How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees

Michele Colledanchise and Petter Ögren

***Abstract*— Behavior Trees (BTs) is a way of organizing the switching structure of a control system, that was originally developed in the computer gaming industry but is now also being used in robotics. The Teleo-Reactive programs (TRs) is a highly cited reactive hierarchical robot control approach suggested by Nilsson and And-Or-Trees are trees used for heuristic problems solving.**

In this paper, we show that BTs generalize TRs as well as And-Or-Trees, even though the two concepts are quite different. And-Or-Trees are trees of conditions, and we show that they transform into a feedback execution plan when written as a BT. TRs are hierarchical control structures, and we show how every TR can be written as a BT. Furthermore, we show that so-called Universal TRs, guaranteeing that the goal will be reached, are a special case of so-called Finite Time Successful BTs. This implies that many designs and theoretical results developed for TRs can be applied to BTs.

I. Introduction

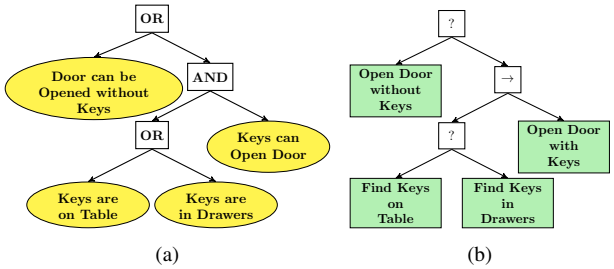
Behavior Trees (BTs) were introduced in the computer gaming industry [1]–[3], as a tool to develop the control structure of in-game opponents, and have since then matured to the level that they are now included in several textbooks on the topic [4], [5]. Following the development in industry, BTs have now also started to receive attention in academia, starting in the game/AI community [6]–[9] and moving over to robotics [10]–[15].

In robotics, BTs have been used in a range of different applications, including robotic manipulation [11], non-expert programming of pick and place operations [15], brain surgery robotics [14], and UAV mission execution [10].

More theoretical work include the verification of mission plans in [12] and the estimation of resulting execution times in [13]. BTs have also been used in learning applications [6], [7]. In the area of modelling, details regarding parameter passing was investigated in [8], and a Modellica implementation of BTs was presented in [16].

The Teleo-reactive programs (TRs) were first proposed by Nilsson in a highly cited seminal paper [17]. This work has then been extended in several directions, including integrating TRs with automatic planning and machine learning [18], [19] removing redundant parts of TRs [20], and playing robot soccer [21]. And-Or-Trees is a tool that is used in heuristic problem solving [22].

The contributions of this paper is that we show how BTs generalize TRs, and how the result on Universal TRs in [17] is analogous to the richer continuous state space result on compositions of Finite Time Successful BTs. The BT result is actually a bit stronger, as it covers a case that can be used as a counter example to the Universal TRs. The analogy result in turn implies that much of the work on TR, with the initial paper [17] being cited over 380 times, can be translated into BT designs. We also show how And-Or-Trees can be directly translated into BT feedback executions, as in Figure 1. This fact, together with the observation that TRs and And-Or-Trees are quite different, shows that BTs are not equivalent to the other two structures, but in fact richer, as it generalizes both.



(a) (b)

Fig. 1. An And-Or-Tree of conditions describing if a door can be opened (a) and the corresponding BT encoding a feedback execution plan to open the door (b). As can be seen, the transformation from (a) to (b) is quite straight forward.

The outline of this paper is as follows. In Section II we review BTs and TRs. Then, in Section III we describe the analogy between BTs and And-Or-Trees. Section IV then shows how a TR can be written in the form of a BT. The formal properties of TRs and BTs are analyzed in Section V, using a new functional description of BTs, with details in the Appendix. Finally, the paper is concluded in Section VI.

II. Background: BTs and TR

In this section, we will introduce BTs and TRs in the classical way. And-Or-Trees are introduced in the following section.

A more detailed description of BTs can be found in [10] and a more detailed description of TRs can be found in [17].

