. In open loop control, a series of commands are executed
over time according to some form of plan, while in closed loop control, the issued commands are constantly adapted based on current information obtained from monitoring key parts of the world state. It is clear that classical closed loop control should be executed at the lowest level of a hierarchical modular robot control system, but it is less clear what kind of observations should be used between two hierarchical levels, to allow one module to use feedback when it determines what submodule to execute. We will come back to this question shortly, but for now we just note that if a submodule fails with achieving its goal, we do not want the parent module to just execute the next submodule in an open loop fashion, but instead chose the proper submodule using feedback, based on the fact that the previous one just failed.

BTs were created to combine feedback with a hierarchical modular design. Thus, modules should capture some functionality that can be combined into larger modules, with a clearly defined interface between modules on all levels. Furthermore, feedback regarding the execution should be passed up the module hierarchy using the same interface.

The formal definition of BTs can be found in Section 3, but here we will make an informal description. We let each module in the discussion above be a BT. Thus a complex BT can contain a number of sub-BTs and so on, as illustrated in Figure 1 where each node in the graph is the root of a sub-BT. The interface of all BTs (the lines in the figure) is given in terms of a function call with return values. When a BT is called it returns two things, first the suggested control action and second the information needed to apply feedback control and determine what sub-BT to execute. This information, or metadata, regarding the execution and applicability of a module is given in terms of one out of three symbols, S (success), F (failure), and R (running). Thus, if a submodule for grasping a cup returns success, the next submodule in the intended sequence, such as lifting the cup, might be invoked. If, on the other hand, the submodule returns failure, some kind of fallback action needs to be invoked, such as trying to re-grasp the cup, or getting a better sensor reading of its pose. Finally, if the submodule returns running it might be preferable to let the execution run for a while longer.

At this point we note that there are basically three fundamental reasons for stopping what you are doing and starting a new activity. Either you succeed and go on to the next action, or you fail, and need to handle this more or less unexpected fact, or an external event happened that makes the current action inappropriate. Imagine a robot tasked to fetch an object, such as in Figure 1. If the robot is grasping the object, it might switch to moving if the grasping succeeds. If the grasping fails, it might try with the other arm. A number of events might also occur to end the process of grasping the object. Another agent might put the object in its proper place (positive surprise, we are done), or, another agent might move the object further away (negative surprise, we need to move closer again), or the fire alarm might go off making the entire building unsafe (unrelated surprise, we need to leave the building).

The outline of this paper is as follows. In Section 2 we give a brief history of BTs, followed by a formal definition in Section 3. Then, we investigate the property of modularity in some more detail in Section 4. The issue of convergence is analyzed in Section 5 followed by a design principle in Section 6. Safety guarantees and the connection to control barrier functions are treated in Section 7. Then, we see how BTs are related to explainable AI in Section 8 and can be connected to reinforcement learning in Section 9, evolutionary algorithms in Section 10 and planning in Section 11. Finally, conclusions, together with a set of summary points and future important issues can be found in Section 12.

2. The history of behavior trees and their relation to finite state machines

The need for a modular hierarchical control structures is shared between the domains of robotics and computer games. However, low-level capabilities such as grasping and navigation are research areas
in their own right in robotics, but trivial in the virtual worlds of a computer game. Therefore, computer
game programmers started putting together larger sets of low level capabilities earlier than roboticists,
and hence experienced the drawback of finite state machines (FSMs) described below earlier as well.
BTs were thus proposed as a response to those drawbacks by programmers in the gaming industry. It is
hard to determine who was the first to conceive BTs, as important ideas were shared in partially docu-
mented workshops, conferences and blog posts. However, important milestones were definitely passed
through the work of Michael Mateas and Andrew Stern (4), and Damian Isla (5). The development
continued in the game AI community, and a few years later the first journal paper on BTs appeared
(6), followed by the first papers on BTs in robotics, independently described in (7) and (8). Note that
there is also a completely different tool called behavior trees, that is used for requirement analysis.

As mentioned above, BTs were partially developed to improve modularity of (FSM) controllers.
FSMs, and in particular hierarchical FSMs (9), do have mechanisms for hierarchical modularity. How-
ever, a key problem is that the transitions of a FSM are encoded inside the modules (states), thus each
module needs to know about the existence and capabilities of the other modules, as well as the purpose
of its own supermodule. In this way, each transition creates a dependence between two modules, and
with N modules there is N² possible transitions/dependencies. In comparison, a BT module only has
to know if it succeeded or not. Regarding expressivity, it was shown in (10) that BTs with internal vari-
ables are equally expressive as FSMs. Thus, like with two general purpose programming languages,
the choice between the two is not governed by what is possible, but rather what makes the design
process smooth. A more detailed description of the relationship between BTs and FSM, as well as a
broad overview of research on BTs can be found in the recent survey (11) and the book (12).

3. Definition of Behavior Trees

In this section we will formally define BTs and their execution for both discrete and continuous-time
systems. This formulation is based on (13, 14, 15) and chosen to enable a control-theoretic analysis of
BT.

The core idea is to formally define a BT as a combination of a controller and a metadata function
used to provide feedback regarding the execution. Using the metadata, these BTs can then be combined
in order to create more complex BTs in a hierarchical tree structure, as in Figure 1, hence the name
behavior tree.

Let the system state be \( x \in X \subset \mathbb{R}^n \) and the system dynamics be given by \( \dot{x} = f(x,u) \) or \( x_{t+1} = f(x_t,u) \), see Definition 2 with \( u \in U \subset \mathbb{R}^k \).

**Definition 1 (Behavior tree).** A BT \( \mathcal{T} \) is a pair

\[
\mathcal{T} = (u_i, r_i)
\]

where \( i \) is an index, \( u_i : X \to U \) is the controller that runs when the BT is executing, and \( r_i : X \to \{R, S, F\} \) provides metadata regarding the applicability and progress of the execution.

A BT can either be created through a hierarchical combination of other BTs, using the Sequence
and Fallback operators described below, or it can be defined by directly specifying \( u_i(x) \) and \( r_i(x) \).

---

1. A different concept with the same name: [https://en.wikipedia.org/wiki/Behavior_tree](https://en.wikipedia.org/wiki/Behavior_tree)

2. Other BT formulations exist, including memory versions of interior nodes, and leaf nodes encapsulating the
execution of the system dynamics, thereby allowing the parallel execution of two leaves e.g., controlling different
motors on the same robot, see (13, 16).
Figure 2: The state space $X$ is partitioned into a set of operating regions $\Omega_i$, and the global success region $S_0$ and failure region $F_0$. We want to design the BT such that the state reaches $S_0$ and avoids $F_0$. The solid line illustrates an execution starting in $\Omega_4$ and ending in $S_0$. Note that the $\Omega_i$ of each subtree is not defined in the subtree itself, but depends on feedback in the form of the return status $r_i(x)$ of a number of neighboring subtrees as described in Definition 11.

The metadata $r_i$ is interpreted as follows: Running ($\mathcal{R}$), Success ($\mathcal{S}$), and Failure ($\mathcal{F}$). Let the Running region ($R_i$), Success region ($S_i$) and Failure region ($F_i$) correspond to a partitioning of the state space, defined as follows:

$$R_i = \{x \in X : r_i(x) = \mathcal{R}\}, \quad S_i = \{x \in X : r_i(x) = \mathcal{S}\}, \quad F_i = \{x \in X : r_i(x) = \mathcal{F}\}.$$

**Definition 2.** Assuming the BT $\mathcal{T}_i$ is the root, and not a subtree of another BT, and $x \in R_i$, the system evolves according to $\dot{x} = f(x, u_i(x))$ or $x_{t+1} = f(x_t, u_i(x_t))$ depending on if the system is continuous time or discrete time.

**Remark 1.** If $x \in S_i \cup F_i$ of the root BT, it has either succeeded or failed, and it is up to the user to apply an appropriate action, such as shutting down the robot or entering an idle mode. If this is not desired, an additional top layer of the BT can be designed that executes when the main BT returns success or failure. If this top layer always return running, we have $S_i = F_i = \emptyset$ for the overall tree.

