Formal Verification based Synthesis for Behavior Trees

Weijiang Hong[1,2,3], Zhenbang Chen[1,2(✉)][0000−0002−4066−7892], Minglong Li[1,3], Yuhan Li[1], Peishan Huang[1,2,3], and Ji Wang[1,2,3(✉)][0000−0003−0637−8744]

[1] College of Computer, National University of Defense Technology, China
[2] Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, China
[3] Institute for Quantum Information & State Key Laboratory of High Performance Computing, National University of Defense Technology, China
`{hongweijiang17, zbchen, liminglong10, liyuhan, Huang_ps, wj}@nudt.edu.cn`

Abstract. Behavior trees (BTs) have been extensively applied in the area of both computer games and robotics, as the control architectures. However, the construction of BTs is labor-expensive, time-consuming, and even impossible as the complexity of task increases. In this work, we propose a formal verification based synthesis method to automatically construct BTs whose behaviors satisfy the given Linear Temporal Logic (LTL) specifications. Our method first explores candidate BTs by a grammar-based Monte Carlo Tree Search (MCTS), then the explored BTs are transformed into Communicating Sequential Processes (CSP) models. After that, we invoke the verifier to check the models' correctness *w.r.t.* specifications, and provide feedback based on the verification result for guiding the search process. The application of our method on several representative robotic missions indicates its promising.

Keywords: Behavior Trees · MCTS · CSP · Synthesis.

1 Introduction

Behavior Trees (BTs) [15] are models that control the agent's decision-making behavior through a hierarchical tree structure. The development of BTs can be traced back to their applications in the field of computer games, wherein BTs are initially used to facilitate the design and control of non-player characters [26, 18, 20]. Afterwards, BTs' application is gradually extended to the area of robotics, like mobile ground robots [19, 7], unmanned aerial vehicles [17, 28], to name a few. It's incontestable that BTs play a more and more important role in the area of both computer games and robotics, as the control architectures.

Compared with other control architectures, like Decision Trees (DTs) [27], Teleo-reactive Programs (TRs) [22] and Finite State Machines (FSMs) [14], the *reactivity* and *modularity* of BTs make it more applicable and flexible to accomplish tasks in unpredictable environments [8]. The support for *reactivity* means the agent can interrupt an ongoing task to execute another task for reacting to the environment changes, while the support of *modularity* means we can naturally combine several individually designed BTs into a more complex BT without providing any auxiliary glue-codes. Although BTs are being adopted and developed due to their promising features, their construction is still problematic especially when dealing with complex robotic tasks in unpredictable environments. Manually designing BTs usually becomes labor-expensive, time-consuming, and even impossible as the task involves more and more objects and subtasks. Therefore, it's expected that the construction of BTs can be automatic.

In this work, we propose a formal verification based synthesis method to automatically construct BTs whose behaviors satisfy the given Linear Temporal Logic (LTL) specifications. Our method first explores candidate BTs by a grammar-based Monte Carlo Tree Search (MCTS). Considering that BTs are not suitable formal models for formal verification *w.r.t.* properties, the explored BTs are transformed into Communicating Sequential Processes (CSP [13]) models. After that, we invoke the verifier to check the models' correctness *w.r.t.* specifications, and provide feedback based on the verification result for guiding the search process. The main contributions of this work are as follows:

- We proposed a formal verification based synthesis method for BTs. To the best of our knowledge, the combination of the verification method and the synthesis method for BTs hasn't been investigated before.
- We designed a CSP modelling method for BTs to capture their behaviors. The correctness of BTs' behaviors thereby can be checked by verifying the CSP model *w.r.t.* LTL specifications.
- We provided a method to evaluate the verification result and utilized this evaluation feedback to guide the search process, which improved the search efficiency.
- We successfully synthesized the expected BTs for several representative robotic missions within 1 hour, which indicates the effectiveness of our method.