A. Behavior Trees

Following the definition of [10], a BT is a directed tree with the classical definition of parent, child, leaf and root node. There are four different BT node types:

Action, *Condition*, *Fallback*, and *Sequence*. Actions and Condition are always leaf nodes., Fallbacks and Sequences are always internal nodes (non-leaf nodes). Table I lists the BT node types used in this paper. Additional ndoe types (e.g. *Decorator* and *Parallel* nodes) can be found in [1]–[3], [10].

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

The BT execution is driven by *ticks*. A tick is a signal that allows the execution of a node that receives it. The ticks are generated by the root with a chosen frequency and they progress from a parent to the children according to rules of the different node types. Whenever the tick reaches a leaf node, the node does some computation and then it returns the status of *Success*, *Running*, or *Failure* to its parent.

Now we are ready to describe how the tick is handled by each different node types and which return status is sent.

Fallback. Whenever a *fallback* node receives a *tick*, it send a *tick* in turn from the most left to the most right child. If a ticked child returns a status of *success* or *running*, the *fallback* node sends such status to its parent. If all the children return a status of *failure*, the fallback node returns such status to its parent. A *fallback* node is graphically represented with a white box with the label "?" as in Figure 2. Algorithm 1 describes the tick handling of a *fallback* node.

Fallback nodes are used for situations where a set of tasks can be achieved using different alternatives. In such cases it is enough to execute one working alternative.

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
```

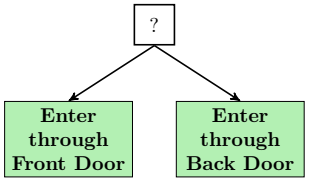


Fig. 2. Fallback node with 2 children. The Fallback ticks its children in order until one returns Success or Running. The action *Enter through back door* is a fallback alternative to *Enter through front door*, and is only executed if the first one fails.

Sequence. Whenever a *sequence* node receives a *tick*, it send a *tick* in turn from the most left to the most right child. If a ticked child returns a status of *failure* or *running*, the *sequence* node sends such status to its parent. If all the children return a status of *success*, the fallback node returns such status to its parent. A *sequence* node is graphically represented with a white box with the label " → " as in Figure 3. Algorithm 2 describes the tick handling of a *sequence* node.

Sequence nodes are used for situations where some tasks have to be executed in a given sequence. In such cases whenever a task fails, it is useless to execute the next tasks.

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
```

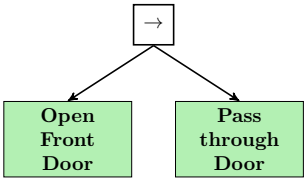


Fig. 3. The Sequence ticks its children in order until one returns failure or running. Sequences are denoted by a white square with an arrow. The action *Pass through Door* is only executed if *Open front door* succeeds.

Action. The *action* node is a leaf node (i.e. it does not have children to send ticks). Whenever an *action* node receives a tick, it performs some interaction with the system controlled and it returns a status of *Success*, *Failure* or *Running* according to whether the action is completed, it is not possible to complete or it is too early to determine whether an the action will succeed or fail. An *action* node is graphically represented with a rectangle with a custom made label, usually describing what the action does as in Figures 3 and 2.

Condition. The *condition* node is a leaf node. Whenever a *condition* node receives a tick, it evaluates a proposition of the system controlled or the environment and it returns a status of *Success* or *Failure* if the proposition is satisfied or not. A *condition* node is graphically represented with an ellipse with a custom made label, usually describing what the condition verifies as in Figure 4 .

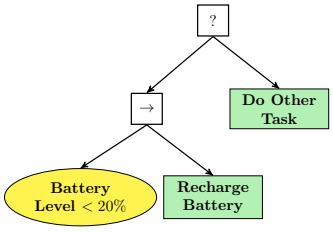


Fig. 4. A Fallback, a Sequence, a Condition (yellow) and two Actions (green). *Recharge Battery* is only executed when the condition *Battery Level* < 20% is true. Otherwise, *Do other tasks* is executed.