The execution of a BT can thus be seen as a discontinuous dynamical system \[17\], as illustrated in Figure 2. Below, in Lemma 8 we will show that if the BT $\mathcal{T}_0$ is composed of a set of subtrees $\{\mathcal{T}_i\}$ then the state space $X$ is divided into different so-called operating regions $\Omega_i$ such that $u_0(x) = u_i(x)$ if $x \in \Omega_i$, as illustrated in Figure 2. Thus, in the continuous time case above, the execution will in most cases be a discontinuous dynamical system with corresponding issues regarding existence and uniqueness \[18\]. Going deeper into these issues is beyond the scope of this paper, therefore we will just make the following assumption.

**Assumption 1.** The BTs $\mathcal{T}_i$ are defined in such a way that the execution in Definition 2 has solutions that exist and are unique.

As described above, the main point of BTs is to enable the creation of complex controllers from simpler ones in a modular fashion. There are two ways of combining BTs, the sequence and fallback compositions.

**Definition 3.** (Sequence Compositions of BTs) Two or more BTs can be composed into a more com-
plex BT using a Sequence operator, \( \mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2) \). Then \( r_0, u_0 \) are defined as follows

\[
\begin{align*}
\text{If } x \in S_1 : & \quad r_0(x) = r_2(x), \quad u_0(x) = u_2(x) \\
\text{else : } & \quad r_0(x) = r_1(x), \quad u_0(x) = u_1(x)
\end{align*}
\]

\( \mathcal{T}_1 \) and \( \mathcal{T}_2 \) are called children of \( \mathcal{T}_0 \). Note that when executing \( \mathcal{T}_0 \), the first child \( \mathcal{T}_1 \) in (3) is executed as long as it returns \text{Running} or \text{Failure} (\( x_k \notin S_1 \)). The second child of the Sequence is executed in (2), only when the first returns \text{Success} (\( x_k \in S_1 \)). Finally, the Sequence itself, \( \mathcal{T}_0 \) returns \text{Success} only when all children have succeeded (\( x \in S_1 \cap S_2 \)).

For notational convenience, we write

\[
\text{Sequence}(\mathcal{T}_1, \text{Sequence}(\mathcal{T}_2, \mathcal{T}_3)) = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3),
\]

and similarly for arbitrarily long compositions. The sequence node is also denoted by \( \rightarrow \), as seen in Figure 1.

**Remark 2** (Giving names to sequence and fallback nodes). When drawing BTs, as in Figure 1, the symbols \( \rightarrow \) and \( ? \) are used to denote sequences and fallbacks. However, some users prefer to also give descriptive names to the subtrees starting from each node, to improve readability. We believe this is a useful practice, similar to choosing good names for functions when programming. Giving all nodes names improves readability, underlines the fact that all subtrees, including single leaf nodes, have an identical interface to its parent, see Definition 7, and is very convenient in combination with software GUIs that enable a subtree to be visually collapsed into a single node and expanded back again.

The advantage of properly named subtrees can be seen in Figure 1. As described in the caption, the reason for executing a leaf node is clear from the names of the subtrees it belongs to. This is discussed further in Section 8 on explainable AI.

A key element of BTs is how the operating regions \( \Omega_i \) of Figure 2 depend on the success, failure and running regions, \( S_i, F_i, R_i \) of all subtrees across a hierarchical structure. Thus we need to determine a number of properties of these sets.

**Lemma 1.** If \( \mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2) \), then Definition 3 implies that

\[
\begin{align*}
S_0 &= S_1 \cap S_2, \\
F_0 &= F_1 \cup (S_1 \cap F_2), \\
R_0 &= R_1 \cup (S_1 \cap R_2)
\end{align*}
\]

**Proof.** A straightforward application of the definition gives the result above.
Definition 4. (Fallback Compositions of BTs) Two or more BTs can be composed into a more complex BT using a Fallback operator, $T_0 = \text{Fallback}(T_1, T_2)$. Then $r_0, u_0$ are defined as follows

$$
\text{If } x \in F_1 : \quad r_0(x) = r_2(x), \quad u_0(x) = u_2(x)
$$

$$
\text{else} : \quad r_0(x) = r_1(x), \quad u_0(x) = u_1(x)
$$

Figure 3: This mobile manipulator BT is a subset of the one in Figure 1, with the overall objective of moving a given object to a designated goal area, while staying in the safe part of the working area. All nodes are numbered with the index of the corresponding subtree. The sequence nodes 0, 3, 11, 34 are denoted by a $(\rightarrow)$ symbol and the fallback nodes 1, 9 by a $(\text{?})$. Conditions are indicated by ovals. Note that the children of nodes 3, 11, 34 are not shown in this figure.

Lemma 2. If $T_0 = \text{Fallback}(T_1, T_2)$, then Definition 4 implies that

$$
S_0 = S_1 \bigcup (F_1 \cap S_2),
$$

$$
F_0 = F_1 \cap F_2,
$$

$$
R_0 = R_1 \bigcup (F_1 \cap R_2),
$$

Proof. A straightforward application of the definition gives the result above.

Definition 5 (Condition). If a BT $T_i$ is such that $R_i = \emptyset$ we call it a Condition. Being a BT, it still has $u_i$ defined, but as we will see in Lemma 8 below, that control will not be executed.

Consider again the Mobile Manipulator example in Figure 3. Make sure in safe area (1) is a fallback composition of nodes 2, 3. Equation (8) now states that node 1 executes $\text{Move to safe area if possible (3)}$, $u_1(x) = u_3(x)$, only when $\text{In Safe Area (2)}$ returns failure, $x \in F_2$. Furthermore, node 2 is a condition, $R_2 = \emptyset$, thus if $x \notin F_2$ we have $x \in S_2$ and success will be returned by node 1 up to
node 0 which would then execute Make sure object at goal (9) and so on. Furthermore, Equation (12) indicates that the only way for node 1 to fail is if both node 2 and node 3 fails. That is Make sure in safe area (1) only returns failure if both In safe area (2) and Move to safe area if possible (3) return failure.

Now we have all we need to create and execute BTs. In the next section we will explore the modularity of BTs, and then analyze under what circumstances the execution will converge to the success region.

4. Optimal Modularity

One of the key advantages of BTs is their modularity, a property made possible by the fact that all subtrees on all levels of a BT have the same interface, given by Definition 1. However, as was shown in (19), a deeper analysis can be made, by extending a measure of modularity/complexity used in graph theory. In this section we give a very brief overview of the key theoretical results, showing that BTs have a so-called cyclomatic complexity of one, a fact that makes them optimally modular, within a particular class of control structures.

To investigate the concept of modularity in general reactive control architectures, so-called Decision Structures (DS) were defined in (19). These are directed graphs, as illustrated in the upper left part of Figure 4. Each node in this structure corresponds to a controller $u_i$, and each edge label corresponds to a return status $r_i$ that can be returned by the node the arc is leaving. The DS is executed in the following way: starting at the source node, (a) in Figure 4 (top left), look at the return status $r_i(x)$ of that node, and follow the edge (if there is one) with a label corresponding to $r_i(x)$. Similarly, the

Figure 4: A BT (top right) with its corresponding DS (top left), and the module decomposition of the DS (bottom). Note how all graphs in the decomposition correspond to paths without cycles. By Theorem 2 all BTs will give rise to DSs with such non-cyclic graphs, a fact that results in all BTs having a cyclomatic complexity of one.
return status of the new node is checked and the corresponding edge is followed until you find a \( r_i(x) \) without a corresponding outgoing edge label. Then this controller \( u_i \) is chosen. This process is then constantly iterated from the source to keep track of the proper controller to run.

Given the above it can be seen in Figure 4 that the execution of the DS on the upper left is identical to the execution of the BT in the upper right. Thus, this DS is equivalent to the BT. In fact, DSs are generalizations of BTs in the sense that all BTs can be written as a DS, but not all DSs can be written as BTs. DSs are fairly general, as the label set can be of any size, as long as it is finite. Thus a DS is very similar to a FSM, with the difference that a DS constantly starts from the source in order to determine what controller to execute.