This paper is organized as follows. Some backgrounds are provided in Section 2. The problem formulation and the proposed approach are presented in Section 3 and Section 4, respectively. The demonstration result is shown in Section 5. The overview for the existing work is given in Section 6. Finally, Section 7 concludes the paper.

2 Background

This section first briefly introduces Behavior Trees with an example that used throughout the paper. Then, we provide some necessary prerequisite knowledge about Linear Temporal Logic and Communicating Sequential Processes.

2.1 Behavior Trees

Behavior Trees (BTs) [15] are the hierarchical trees that control the agent's decision-making behavior. Figure 1 shows a BT example that controls the robot to pick up a cube from a specific position. The leaf nodes of BTs are execution nodes that can be classified into *action nodes* and *condition nodes*. Apart from those execution nodes, there are four types of control flow nodes that include *sequences nodes*, *fallbacks nodes*, *parallel nodes*, and *decorator nodes*. In this work, we mainly focus on the usage of *sequences nodes* and *fallbacks nodes*. More details about them can be found below:

- *action nodes*: The gray and rectangle-shaped ones, like `GotoPos`, `Pickup`. It may return one of the three statuses: success (the action is completed), failure (the action is impossible to complete), running (the action is not completed yet).
- *condition nodes*: The ellipse-shaped ones, like `Picked`, `AtPos`. It may only return success (the condition holds) or failure (the condition does not hold).
- *sequences nodes*: It is represented as the symbol $\rightarrow$. It returns success when all of its children return success in an order from left to right; returns failure/running as soon as one of its children returns failure/running.
- *fallbacks nodes*: It is represented as the symbol `?`. It returns failure when all of its children return failure in an order from left to right; returns success/running as soon as one of its children returns success/running.

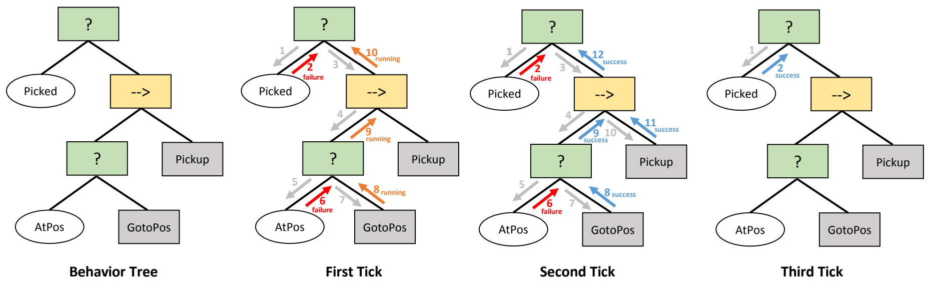

Fig. 1. BT example.

The execution of BTs starts from the root and infinitely delivers *ticks* to its children at a particular frequency. *Ticks* can be seen as a signal to identify which node is executable. We take one possible execution of the BT example for illustrations, which requires that the robot firstly goes to the specific position, then picks up a cube. The node's execution order *w.r.t. ticks* is labeled, as shown in Figure 1. We assume that the conditions `Picked` and `AtPos` don't hold initially. In the first tick, the condition node `Picked` is ticked but returns failure, then the condition node `AtPos` is ticked due to the functionality of these control flow nodes. `AtPos` also returns failure and the action node `GotoPos` is ticked and returns running. The status running is propagated to the root since all the control flow nodes will return running if one of its children returns running. Similar to the first tick, the action node `GotoPos` is ticked again but returns success in the second tick. During the second tick, the action node `Pickup` is also ticked since the first child of the sequences node returns success and `Pickup` returns success. The success of `Pickup` means the condition `Picked` holds. Therefore, the condition node `Picked` always returns success in the third and following ticks.