B. The Teleo-Reactive Approach

As described in [17], [23], a TR program is composed of a set of prioritized condition-action rules. The conditions

TABLE I. The six 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 succeeds	If one child fails	If one child returns running
Action	Upon completion	When impossible to complete	During completion
Condition	If true	If false	Never

for each rule are continuously evaluated, and the action with the highest priority out of the ones were the condition is satisfied, is executed. Thus, a TR-program is denoted by

$$k_1 \to a_1; \ldots; k_m \to a_m, \tag{1}$$

where the $k_i : \mathbb{R}^k \to \{0, 1\}$ are conditions that are true of false, depending on the sensor data, and $a_i : \mathbb{R}^k \to U$ are actions mapping sensor data to control variables. The actions a_i might be either atomic functions, or TR-programs themselves.

Often, k_1 is the *goal* condition, and a_1 is the *idle* action, corresponding to doing nothing when you reach the goal.

In [17] the following analysis was presented.

Definition 1 (Regression property): A TR has the Regression property if, for each $k_i, i > 1$ there is $k_j, j < i$ such that the execution of action a_i leads to the satisfaction of k_j.

Definition 2 (Complete): If k_i are such that $k_1 \vee k_2 \vee \ldots \vee k_m$ always holds, then the TR-program is called Complete.

Definition 3 (Universal): A TR-program is called Universal, if k_i, a_i are such that the TR-program is Complete and satisfies the Regression property.

Lemma 1 (Nilsson 1994): If a TR-program is Universal, and there are no sensing and execution errors, then the execution of the program will lead to the satisfaction of k_1.

Proof: In [17] it is stated that it is easy to see that this is the case. ■

The idea of the proof is indeed straight forward, but as we will see when we compare it to the BT results in Section V below, the proof is incomplete.

III. Analogy between And-Or-Trees and BTs

In this section we describe the analogy between And-Or-Trees and BTs. And-Or-Trees are used in heuristic problem solving [22], and have two types of nodes, *OR nodes* that combine subtrees representing *alternative* ways of solving a problem, and *AND nodes* which represent problem composition into independent subproblems, *all of which* needs to be solved to solve the original problem.

It can be noted that And-Or-Trees have alternating levels of AND and OR nodes, and a solution S to a And-Or-Search tree T is not a path, but a subtree, which contains the root node of T and a sufficiently large set of the other nodes such that no contradiction occurs when all nodes in S are True and all others are False. Thus if a node in S is an AND node, all its children needs to also be in S, but if it is an OR node, only one of its children needs to be in S, [24].

Example 1: Consider the And-Or-Tree in Figure 5. The problem at hand is that of opening a door. The possible solutions to the problem are 1) *Door can be opened without keys* is true, or 2) the combination of *Keys can open door* and *Keys are on table* is true, or 3) the combination of *Keys can open door* and *Keys are in drawer* is true. The latter solution is represented by the sub-tree indicated by thicker edges in the figure.

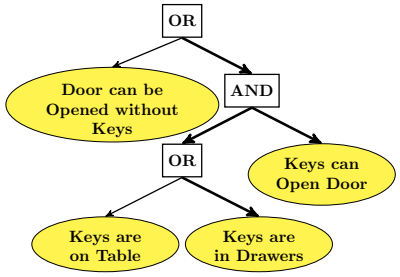


Fig. 5. The door can be opened if the top node is true. This happens when e.g., all conditions with thick edges are true, see Example 1.

The BT analogy of Example 1 can be found in Figure 6. There, the OR nodes are replaced by Fallbacks (requiring just one child to succeed) and the AND nodes are replaced by Sequences (requiring all children to succeed). If this is done, the BT works as an exact copy of the And-Or-Tree, as conditions never return Running. However, if the conditions are replaced by actions trying to make the corresponding conditions true, returning Running while trying, and returning *Success* or *Failure* after the action is completed, we get a rational feedback execution aiming towards completing the overall task as described below.

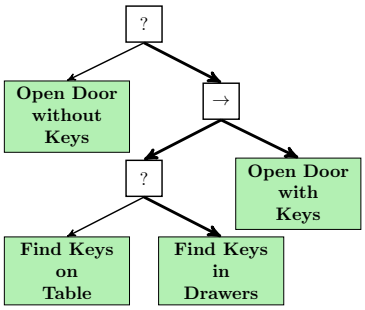


Fig. 6. The BT analogy of the And-Or-Tree in Figure 6. The door is successfully opened when the top node returns Success. This happens e.g., when all the nodes with thick edges return Success.