Using inspiration from modules in graph theory \(^{20}\), the authors of \(^{19}\) define modules in DS as follows.

**Definition 6** (Definition 6.3 in \(^{19}\), Modules in decision structures). *Let \( Z \) be a decision structure. Let \( Y \subset N(Z) \) be a subset of the nodes where \( Z[Y] \) is also a decision structure. We say \( Y \) is a module if for every node \( v \in N(Z) \setminus Y \), any arc from \( v \) into \( Y \) goes to \( Y \)'s source, and if there is an arc labelled \( r \) out of \( Y \) to \( v \), then for every \( y \in Y \) the \( r \) out of \( y \) exists and goes either to \( v \) or to another element of \( Y \).*

After that they define quotient DS, where the modules are collapsed into single nodes.

**Lemma 3** (Lemma 6.8 in \(^{19}\)). *Let \( Z \) be a decision structure and \( P \) a modular partition. Then the quotient \( Z/P \) is also a decision structure. Moreover, if \( P \) is maximal then \( Z/P \) is prime.*

Given these, a maximal module decomposition is defined (the interested reader is referred to \(^{19}\)) leading to the following theorem.

**Theorem 1** (Theorem 6.15 in \(^{19}\)). *Let \( Z \) be a decision structure with \( k \) distinct arc labels. Then \( Z \) is structurally equivalent to a \( k \)-BT if and only if every quotient graph in \( Z \)'s module decomposition is a path.*

**Proof.** See \(^{19}\). \(\square\)

Here, a \( k \)-BT is a generalization of BTs having a label set of size \( k \), with 2-BTs corresponding to normal BTs, counting the labels success and failure, but not running since it does not lead to a transition in the DS. The \( k \)-BT also has \( k \) different interior nodes, a generalization of the two nodes Sequence and Fallback, defined for 2-BTs.

This theorem is clearly illustrated in Figure 4 where we can see that there are no cycles in the module decomposition of the DS. The number of cycles has been shown to be correlated with the difficulty of testing and debugging a piece of code \(^{21}\). Therefore, the concept of cyclomatic complexity has been defined for graphs and the authors of \(^{19}\) extend it to DSs as follows:

**Definition 7** (Definition 6.19 in \(^{19}\)). *Let \( Z \) be a decision structure. The cyclomatic complexity of \( Z \) is the number of linearly independent undirected cycles in \( Z \), plus one.*

Then, the concept is extended to account for modularity.

**Definition 8** (Definition 6.21 in \(^{19}\)). *Let \( Z \) be a decision structure. The essential complexity of \( Z \) is the maximum cyclomatic complexity of any quotient graph in its module decomposition.*

Finally they prove the following theorem.

**Theorem 2** (Theorem 6.23 in \(^{19}\)). *Let \( Z \) be a decision structure with \( k \) distinct edge labels. \( Z \) is equivalent to a \( k \)-BT if and only if it has essential complexity 1.*

---

\(^{10}\) Ögren and Sprague
Proof. See [19].

Looking at the case of $k = 2$ (two distinct edge labels, $S$ and $F$), the theorem says that BTs are exactly the DSs with essential complexity $1$. Thus, BTs correspond to the class of optimally modular DSs.

5. Proving convergence

Many control problems are formulated in terms of making some equilibrium point stable, such that a wide set of state trajectories starting from different states all converge to that equilibrium point. For BTs we do not pick some particular equilibrium point, but instead assume that the success region of the root, $S_0$ is chosen to capture the set of desired outcomes, and therefore we also assume that the design objective is to make a large set of state trajectories converge to points inside $S_0$. Thus, in this section we will study the problem of when we can guarantee that the state will end up in $S_0$.

The main result is a general convergence proof for BTs, Theorem 3 including a few examples. We will try to make use of the modularity of BTs, in the sense that the result can be applied at all levels of abstractions, either treating an entire subtree as a single entity as in Figure 5 or in terms of its parts, as in Figure 4.

The idea behind the proof is straightforward, and illustrated in Figure 5. As in Figure 2 the state space is partitioned into operating regions, $\Omega_i$, and overall failure and success regions, $F_0, S_0$, while the arrows indicate possible transitions between these sets. If all transitions from some $\Omega_i$ are to either $S_0$ or $\Omega_j, j > i$, and the state never stays indefinitely in $\Omega_i$, it will eventually reach $S_0$. Note that a similar analysis can be done at several different levels of abstraction. If $\Omega_6 = \Omega_4 \cup \Omega_5$ the analysis can either be done considering $\Omega_4, \Omega_5$ separately as in Figure 5(a), or together, as in Figure 5(b).

![Figure 5](image)

(a) (b)

Figure 5: The idea behind Theorem 3. If the state only transits between sets connected by arrows, and never stays indefinitely in any $\Omega_i$, it will eventually reach $S_0$.

5.1. The general result

The sets $\Omega_i$ are defined below, and correspond to the regions where $u_t$ is running, see Lemma 8. But, in order to define $\Omega_i$, we first need to define the influence regions $I_i$, the parts of the state space where a change of $f_t$ might alter $u_0$, and to define $I_i$ we need notation for parents and older siblings of a node. Some of these results are taken from [13] [14] [15].
Definition 9 (parent and big brother of a node). Given a node \( i \), let \( p(i) \) be the parent of the node and \( b(i) \) be the closest sibling to the left (the big brother) of the node.

Note that \( p(i) \) is undefined if \( i \) is the root, and \( b(i) \) is undefined if there is no sibling to the left.

Definition 10 (Influence region). The Influence Region \( I_i \) of node \( i \) is defined as follows

\[
\begin{align*}
I_i & = X & \text{If } i \text{ is the root} \\
I_i & = I_{p(i)} & \text{If } i \text{ is the leftmost sibling and } \exists p(i) \\
I_i & = I_{b(i)} \cap S_{b(i)} & \text{If } p(i) \text{ is a Sequence and } \exists b(i) \\
I_i & = I_{b(i)} \cap F_{b(i)} & \text{If } p(i) \text{ is a Fallback and } \exists b(i)
\end{align*}
\]

Note that the influence region \( I_i \) is the part of the state space where the design of \( T_i = (u_i, r_i) \) influences the execution of \( T_0 \). Also note that \( I_i \) is fundamentally different from \( S_i, F_i, R_i \) in the sense that \( I_i \) depends entirely on the part of \( T_i \) that is outside \( T_i \), the parent and siblings, while, on the other hand \( S_i, F_i, R_i \) depends entirely on what is inside \( T_i \) itself, \( u_i(x) \).

Lemma 4. If \( x \notin I_i \) then changing the implementation of \( u_i, r_i \) will not change the value of \( u_0(x) \).

Proof. We will use Lemma 5 below, that shows that if \( x \in \Omega_i = I_i \cap R_i \), then \( u_0(x) = u_i(x) \). To maximize the influence of \( u_i, r_i \), we change \( r_i(x) \) to always return running, making \( R_i = X \) and \( \Omega_i = I_i \). However, if \( x \notin I_i \) we have \( x \notin \Omega_i \) and another subtree is still controlling the execution.

As seen above, \( I_i \) depends on external factors and \( S_i, F_i, R_i \) depends on internal factors. We will now define \( \Omega_i \) such that \( \Omega_i \) is the region where \( u_0(x) = u_i(x) \) and \( r_i(x) = R \), that is the region where \( T_i \) is controlling the execution.