2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is widely used to describe specifications about long-term goals, like safety and liveness. It can be used to specify the BTs' behaviors [3]. The basic elements of LTL include a set of atomic proposition $p \in P$, the propositional logic operators $\neg$ (*negation*), $\vee$ (*disjunction*), and $\wedge$ (*conjunction*), and several temporal logic operators $\mathcal{X}$ (*next*), $\mathcal{U}$ (*until*), $\mathcal{G}$ (*always*), and $\mathcal{F}$ (*eventually*).

$$\varphi ::= p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathcal{X}\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \mathcal{G}\varphi \mid \mathcal{F}\varphi$$

For example, the previous BT example controls the robot to finally pick up a cube can be expressed as the LTL specification $\mathcal{F}\, p$, wherein p denotes "*Picked holds*". We only mentioned those notions used in this paper, and more details about LTL can be found in [1].

2.3 Communicating Sequential Processes

Communicating Sequential Processes, or CSP [13] is a kind of language for describing an interaction system consisting of processes. For convenience, we use the lower case letters $(\mathtt{a}, \mathtt{b}, ...)$ to represent events in the process and the upper case letters $(\mathtt{P}, \mathtt{Q}, ...)$ to represent processes. The following provides part of the standard semantics, and more details about the completed semantics can be found in [13].

$$\mathtt{P} \ ::= \mathtt{Skip} \mid \mathtt{a} \rightarrow \mathtt{P} \mid \mathtt{P;}\ \mathtt{P} \mid \mathtt{P} \,\Box\, \mathtt{P}$$

We take some expressions used in this paper as examples:

- `Skip` is a normal terminated process that does nothing.
- $\mathtt{a} \rightarrow \mathtt{P}$ executes the event `a` then behaves like the process `P`. If `a` is additionally decorated by $[\mathtt{Guard}]\ \mathtt{a}\ \{\mathtt{Program}\}$, it requires that `a` can only be executed when the condition `Guard` satisfies and the effect of executing `a` is represented by `Program`.
- $\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2$ executes the process $\mathtt{P}_1$ first, then executes the process $\mathtt{P}_2$.
- $\mathtt{P}_1 \,\Box\, \mathtt{P}_2$ executes the process $\mathtt{P}_1$ or $\mathtt{P}_2$, wherein which process executed depends on the external environment.

The following CSP model represents all executions in the form of $\mathtt{a}^*\mathtt{b}$. The whole process may continuously executes the event `a` due to the process `P`'s recursive behavior or executes the event `b` once and terminates immediately due to `Skip`.

$$\mathtt{P} = (\mathtt{a} \rightarrow \mathtt{P}) \,\Box\, (\mathtt{b} \rightarrow \mathtt{Skip});$$

There are many verifiers to support the verification of CSP model *w.r.t.* the LTL specification φ, like PAT [24] and FDR [11], to name a few.

3 Problem Formulation

Before carrying out our formal verification based synthesis method, there are some prerequisites: the BTs' construction grammar, the action nodes' function descriptions and the LTL specification. The BTs' construction grammar is shown in Figure 2, wherein the action nodes $\{\texttt{act}_1, .., \texttt{act}_N\}$ and the condition nodes $\{\texttt{cond}_1, .., \texttt{cond}_M\}$ are regarded as the terminal symbols. The grammar will be used to carry a grammar-based MCTS to generate plenty of candidate BTs.

$$\begin{array}{rl} \texttt{Root} ::= & (\ \texttt{? Root Root}\ ) \mid (\ \rightarrow\ \texttt{Root Root}\ ) \\ & \mid (\ \texttt{? C Root}\ ) \mid (\ \rightarrow\ \texttt{C Root}\ ) \mid \texttt{A} \\ \texttt{A} ::= & \texttt{act}_1 \mid \texttt{act}_2 \mid ... \mid \texttt{act}_N \\ \texttt{C} ::= & \texttt{cond}_1 \mid \texttt{cond}_2 \mid ... \mid \texttt{cond}_M \end{array}$$

Fig. 2. The BTs' construction grammar.