Executing the BT in Figure 6 on a robot would thus lead to the robot first trying to *Open door without keys*. If this action succeeds, the robot is done. If the action fails the sequence is ticked, which in turn ticks the fallback, which

in turn ticks *Find keys on table*. If this action succeeds, the robot continues to *Open door with keys*. If *Find keys on table* fails on the other hand, the robot continues with *Find keys in drawer*. If both actions aimed at finding the keys fail, there is no need to try *Open door with keys*. Instead, the robot returns failure.

Thus, the And-Or-Tree of conditions was turned into a BT feedback execution plan, executing only actions that might lead to overall task completion.

IV. Analogy between TRs and BTs

In this section, we use the following Lemma to show how to create a BT with the same execution as a given TR. The lemma is illustrated by Example 2 and Figure 7.

Lemma 2 (TR-BT analogy): Given a TR in terms of conditions k_i and actions a_i, an equivalent BT can be constructed as follows

$$\mathcal{T}_{TR} = \text{Fallback}(\text{Sequence}(k_1, a_1), \ldots, \text{Sequence}(k_m, a_m)), \tag{2}$$

where we convert the True/False of the conditions to Success/Failure, and let the actions only return Running.

Proof: It is straightforward to see that the BT above executes the exact same a_i as the original TR would have, depending on the values of the conditions k_i, i.e. it finds the first condition k_i that returns Success, and executes the corresponding a_i. ■

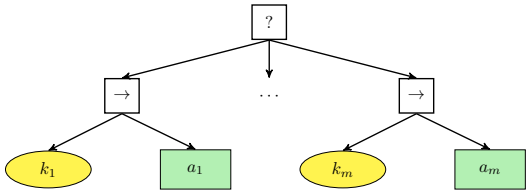


Fig. 7. The BT that is analogous to a given TR.

We will now illustrate the lemma with an example from Nilssons original paper [17].

Example 2: The TR *Goto(loc)* is described as follows, with conditions on the left and corresponding actions to the right:

$$\text{equal(pos,loc)} \rightarrow \text{idle} \tag{3}$$
$$\text{heading towards (loc)} \rightarrow \text{go forwards} \tag{4}$$
$$\text{(else)} \rightarrow \text{rotate} \tag{5}$$

where *pos* is the current robot position and *loc* is the current destination.

Executing this TR, we get the following behavior. If the robot is at the destination it does nothing. If it is heading the right way it moves forward, and else it rotates on the spot. In a perfect world without obstacles, this will get the robot to the goal, just as predicted in Lemma 1. Applying Lemma 2, the Goto TR is translated to a BT in Figure 8.

The example continues in [17] with a higher level recursive TR, called *Amble(loc)*, designed to add a basic obstacle

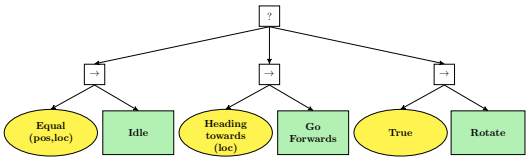


Fig. 8. The BT version of the TR Goto.

avoidance behavior

$$\text{equal(pos,loc)} \rightarrow \text{idle} \tag{6}$$
$$\text{clear-path(pos,loc)} \rightarrow \text{Goto(loc)} \tag{7}$$
$$\text{(else)} \rightarrow \text{Amble(new-point(pos,loc))} \tag{8}$$

where *new point* picks a new random point in the vicinity of *pos* and *loc*.

Again, if the robot is at the destination it does nothing. If the path to goal is clear it executes the Goto TR. Else it picks a new point relative to its current position and destination (loc) and recursively executes a new copy of Amble with that destination. Applying Lemma 2, the Amble TR is translated to a BT in Figure 9.

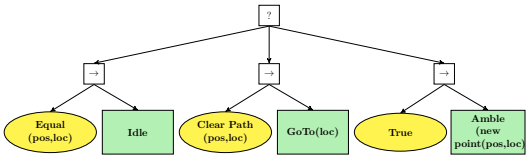


Fig. 9. The BT version of the TR Amble.