Definition 11 (Operating region). The Operating Region \( \Omega_i \) of node \( i \) is defined as follows

\[
\Omega_i = I_i \cap R_i
\]

Lemma 5. For a given node \( j \), the operating regions of the children is a partitioning of \( \Omega_j \), that is

\[
\bigcup_{i \in \mathcal{P}(i) = j} \Omega_i = \Omega_j
\]

Proof. Let the parent index be 0 and the two children indices be 1 and 2. We need to show that this holds for both Sequence and Fallback compositions. If the parent node is a Sequence we have that \( R_0 = R_1 \cup (S_1 \cap R_2) \). For the influence regions, assume that \( I_0 \) is given, which gives \( I_1 = I_0 \) and \( I_2 = I_1 \cap S_1 = I_0 \cap S_1 \). Thus we have

\[
\begin{align*}
\Omega_0 & = I_0 \cap R_0 = I_0 \cap (R_1 \cup (S_1 \cap R_2)) \\
\Omega_1 & = I_1 \cap R_1 = I_0 \cap R_1 \\
\Omega_2 & = I_2 \cap R_2 = I_0 \cap S_1 \cap R_2
\end{align*}
\]

This gives \( \Omega_1 \cap \Omega_2 = I_0 \cap R_1 \cap I_0 \cap S_1 \cap R_2 = \emptyset \), since \( R_1 \cap S_1 = \emptyset \). Furthermore, \( \Omega_1 \cup \Omega_2 = (I_0 \cap R_1) \cup (I_0 \cap S_1 \cap R_2) = I_0 \cap (R_1 \cup (S_1 \cap R_2)) = I_0 \cap R_0 = \Omega_0 \).
Similarly, if the parent node is a Fallback we have that \( R_0 = R_1 \cup (F_1 \cap R_2) \). For the influence regions, assume that \( I_0 \) is given, which gives \( I_1 = I_0 \) and \( I_2 = I_1 \cap F_1 = I_0 \cap F_1 \). Thus we have

\[
\Omega_0 = I_0 \cap R_0 = I_0 \cap (R_1 \cup (F_1 \cap R_2)) \quad 24.
\]

\[
\Omega_1 = I_1 \cap R_1 = I_0 \cap R_1 \quad 25.
\]

\[
\Omega_2 = I_2 \cap R_2 = I_0 \cap F_1 \cap R_2 \quad 26.
\]

This gives \( \Omega_1 \cap \Omega_2 = I_0 \cap R_1 \cap I_0 \cap F_1 \cap R_2 = \emptyset \), since \( R_1 \cap F_1 = \emptyset \). Furthermore, \( \Omega_1 \cup \Omega_2 = (I_0 \cap R_1) \cup (I_0 \cap R_1 \cap F_1 \cap R_2) = I_0 \cap (R_1 \cup (F_1 \cap R_2)) = I_0 \cap R_0 = \Omega_0 \).

**Lemma 6.** For a given subtree, the operating regions of the leaves is a partitioning of the operating region of the root.

**Proof.** A recursive application of Lemma 5. \( \square \)

As described above, we want to enable the convergence analysis to be done at different levels of abstraction, as illustrated in Figure 5. Thus we make the following definition.

**Definition 12.** A level of abstraction \( L \subset \{0, 1, 2, \ldots \} \) is a set of indices such that

\[
X = \bigcup_{i \in L} \Omega_i \cup S_0 \cup F_0, \quad 27.
\]

and \( \Omega_i \cap \Omega_j = \emptyset, \forall i \neq j \in L \).

**Lemma 7.** The root is one level of abstraction, with \( L = \{0\} \) and all the leaves is another level of abstraction with \( L = \{\text{indices of leaves}\} \).

**Proof.** For the root we have \( \Omega_0 = I_0 \cap R_0 = X \cap R_0 = R_0 \). This makes \( \bigcup_{i \in L} \Omega_i \cup S_0 \cup F_0 = R_0 \cup S_0 \cup F_0 = X \). If it holds for the root it must also hold for the leaves, since \( \bigcup_{i \in L} \Omega_i = \Omega_0 \) by Lemma 6. \( \square \)

**Lemma 8.** If \( \mathcal{T}_i \) is a subtree of \( \mathcal{T}_0 \), then \( x \in \Omega_i \), implies that \( u_0(x) = u_i(x) \), that is controller \( i \) is executed while inside \( \Omega_i \).

**Proof.** It holds for the root since \( \Omega_0 = I_0 \cap R_0 = R_0 \). We will now show that if it holds for a parent, it will also hold for a child. Assume it holds for the parent \( j = p(i) \). It remains to show that \( u_j(x) = u_i(x) \). We know that \( x \in \Omega_j \subset \Omega_i \), and \( \Omega_j = I_j \cap R_j \). If \( j \) is the leftmost child then \( x \in I_j \) implies \( x \notin S_j \cap F_j \) which gives \( u_j(x) = u_j(x) \) by equation 22 and 24. Assume the parent is a Sequence node. If \( j \) is not the leftmost child then \( x \in I_j \) implies \( x \in S_{b(i)} \) which gives \( u_j(x) = u_j(x) \) by Equation 23 and 25. Conversely, assume the parent is a Fallback node. If \( j \) is not the leftmost child then \( x \in I_j \) implies \( x \in F_{b(i)} \) which gives \( u_j(x) = u_j(x) \) by Equation 24 and 26. \( \square \)

Given these concepts, we can now formulate our main theorem on convergence of BTs, inspired by 22-24. The idea is that if the state moves through the operating regions \( \Omega_i \) in strictly increasing order, without staying longer than \( \tau \) in any region, and the only other allowed region is \( S_0 \), then the system will reach \( S_0 \) in finite time. Formally, we write

**Theorem 3** (Convergence of BTs). Given a BT, an external constraint region \( \bar{C} \subset X \) that is to be kept invariant, and a level of abstraction \( L \). If there exists a re-labelling of the \( N \) nodes in \( L \) such that

\[
C_i = \left( \bigcup_{j \in L, j \geq i} \Omega_j \right) \cup S_0 \cap \bar{C} \quad 28.
\]

is invariant under \( u_i \) for all \( i \in L \), and there exists a \( \tau > 0 \) such that if \( x(t) \in \Omega_i \) then \( x(t + \tau) \notin \Omega_i \), then there exist a time \( t' \leq N \tau \) such that if \( x(0) \in C_1 \), then \( x(t') \in S_0 \).
Proof. We have that if \( x(t) \in \Omega_i \) then \( x(t + \tau) \notin \Omega_i \), but \( C_i \) is invariant under \( u_i \), so either \( x(t + \tau) \in \Omega_j, j > i \) or \( x(t + \tau) \in S_0 \). Thus there can be at most \( N \) transitions before \( x(t) \in S_0 \), in total taking at most time \( N \tau \).

By saying that a set \( B \subset X \) is invariant under \( u_i \) we mean that if \( x(0) \in B \) and \( \dot{x} = f(x, u_i(x)) \), then \( x(t) \in B \) for all \( t > 0 \), and similarly for discrete time executions with \( x_{t+1} = f(x_t, u_i(x_t)) \).

Remark 3. Note that the challenge in proving convergence for BTs now lies in choosing a level of abstraction, re-ordering the nodes in \( L \), and designing \( u_i \) such that \( C_i \) is invariant and \( x(t + \tau) \notin \Omega_i \).

Remark 4. The purpose of the external constraint region \( \bar{C} \) is to enable separate analysis of a BT that is then being used as a subtree of another BT. If this is not needed, set \( \bar{C} = X \).

The deterministic analysis above can be complemented with a probabilistic result from (25).

Lemma 9 (Probabilistic transitions). If the execution of Definition 2 was replaced by nondeterministic transitions, and the controllers \( u_i \) are such that undesired transitions, from an \( \Omega_i \) to an \( \Omega_j \) with \( j < i \), happen with a probability \( 1 - p_i \), with \( 0 < p \leq p_i < 1 \), then the expected number of transitions \( T \) before reaching \( S_0 \) is bounded above, \( E(T) \leq N/p_i^N \), and the probability of reaching the goal with at most \( k \) transitions is \( P_k = 1 - \gamma^k + 1 \), with \( \gamma = 1 - p_i \), which makes \( P_\infty = 1 \).

Proof. See (25).}

The Lemma above can be interpreted in two ways. Either for nondeterministic executions, as stated in the Lemma, or for deterministic executions, where an un-modeled external agent moves things around, thereby causing a finite set of jumps in the state. Also note that the expected number of transitions gives an upper bound on the expected convergence time of \( \tau N/p_i^N \).

5.2. Three examples

We will now apply Theorem 3 to the sequence of desired goals in Figure 6(a), a fallback of actions where each one is designed to satisfy the preconditions of the one to the left in Figure 6(b) and the more complex mobile manipulation BT from Figure 1. The resulting sets are shown in Table 1. As can be seen, the sets \( C_i \) to be kept invariant are often not that complex, and creating controllers \( u_i \) to satisfy them is often reasonable.)
Lemma 10 (Standard sequence). If $\mathcal{F}_0 = \text{Sequence}(\mathcal{F}_1, \ldots, \mathcal{F}_N)$, we have

$$I_i = \bigcap_{j<i} S_j \cap I_{p(i)}$$

$$\Omega_i = \bigcap_{j<i} S_j \cap I_{p(i)} \cap R_i$$

$$C_i = (\bigcap_{j<i} S_j \cup S_0) \cap \bar{C}$$

Proof. $I_i, \Omega_i$ are clear from the definition of a sequence node, only executing the next child if the previous succeeds. From Theorem $i$ we have that $C_i = ((\bigcup_{j<i} \Omega_j) \cup S_0) \cap \bar{C}$. Thus we need to show that $\bigcup_{j<i} \Omega_j = \bigcap_{j<i} S_j$. Assume $x \in \bigcap_{j<i} S_j$ then $x \in S_j$ for all $j < i$ thus $x \notin \Omega_j$ for all $j < i$ thus $\exists k \geq i : x \in \Omega_k$, since $\Omega_k$ is a partition of $\Omega_{p(i)}$. Conversely, if $x \in \bigcup_{j<i} \Omega_j$ then $\exists k \geq i : x \in \Omega_k$ and thus $x \in \bigcap_{j<i} S_j$. $\square$

For the next example we use the design principle implicit sequence (12, 25), where each child of a fallback node is designed to satisfy the precondition of a sibling to its left, while not failing, see $C_i \subset C_j$ in Equation (34) below. Therefore, the numbering is also done from right to left as part of applying Theorem $i$.

Lemma 11 (Implicit sequence). If $\mathcal{F}_0 = \text{Fallback}(\mathcal{F}_N, \ldots, \mathcal{F}_i)$, and for all $i < N$ there is $j > i$ such that $S_j \cup R_j \supset S_i$ we have that

$$I_i = \bigcap_{j>i} F_j \cap I_{p(i)}$$

$$\Omega_i = \bigcap_{j>i} F_j \cap I_{p(i)} \cap R_i$$

$$C_i \supset C_j = (R_i \cup S_i) \cap \bar{C}$$

Proof. $I_i, \Omega_i$ are clear from the definition of a Fallback node, only executing the next child if the previous fails. Furthermore, $S_0 = S_N$, since $S_i \cap I_i = \emptyset$ for $i \neq N$, due to $S_j \cup R_j \supset S_i$ for some $j > i$.

From Theorem $i$ we have that $C_i = ((\bigcup_{j>i} \Omega_j) \cup S_0) \cap \bar{C}$. Thus we need to show that $\bigcup_{j>i} \Omega_j \cup S_N \supset R_i \cup S_i \cup S_N$. Assume $x \in R_i \cup S_i \cup S_N$ then $x \notin F_i$ and hence $x \notin \bigcup_{j<i} \Omega_j$. Therefore $x \in \bigcup_{j>i} \Omega_j$. $\square$

Using the results above we can compute the sets $C_i$ that needs to be kept invariant by the controller $u_i$ for each region $\Omega_i$. Then a table like the one in Table $i$ can be created, and used as design specification for the $u_i$.

Table $i$ also includes the results for the more complex BT in Figure $i$ with node numbers given by a depth first traversal of the tree. We will now see how such a BT can be created recursively from a list of actions, with corresponding precondition and postconditions.

6. A design principle exploiting modularity and feedback

In this section we will give a concrete example on the use of hierarchical modularity and feedback in BTs. Consider the example BT in Figure $i$ designed to make sure some condition $X$ holds. If $X$ is true it will immediately return success. If not, it will act to make $X$ true, hence the name Make sure $X$. Sometimes there are multiple ways of making $X$ true, in those cases the options are collected under a fallback node, so if one option (say Y) fails, or is not applicable, another option (say Z) can be invoked.

www.annualreviews.org • Behavior Trees in Robot Control Systems
Table 1: Three examples of applying Theorem 3

| Name | Objective | Operating region $\Omega_i$ | To keep invariant $C_i$ |
|------|-----------|-----------------------------|------------------------|
| $T_1$: Make sure in safe area | object at goal | $\in \text{safe area}$ | $\in \text{safe area}$ |
| $T_2$: Make sure object at goal | robot at charger | $\in \text{safe area}$ | $\in \text{safe area}$ |
| $T_3$: Make sure robot at charger | object at goal | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 6(a)

| $T_1$: Make sure object at goal | Make sure object at goal | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 6(b)

| $T_5$: Move to safe area | Proper battery level | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_20$: Grasp object with left arm | Object in gripper | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_22$: Ask other agent to place object | object at goal | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_29$: Move to charger | Robot near object | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_33$: Place object at goal | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_36$: Ask other agent to place object | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1

| $T_39$: Move to goal | $\in \text{safe area}$ | $\in \text{safe area}$ |

Figure 1
Both of Y and Z have their own preconditions, describing when they can be invoked, as illustrated in the figure.

The key idea is now to recursively apply the design on the left of Figure 7 as illustrated on the right of the figure. First we list four top priority goals in a sequence node. Then, instead of just checking the conditions we can replace them with a small BT of the form to the left that tries to make them true. The resulting BT will have some new conditions, which in turn can be replaced by small BTs and so on. All conditions that can be replaced, or have been replaced, are marked with double strokes in both Figure 1 and Figure 7. Note that it does not make sense to replace the single stroked conditions, such as Check X, as these already have actions for achieving them.

This recursive approach is a good example of the hierarchical modularity made possible by BTs. The design of Figure 7 is only about achieving X, not about why, or what will happen later. It also illustrates the use of feedback. The BT first checks if something needs to be done, if not everything is fine. If something needs to be done it tries to achieve it, and if one option failed, another one is applied.

A detailed analysis of the design above can be found in (14), including a discussion on when it works and when it does not work. Here we note that applying Theorem 3 we get the sets of Table 1. Remember that the execution is supposed to progress in increasing order of index labels, thus the table can be read from top to bottom. Note that the sets $C_i$ mostly correspond to not violating previously achieved subgoals, such as in safe set and proper battery level, but when carrying the object to the goal it also includes object in gripper. Perhaps the most surprising item is $C_{36}$ for $T_{36}$, including subtree $T_{11}$ returns failure. This is needed to avoid cases where one option first fails, but during the execution of a fallback option, the first option is somehow activated again before it fails yet again, causing a switching back and forth, see (14) for details.

7. Guaranteeing safety and invariance using control barrier functions

In this section we will see how Control Barrier Functions (CBFs) can sometimes be used to provide the invariance guarantees we need in Theorem 3. Furthermore, we will see how the standard way of using CBFs to guarantee safety appears as a special case of Theorem 3. Finally, we will see how to handle conflicting objectives, as trying to keep several sets invariant might not always be possible.
7.1. Control Barrier Functions

As seen in Section 5 above, the key aspect of Theorem 3 is for \( u_i \) to keep the set \( C_i \) invariant. Fortunately, keeping sets invariant is the main objective of CBFs. Below we will assume that the execution in Definition 2 runs in continuous time, but corresponding discrete time concepts can also be created.

The key idea behind CBFs (26, 27) is to specify a barrier function \( h : X \rightarrow \mathbb{R} \) such that the so-called safe set \( \mathcal{C} \) is characterized by:

\[
\mathcal{C} = \{ x \in X : h(x) \geq 0 \}.
\]

Given the continuous system dynamics \( \dot{x} = f_c(x, u) \), if we choose controls \( u \in U_{inv} \) where \( U_{inv} = \{ u \in U : \frac{dh}{dx} f_c(x, u) \geq -\alpha(h(x)) \} \), and \( \alpha \in \mathcal{K} \) is a class K functions (26), we are guaranteed to stay in the safe set \( x \in \mathcal{C} \).