V. Universal TRs and FT-Successful BTs

Using the functional form of BTs that was introduced in [25], and included in the Appendix for completeness, we can prove a richer version of Lemma 1, and also fix one of its assumptions.

This new lemma includes execution time, but more importantly builds on a finite difference equation system model over a continuous state space. Thus control theory concepts can be used to include phenomena such as imperfect sensing and actuation into the analysis, that was removed in the strong assumptions of Lemma 1. Thus, the BT analogy provides a powerful tool for analyzing TR designs.

Lemma 3: (Robustness and Efficiency of Fallback Compositions) If $\mathcal{T}_1, \mathcal{T}_2$ are Finite Time Successful, with $S_2 \subset R_1'$, then $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2)$ is Finite Time Successful with $\tau_0 = \tau_1 + \tau_2$, $R_0' = R_1' \cup R_2'$ and $S_0 = S_1$.

Proof: See Appendix. ■

Here, S_i, R_i, F_i correspond to Success, Running and Failure regions, see Equations (12) and R' denotes the region of attraction, see Definition 10.

To illustrate Lemma 5 we look at Figures 10 and 11. The BT to be analyzed is $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2)$, the corresponding sets S_i, R_i, F_i are shown in Figure 10 and the corresponding vector fields are illustrated in Figure 11.

The Lemma shows under what conditions we can guarantee that the Success region S_0 is reached in finite time. If we for illustrative purposes assume that the regions of attraction

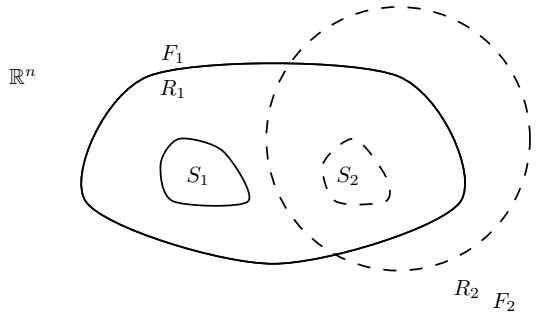


Fig. 10. The different sets of the two BTs. Solid lines indicated set boundaries for $\mathcal{T}_1$ and dashed lines indicate boundaries for $\mathcal{T}_2$.

are identical to the running regions $R_i = R_i'$, the Lemma states that as long as the system starts in $R_0' = R_1' \cup R_2'$ it will reach $S_0 = S_1$ in less than $\tau_0 = \tau_1 + \tau_2$ time units. The condition analogous to the Regression property is that $S_2 \subset R_1'$, i.e. that the Success region of the second BT is a subset of the region of attraction R_1' of the first BT.

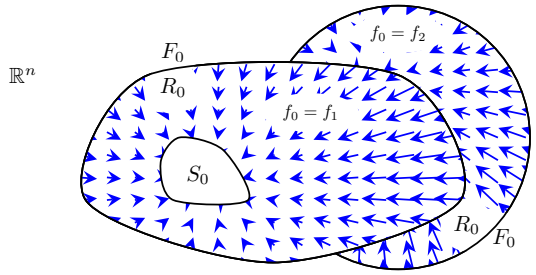


Fig. 11. An illustration of the vector field f_0 of the composition $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2)$, with the corresponding set S_0, R_0, F_0.

As described above, the regions of attraction, R_1' and R_2' are very important, but there is no corresponding concept in Lemma 1. In fact, we can construct a counter example showing that Lemma 1 does not hold.

Example 3 (Counter Example): Assume that a TR program is Universal in the sense described above. Thus, the execution of action a_i eventually leads to the satisfaction of k_j where $j < i$ for all $i \neq 1$. However, assume it is also the case that the execution of a_i, on its way towards satisfying k_j actually leads to a violation of k_i. This would lead to the first true condition being some k_m, with $m > i$ and the execution of the corresponding action a_m. Thus, the chain of decreasing condition numbers is broken, and the goal condition a_1 might never be reached.

The fix is however quite straightforward, and amounts to using the following stronger assumption.