7.2. Guaranteeing safety and handling conflicting objectives

Guaranteeing safety is often of highest priority. In this section we will see how we can use CBF to address the invariance property of Theorem 3, in a way that includes safety guarantees as a special case, as described in (28).

Since we need the conditions \( C_i \) to be invariant, we first make the following assumption:

**Assumption 2.** Each condition \( C_i : X \rightarrow \{0, 1\} \) can be formulated in terms of a CBF \( h_i \), see Equation (35), as follows

\[
C_i = \{ x \in X : h_i(x) \geq 0 \}\] 36.

Having just one CBF we can guarantee invariance, but if there are several, they might represent conflicting objectives, such as in safe area and at charger, if the charger happens to be located outside the safe set. Normally, the intersection of the corresponding control sets \( U_{inv} \) would guarantee invariance of all sets, but if the objectives are conflicting, the intersection might be empty. In these cases we will make use of the fact that the BT includes a clear priority order of the objectives, e.g., that in safe area is more important than at charger. The idea is then to include as many sets as possible in the intersection, while still making sure the intersection is non-empty.

Thus, we define the following sets of controls, where \( U_i \subset U \) guarantees invariance of \( C_i \), \( \hat{U}_i \subset U \) guarantees invariance of all \( C_j, j \leq i \) (but might be empty) and \( \hat{U}_i \subset U \) guarantees invariance of some of the \( C_j, j \leq i \) (but is guaranteed to be non-empty).

**Definition 13.** Let

\[
U_i = \{ u \in U : \frac{dh_i}{dx} f(x, u) \geq -\alpha(h_i(x)) \} \] 37.

\[
\hat{U}_i = \bigcap_{j=1}^i U_j \] 38.

\[
\check{U}_i = \bigcup_{j=1}^i \bar{U}_j : j \leq i, \bar{U}_j \neq \emptyset \wedge (j = i \lor \bar{U}_{j+1} = \emptyset) \] 39.

We can now choose a control inside \( \check{U}_i \) that is as close as possible to some other desired value \( w_i(x) \) that is designed to reach the current subgoal, as in the CBF-QP of (26),

\[
u_i = \arg\min_u ||u - w_i(x)||^2 \quad \text{s.t. } u \in \check{U}_i\] 40.
If we apply this approach to an arbitrarily complex BT, such as the one in Figure 1, with a safety objective as first priority, the CBF approach above will guarantee that we will never violate this objective. In the best of worlds, we might achieve all objectives, but we know that the robot will always be safe.

8. Explainable AI and human robot interaction

As robots share workspaces with humans to an increasing extent, questions regarding human robot interaction become more important. Safety, as seen above, is often most important, but to achieve efficient interaction it is also important that the human can predict, trust and understand the robot.

In (29) a number of guidelines for trustworthy autonomy are mentioned. These include that the system should be transparent and traceable, in the sense that “the system must be able to explicitly explain its reasoning in a concise and usable format (either visual or textual)”. As illustrated in Figure 1 this requirement is satisfied by BTs in the sense that at any point, you can find the leaf node that is executing and follow the branch all the way up to the root to see why this subtree is executing. If the recursive backward chained approach described in Section 6 is used, reading the expanded preconditions (double stroked in Figure 1) we see that the robot is currently executing Move to object, (in order to) Make sure robot near object, (to) make sure object in gripper, (to) make sure object at goal. There is a need for more work on user aspects of BTs, including human robot interaction, but early examples include (30).

9. Reinforcement Learning, Utility and BTs

Reinforcement learning (RL) is a research area aiming to produce near optimal controllers for a very general family of control problems. Sometimes so-called end-to-end solutions can be found, mapping raw sensor readings to actions, to extremely challenging problems (31). If all problems could be solved end-to-end using RL, there would be no need for BTs, but there is reason to believe that modular hierarchical control structures will still be useful for a number of years, especially since they enable safety guarantees, see Section 7, and transparency to a human operator, see Section 8. A natural question is then how we can combine RL with BTs, ideally to get the performance of RL, and the guarantees and transparency of BTs.

Figure 8: There are many ways of combining RL with BTs. Single actions can be replaced (1), as can entire subtrees (2). One could also add a new subtree (3), and keep the old subtrees as fallback options if the new one fails. Finally, the order of fallback options can be changed (4), as can the order of satisfying preconditions (5).

A number of different ways of combining BTs with RL are illustrated in Figure 8. The first option
that comes to mind is perhaps to replace a single action with RL (1 in Figure 8). This was explored in (32) (33), where an RL problem including states, actions and rewards was specified by the user. If the problem domain is well suited for RL this approach can then be expanded, replacing subtrees (2 in Figure 8) in a bottom up fashion. This gradual approach can be seen as a low risk option to replacing an entire control structure with end-to-end RL.

RL can also be used to increase the performance of an existing BT. One way of doing this is to keep the current structure, but add an additional option using RL, as suggested in (34) and illustrated in (item 3 of Figure 8). If the RL option fails, the other ones will execute and achieve the subgoal.

Another approach, (item 4 in Figure 8), focusing on subtree order was explored in (35, 36, 37, 38). As the Q-value of a state-action pair in RL estimates the future reward, it was noted that Q-values could be used to choose between fallback options, reordering them based on the Q-value in the current state. A similar idea was explored in (39), where the success probability of each child was estimated by gathering data during execution, and the order was updated to keep the child with highest value first. Finally, the least explored option (item 5 in Figure 8) is to also reorder pre-conditions in the BT. For example, fetching a bunch of items, as in Figure 8, amounts to a small instance of a traveling salesman problem (TSP) where ordering might have impact on performance.

10. Evolutionary Algorithms and BTs

Evolutionary algorithms, or genetic algorithms, (40), are local optimization algorithms inspired by the theory of evolution. The basic idea is to maintain a family of solution candidates, and then create new solution candidates from the previous ones by applying mutations (small alteration of a candidate) and crossover (taking two candidates, mark a subset in each and swap the subsets). The candidates are then evaluated using a fitness function, and some portion of them are removed.

The modularity of BTs, with a uniform interface on all subtree levels, makes them well suited for evolutionary algorithms. Applying mutation to a BT can be done by picking an arbitrary subtree, and replacing it with some other subtree, as illustrated in Figure 9(a). Furthermore, crossover can similarly be done by taking two BTs, choosing a subtree in each and swap them, as illustrated in Figure 9(b).

![Figure 9: Applying mutation and crossover to a BT](image)

It has been shown that locality, in terms of small changes in design giving small changes in performance, is important for the performance of evolutionary algorithms (41). As seen in Lemma 4 above, the influence region $I_i$ captures the part of the state space where a subtree can influence system...
behavior. Thus, for larger BTs, a given subtree will often have a fairly limited $I_i$ providing the locality described in (41).

A well known problem of evolutionary algorithms is so-called bloating (42), where the average size of the individuals in a population grows, without a corresponding increase in fitness. For BTs it is clear that a design could have large parts that do not contribute at all. Both (43) and (44) describe methods for addressing this problem. A practical approach is to try pruning different subtrees to see if fitness is reduced, whereas a more theoretical approach is to compute the influence regions $I_i$ for all the subtrees, and remove the ones with $I_i = 0$.

Work on BTs and evolutionary algorithms can be found in (45, 46, 43, 47, 48). Finally, combinations of BTs, evolutionary algorithms and planning can be found in (49), and multi-agent problems using BTs and evolutionary algorithms have been addressed in (50, 51).

11. Planning and BTs

Planning algorithms are typically used to create a sequence of actions that will move the world state from a given starting state to some desired goal state. Planning can either be on a lower level, such as motion planning or grasp planning, or a higher level, such as task planning. Low level planning are usually integrated as leaves in a BT, whereas high level planning can be used to create the BT itself. The reason for combining BTs and planning algorithms is often to add the reactive feedback properties of a BT to the goal directed actions output by the planner.

The most straightforward way of using a task planner is to first run the planner to get a sequence of actions, and then execute this sequence. This works fine if the world is static and the actions are predictable. However, if an action fails, the sensing of the world was inaccurate, or an external agent changes the world state, the planned sequence of actions will not lead to the goal state. A natural way to add feedback to the system is to monitor the execution to see if it runs as predicted, and re-plan once there is a significant enough deviation. However, in many cases the original plan is still valid, we just need to jump to the proper phase. If a grasping action fails, we can try grasping again. If an object is picked up and later dropped, we can jump back to the pick up action. If an external agent helps us with some subgoal, we can jump ahead to the proper action in the list. As seen above, BTs are an appropriate tool for supplying this kind of feedback control, and were used in e.g. (25).

Planning might also be used to create a feedback policy. One example of this is the A* algorithm that computes the shortest path to goal from all initial states, not only the one currently occupied. Thus, if an unexpected action not only moves us back or forth on the expected path, but also sideways to a state that was not intended to be occupied, the plan still contains the proper action. The design described in Section 6 provides this functionality in a BT (14). An advantage is the larger region of attraction, while a drawback is that the BT needed to cover all potential situations can be very large.

Another use of the reactivity of BTs in connection to planning can be found in (52). Here a BT is created from the output of the planner with the intent of reactively taking advantage of opportunities for parallel execution, and tasks finishing earlier than expected. By tracking the preconditions for each action, they are executed as early as possible, based on information that was not available at planning time.

The combination of planners with BT has been explored in (53, 54, 55, 56, 57, 58, 59, 49). Furthermore, the special cases of HTN planners were investigated in (60, 61, 62, 63), and LTL planners in (64, 65, 66).
12. Conclusions

In this paper we have provided a control theoretic approach to BTs, showing how they can be seen as a hierarchically modular way to create a switched dynamical system, where the switching is based on feedback from lower level modules. We have also showed how the resulting operating regions can be computed, based on the specifications of parents, siblings and children of the node. Using these operating regions, we present sufficient conditions of convergence to the goal region of the entire BTs, as well as practical designs that can be used to create convergent BTs. Finally, we have showed how these core result connect to other research efforts on BTs, including control barrier functions, explainable AI, reinforcement learning, genetic algorithms and planning.

SUMMARY POINTS

1. Behavior trees represent a hierarchically modular way to combine controllers into more complex controllers.
2. Behavior trees enable feedback control, not only on the lowest level, but on all levels, as the interface explicitly includes meta information (feedback) regarding the applicability and progress of a controller, that enables the parent level to act based on this feedback.
3. The modular structure of behavior trees lends itself to formal analysis regarding convergence and region of attraction.
4. Ongoing work connects behavior trees to other research areas such as planning and learning.

FUTURE ISSUES

1. Reinforcement learning can solve many problems end-to-end. However, many robot systems will need a modular structure combining separate capabilities, such as path planning and grasping. Behavior trees is a viable option for this structure and the connections between reinforcement learning and behavior trees needs to be explored further.
2. Explainable AI, learning by demonstration and human robot interaction (HRI) are areas where the transparency of BTs could play an important role.
3. Behavior trees have been explored from an AI and robotics perspective, but very little work has been done from a control theoretic point of view.

DISCLOSURE STATEMENT

The authors are not aware of any affiliations, memberships, funding, or financial holdings that might be perceived as affecting the objectivity of this review.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support from SSF through the Swedish Maritime Robotics Centre (SMaRC) (IRC15-0046), and by FOI through project 7135.
1. Blume M, Appel AW. 1999. Hierarchical modularity. ACM Transactions on Programming Languages and Systems 21(4):813–847
2. Sacerdoti ED. 1975. A Structure for Plans and Behavior. Tech. rep., SRI International AI center
3. Erol K, Hendler J, Nau DS. 1994. UMCP: a sound and complete procedure for hierarchical task-network planning. In Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, AIPS’94, pp. 249–254. Chicago, Illinois: AAAI Press
4. Mateas M, Stern A. 2002. A behavior language for story-based believable agents. IEEE Intelligent Systems 17(4):39–47
5. Isla D. 2005. Handling Complexity in the Halo 2 AI. In Proceedings of the Game Developers Conference (GDC)
6. Florez-Puga G, Gomez-Martín MA, Gomez-Martín PP, Díaz-Aguado B, González-Calero PA. 2009. Query-Enabled Behavior Trees. IEEE Transactions on Computational Intelligence and AI in Games 1(4):298–308
7. Oğren P. 2012. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In AIAA Guidance, Navigation, and Control Conference. Minneapolis, Minnesota: American Institute of Aeronautics and Astronautics
8. Bagnell JA, Cavalcanti F, Cui L, Galluzzo T, Hebert M, et al. 2012. An Integrated System for Autonomous Robotics Manipulation. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2955–2962
9. Harel D. 1987. Statecharts: a visual formalism for complex systems. Science of Computer Programming 8(3):231–274
10. Biggar O, Zamani M, Shames I. 2021. An Expressiveness Hierarchy of Behavior Trees and Related Architectures. IEEE Robotics and Automation Letters 6(3):5397–5404
11. Iovino M, Scukins E, Styrud J, Oğren P, Smith C. 2020. A Survey of Behavior Trees in Robotics and AI. arXiv:2005.05842 [cs]
12. Colledanchise M, Oğren P. 2018. Behavior Trees in Robotics and AI: An Introduction. CRC Press
13. Colledanchise M, Oğren P. 2017. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. IEEE Transactions on Robotics 33(2):372–389
14. Oğren P. 2020. Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees. IEEE Robotics and Automation Letters 5(4):6073–6080
15. Sprague CI, Ögren P. 2021. Continuous-time behavior trees as discontinuous dynamical systems. IEEE Control Systems Letters 6:1891–1896
16. Colledanchise M, Natale L. 2021. On the Implementation of Behavior Trees in Robotics. IEEE Robotics and Automation Letters 8
17. Cortes J. 2008. Discontinuous Dynamical Systems. IEEE Control Systems Magazine 28(3):36–73
18. Filippov AF. 1988. Differential Equations with Discontinuous Righthand Sides: Control Systems. Springer Science & Business Media
19. Biggar O, Zamani M, Shames I. 2020. On modularity in reactive control architectures, with an application to formal verification. arXiv preprint arXiv:2008.12515
20. Gallai T. 1967. Transitiv orientierbare Graphen. Acta Mathematica Academiae Scientiarum Hungarica 18(1):25–66
21. Watson AH, Wallace DR, McCabe TJ. 1996. Structured Testing: A Testing Methodology Using the Cyclo-matic Complexity Metric. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology
22. Burridge RR, Rizzi AA, Koditschek DE. 1999. Sequential Composition of Dynamically Dexterous Robot Behaviors. The International Journal of Robotics Research 18(6):534–555
23. Conner DC, Choset H, Rizzi AA. 2006. Integrated Planning and Control for Convex-bodied Nonholonomic Systems using Local Feedback Control Policies. In Robotics: Science and Systems, vol. 2
24. Reist P, Tedrake R. 2010. Simulation-based LQR-trees with input and state constraints. In 2010 IEEE International Conference on Robotics and Automation, pp. 5504–5510
25. Paxton C, Ratliff N, Eppner C, Fox D. 2019. Representing Robot Task Plans as Robust Logical-Dynamical Systems. In 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5588–5595. ISSN: 2153-0866

26. Ames AD, Coogan S, Egerstedt M, Notomista G, Sreenath K, Tabuada P. 2019. Control Barrier Functions: Theory and Applications. In 2019 18th European Control Conference (ECC), pp. 3420–3431

27. Ögren P. 2006. Autonomous UCAV Strike Missions Using Behavior Control Lyapunov Functions. In AIAA Guidance, Navigation, and Control