Definition 4 (Stronger Regression property): For each $k_i, i > 1$ there is $k_j, j < i$ such that the execution of action a_i leads to the satisfaction of k_j, *without ever violating* k_i.

VI. Conclusions

In this paper we have shown how BTs generalize TRs as well as And-Or-Trees. The connection to TRs is important as it allows results developed for one framework to be applied in the other. For example, the theory for Finite Time Successful BT compositions can be used to analyze TR designs. The connection to And-Or-Trees shows that BTs are not just a new variation of TRs but instead something richer.

Appendix - Functional Representation of BTs

Following [25], we define a more formal, functional version of the BTs described above. The *tick* is now described by recursive function calls, incorporating both the return status and the dynamics of the control system. These definitions enable us to describe and prove properties of the BTs.

Definition 5 (Behavior Tree (BT)): A BT is a three-tuple

$$\mathcal{T}_i = \{f_i, r_i, \Delta t\}, \tag{9}$$

where $i \in \mathbb{N}$ is the tree index, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ is the right hand side of an ordinary difference equation, Δt is the time step and $r_i : \mathbb{R}^n \to \{R, S, F\}$ is the return status, that can be: *Running* (R), *Success* (S), or *Failure* (F).
The return status r_i is used to combine BTs, as described below.

Definition 6 (Executing a BT): The execution of a BT $\mathcal{T}_i$ is a standard ordinary difference equation

$$\begin{aligned} x_{k+t}(t_{k+1}) &= f_i(x_k(t_k)), && (10)\\ t_{k+1} &= t_k + \Delta t. && (11) \end{aligned}$$

Without loss of generality we assume that all the subtrees in a BT evolve in the same space $\mathbb{R}^n$ with time step Δt_i.

Definition 7: The three regions $R_i, S_i, F_i \subset \mathbb{R}^n$ of a BT $\mathcal{T}_i$ are defined as follows

$$\begin{aligned} R_i &= \{x : r_i(x) = R\} && (12)\\ S_i &= \{x : r_i(x) = S\}\\ F_i &= \{x : r_i(x) = F\} \end{aligned}$$

and denoted Running region (R_i), Success region (S_i) and Failure region (F_i).

Definition 8 (Sequence compositions of BTs): The sequence operator allows to compose two different BTs into a larger BT as follows:

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

Where r_0, f_0 (i.e. the return status and the differential equation describing the tree) are defined below

$$\begin{aligned} &\text{If } x_k \in S_1 && (13)\\ &\quad r_0(x_k) = r_2(x_k) && (14)\\ &\quad f_0(x_k) = f_2(x_k) && (15)\\ &\text{else}\\ &\quad r_0(x_k) = r_1(x_k) && (16)\\ &\quad f_0(x_k) = f_1(x_k). && (17) \end{aligned}$$

When executing $\mathcal{T}_0$ it first keeps executing $\mathcal{T}_1$ (the first child) as long as this returns either *running* or *failure*. $\mathcal{T}_2$ (the second child) is executed only in the case when $\mathcal{T}_1$ returns *success*. $\mathcal{T}_0$ returns *success* to its parent if and only if $\mathcal{T}_1$ and $\mathcal{T}_2$ return *success*.

For 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 any sequence compositions.

Definition 9 (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 follows

$$\begin{aligned} &\text{If } x_k \in \mathcal{F}_1 && (18)\\ &\quad r_0(x_k) = r_2(x_k) && (19)\\ &\quad f_0(x_k) = f_2(x_k) && (20)\\ &\text{else}\\ &\quad r_0(x_k) = r_1(x_k) && (21)\\ &\quad f_0(x_k) = f_1(x_k). && (22) \end{aligned}$$

When executing $\mathcal{T}_0$ it first keeps executing $\mathcal{T}_1$ (the first child) as long as this returns either *running* or *success*. $\mathcal{T}_2$ (the second child) is executed only in the case when $\mathcal{T}_1$ returns *failure*. $\mathcal{T}_0$ returns *failure* to its parent if and only if $\mathcal{T}_1$ and $\mathcal{T}_2$ return *failure*.

For convenience, we write

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

and similarly for any sequence compositions.