28. Özkahraman, Ögren P. 2020. Combining Control Barrier Functions and Behavior Trees for Multi-Agent Underwater Coverage Missions. In 2020 59th IEEE Conference on Decision and Control (CDC), pp. 5275–5282

29. Endsley MR. 2015. Autonomous Horizons: System Autonomy in the Air Force – A Path to the Future. Volume I: Human-Autonomy Teamng. Tech. rep., United States Air Force Office of the Chief Scientist

30. Paxton C, Jonathan F, Hundt A, Mutlu B, Hager GD. 2017. User Experience of the CoSTAR System for Instruction of Collaborative Robots. arXiv:1703.07890 [cs]

31. Vinyals O, Babuschkin I, Czarnecki WM, Silver D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature 575(7782):350–354

32. Pereira RdP, Engel PM. 2015. A Framework for Constrained and Adaptive Behavior-Based Agents. arXiv:1506.02312 [cs]

33. Kartasev M. 2019. Integrating Reinforcement Learning into Behavior Trees by Hierarchical Composition. Master thesis, KTH Royal Institute of Technology

34. Sprague CI, Ögren P. 2018. Adding Neural Network Controllers to Behavior Trees without Destroying Performance Guarantees. arXiv:1809.10283 [cs]

35. Dey R, Child C. 2013. QL-BT: Enhancing Behaviour Tree Design and Implementation with Q-Learning. In 2013 IEEE Conference on Computational Intelligence in Games (CIG), pp. 1–8. Niagara Falls, ON, Canada: IEEE

36. Fu Y, Qin L, Yin Q. 2016. A Reinforcement Learning Behavior Tree Framework for Game AI. In Proceedings of the 2016 International Conference on Economics, Social Science, Arts, Education and Management Engineering. Huhhot, China: Atlantis Press

37. Zhang Q, Sun L, Jiao P, Yin Q. 2017. Combining Behavior Trees with MAXQ Learning to Facilitate CGFs Behavior Modeling. In 2017 4th International Conference on Systems and Informatics (ICSAI), pp. 525–531

38. Zhu X. 2019. Behavior tree design of intelligent behavior of non-player character (NPC) based on Unity3D. Journal of Intelligent & Fuzzy Systems 37(5):6071–6079

39. Hannaford B, Hu D, Zhang D, Li Y. 2016. Simulation Results on Selector Adaptation in Behavior Trees. arXiv:1606.09219 [cs]

40. Whitley D. 1994. A genetic algorithm tutorial. Statistics and Computing 4(2):65–85

41. Rothlauf F, Oetzel M. 2006. On the locality of grammatical evolution. In European Conference on Genetic Programming, pp. 320–330. Springer

42. Luke S, Panait L. 2006. A Comparison of Bloat Control Methods for Genetic Programming. Evolutionary Computation 14(3):309–344

43. Colledanchise M, Parasuraman R, Ögren P. 2019. Learning of Behavior Trees for Autonomous Agents. IEEE Transactions on Games 11(2):183–189

44. Hallawa A, Schug S, Iacca G, Ascheid G. 2020. Evolving Instinctive Behaviour in Resource-Constrained Autonomous Agents Using Grammatical Evolution. In Applications of Evolutionary Computation, ed. PA Castillo, JL Jiménez Laredo, F Fernández de Vega, pp. 369–383, Lecture Notes in Computer Science, pp. 369–383. Cham: Springer International Publishing

45. Lim CU, Baumgarten R, Colton S. 2010. Evolving Behaviour Trees for the Commercial Game DEFCON. In Applications of Evolutionary Computation, ed. D Hutchison, GN Yannakakis, pp. 100–110, vol. 6024. Springer Berlin Heidelberg

46. Nicolau M, Perez-Liebana D, O’Neill M, Brabazon A. 2017. Evolutionary Behavior Tree Approaches for Navigating Platform Games. IEEE Transactions on Computational Intelligence and AI in Games 9(3):227–238

47. Iovino M, Styrud J, Falco P, Smith C. 2020. Learning Behavior Trees with Genetic Programming in Unpre...
48. Paduraru C, Paduraru M. 2019. Automatic Difficulty Management and Testing in Games Using a Framework Based on Behavior Trees and Genetic Algorithms. In arXiv:1909.04368 [Cs]

49. Styri J, Iovino M, Norrlöf M, Björkman M, Smith C. 2021. Combining Planning and Learning of Behavior Trees for Robotic Assembly. arXiv:2103.09036 [cs] ArXiv: 2103.09036

50. Neupane A, Goodrich M. 2019. Learning Swarm Behaviors Using Grammatical Evolution and Behavior Trees. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, pp. 513–520. Macao: International Joint Conferences on Artificial Intelligence Organization

51. Jones S, Studley M, Hauert S, Winfield A. 2018. Evolving Behaviour Trees for Swarm Robotics. In Distributed Autonomous Robotic Systems: The 13th International Symposium, ed. R Groß, A Kolling, S Berman, E Frazzoli, A Martinoli, F Matsuno, M Gauci, pp. 487–501, Springer Proceedings in Advanced Robotics. Cham: Springer International Publishing

52. Martín F, Morelli M, Espinosa H, Lera FJR, Matellán V. 2021. Optimized Execution of PDDL Plans using Behavior Trees. arXiv:2101.01964 [cs]

53. Colledanchise M, Almeida D, Ögren P. 2019. Towards Blended Reactive Planning and Acting using Behavior Trees. In IEEE Int. Conference on Robotics and Automation. Montreal, Canada: IEEE

54. Tadewos TG, Shamghah L, Karimoddini A. 2019. Automatic Safe Behaviour Tree Synthesis for Autonomous Agents. In 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 2776–2781

55. Tadewos TG, Shamghah L, Karimoddini A. 2019. On-the-Fly Decentralized Tasking of Autonomous Vehicles. In 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 2770–2775

56. Zhou H, Min H, Lin Y. 2019. An Autonomous Task Algorithm Based on Behavior Trees for Robot. In 2019 2nd China Symposium on Cognitive Computing and Hybrid Intelligence (CCHI), pp. 64–70

57. Paxton C, Ratliff N, Eppner C, Fox D. 2019. Representing Robot Task Plans as Robust Logical-Dynamical Systems. In 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5588–5595

58. Schwab P, Hlavacs H. 2015. Capturing the Essence: Towards the Automated Generation of Transparent Behavior Models. In Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference

59. Kuckling J, Ligot A, Bozhinoski D, Birattari M. 2018. Behavior Trees as a Control Architecture in the Automatic Modular Design of Robot Swarms. In Swarm Intelligence, ed. M Dorigo, pp. 30–43, Lecture Notes in Computer Science. Springer International Publishing

60. Neufeld X, Mostaghim S, Brand S. 2018. A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees. In Fourteenth International Conference and Interactive Digital Entertainment Conference

61. Rovida F, Grossmann B, Krüger V. 2017. Extended Behavior Trees for Quick Definition of Flexible Robotic Tasks. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 6793–6800

62. Segura-Muros JÁ, Fernández-Olivares J. 2017. Integration of an Automated Hierarchical Task Planner in ROS Using Behaviour Trees. In 2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT), pp. 20–25

63. Hölzl M, Gabor T. 2015. Reasoning and Learning for Awareness and Adaptation. In Software Engineering for Collective Autonomic Systems: The ASCENS Approach, ed. M Wirsing, M Hölzl, N Koch, P Mayer, pp. 249–290, Lecture Notes in Computer Science. Cham: Springer International Publishing

64. Colledanchise M, Murray RM, Ögren P. 2017. Synthesis of Correct-by-Construction Behavior Trees. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 6039–6046

65. Lan M, Lai S, Lee TH, Chen BM. 2019. Autonomous Task Planning and Acting for Micro Aerial Vehicles. In 2019 IEEE 15th International Conference on Control and Automation (ICCA), pp. 738–745

66. Biggar O, Zamani M. 2020. A Framework for Formal Verification of Behavior Trees with Linear Temporal Logic. IEEE Robotics and Automation Letters