In many real-world scenarios the goal can be described as achieving a desired configuration in a time efficient and robust manner. Given a state space, the time efficiency can be described as reaching a desired subset of the state space in time and the robustness can be described as reaching the desired subset of the state space from a larger subset of initial positions.

Definition 10 (Finite Time Successful): A BT is Finite Time Successful with region of attraction R', if for all starting points $x(0) \in R' \subset R$, there is a time τ such that $x(\tau') \in S$ for some $\tau' \leq \tau$ and $x(t) \in R'$ for all $t \in [0, \tau')$.

Given a right choices of the sets S, F, R for a BTs the exponential stability implies finite time success. The following lemma formalizes this result.

Lemma 4 (Exponential stability and FTS): A BT for which x_s is a globally exponentially stable equilibrium of the execution (10), and $S \supset \{x : \|x - x_s\| \leq \epsilon\}$, $\epsilon > 0$, $F = \varnothing$, $R = \mathbb{R}^n \setminus S$, is Finite Time Successful.

Proof: See. [25] ■

Lemma 5: (Robustness and Efficiency of Fallback Compositions) If $\mathcal{T}_1, \mathcal{T}_2$ are Finite Time Successful, with $S_2 \subset R_1'$, then $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2)$ is Finite Time Successful with $\tau_0 = \tau_1 + \tau_2$, $R_0' = R_1' \cup R_2'$ and $S_0 = S_1$.

Proof: See. [25] ■

REFERENCES

[1] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, 2005.

[2] A. Champandard, "Understanding Behavior Trees," *AiGameDev. com*, vol. 6, 2007.

[3] D. Isla, "Halo 3-building a Better Battle," in *Game Developers Conference*, 2008.

[4] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2009.

[5] S. Rabin, *Game AI Pro*. CRC Press, 2014, ch. 6. The Behavior Tree Starter Kit.

[6] C. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON," *Applications of Evolutionary Computation*, pp. 100–110, 2010.

[7] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, no. 99, pp. 1–1, 2016.

[8] A. Shoulson, F. M. Garcia, M. Jones, R. Mead, and N. I. Badler, "Parameterizing Behavior Trees," in *Motion in Games*. Springer, 2011.

[9] 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*. Springer, 2011, pp. 159–168.

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

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

[12] A. Klökner, "Interfacing Behavior Trees with the World Using Description Logic," in *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

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

[14] 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 *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[15] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, "A framework for end-user instruction of a robot assistant for manufacturing," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[16] A. Klöckner, F. van der Linden, and D. Zimmer, "The Modelica BehaviorTrees Library: Mission planning in continuous-time for unmanned aircraft," in *Proceedings of the 10th International Modelica Conference*, 2014.

[17] N. J. Nilsson, "Teleo-reactive programs for agent control," *JAIR*, vol. 1, pp. 139–158, 1994.

[18] S. Benson and N. J. Nilsson, "Reacting, planning, and learning in an autonomous agent." in *Machine intelligence 14*. Citeseer, 1995, pp. 29–64.

[19] B. Vargas and E. Morales, "Solving navigation tasks with learned teleo-reactive programs," *Proceedings of IEEE International Conference on Robots and Systems (IROS)*, 2008.

[20] S. R. Mousavi and K. Broda, *Simplification Of Teleo-Reactive sequences*. Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[21] G. Gubisch, G. Steinbauer, M. Weiglhofer, and F. Wotawa, "A teleo-reactive architecture for fast, reactive and robust control of mobile robots," in *New Frontiers in Applied Artificial Intelligence*. Springer, 2008, pp. 541–550.

[22] J. Pearl, "Heuristics: intelligent search strategies for computer problem solving," 1984.

[23] J. L. Morales, P. Sánchez, and D. Alonso, "A systematic literature review of the teleo-reactive paradigm," *Artificial Intelligence Review*, vol. 42, no. 4, pp. 945–964, 2014.

[24] R. Marinescu and R. Dechter, "And/or tree search for constraint optimization," in *Proc. of the 6th International Workshop on Preferences and Soft Constraints*. Citeseer, 2004.

[25] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Robustness and Safety in Hybrid Systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, June 2014.