As for the action nodes' function descriptions, we noticed that the interaction between BTs and unpredictable environments is actually reflected in the effect of actions $\{\texttt{act}_1, .., \texttt{act}_N\}$ on conditions $\{\texttt{cond}_1, .., \texttt{cond}_M\}$. Therefore, we regard the condition set as a proposition set Σ, and represent the interaction snapshot between BTs and environments as a state set $\mathcal{S} = 2^{\Sigma}$. The function of an action can be represented in the form of $s_1 \xrightarrow{a} s_2$, wherein $s_1, s_2 \in \mathcal{S}$ and $a \in \{\texttt{act}_1, .., \texttt{act}_N\}$. For each action node, we clarify its function as shown in Table 1, which will be used to facilitate the CSP modelling for BTs.

Table 1. The function descriptions of action nodes.

Action	**Requirement**	**Result**
..	...	...
$\texttt{act}_i$	$\texttt{cond}_j$, ..., $\neg\texttt{cond}_k$	$\texttt{cond}_p$, ..., $\neg\texttt{cond}_q$
..	...	...
GotoPos	\	AtPos
Pickup	AtPos, ¬Picked	Picked

We use cond and ¬cond to represent whether the condition holds or not, respectively. For each action node $\texttt{act}_i$, **Requirement** is the pre-condition needed to be satisfied before executing $\texttt{act}_i$, *i.e.*, $s_1 \models \texttt{cond}_j \wedge ... \wedge \neg\texttt{cond}_k$; while **Result** is the post-condition caused by successfully executing $\texttt{act}_i$, *i.e.*, $s_2 \models \texttt{cond}_p \wedge ... \wedge \neg\texttt{cond}_q$. For example, the execution of GotoPos is without any requirements, but results in AtPos to be hold; the execution of Pickup requires that AtPos holds while Picked does not hold, but Picked will hold after the success execution of Pickup.

Goal: Given the BTs' construction grammar, the function descriptions of action nodes, and the LTL specification φ, our goal is to synthesize the expected BT whose behavior satisfies the specification φ.

4 Proposed Approach

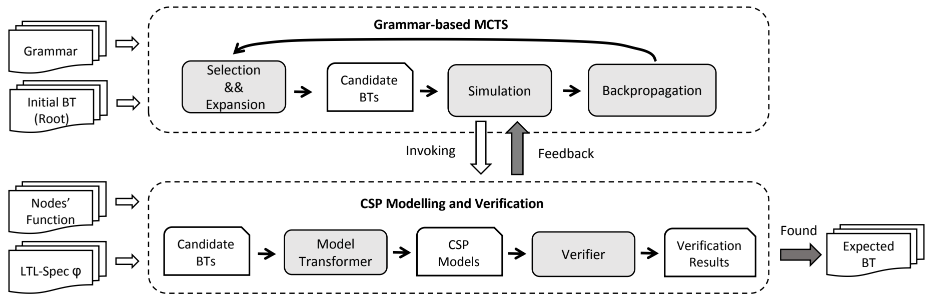

Fig. 3. The formal verification based synthesis framework.

The formal verification based synthesis framework is presented in Figure 3, which consists of two modules: `Grammar-based MCTS` and `CSP Modelling and Verification`. In this framework, we conduct a Monte Carlo Tree Search process based on the grammar rules shown in Figure 2, which starts from the non-terminal symbol `Root` and continually expands non-terminal symbols to obtain the candidate BTs. During the simulation phase, the candidate BT will be transformed into CSP model with the nodes' function embedding, then the model *w.r.t.* LTL specification φ will be checked by the verifier. The verification result will be utilized to provide feedback for guiding the search process. This search process will repeat until the expected BT has been found or the time is running out. In the following subsections, we will first focus on the verification module, then combine it with the search module.

4.1 CSP Modelling and Verification

Given the candidate BT $\mathcal{B}$ and the nodes' function $\mathcal{F}$, we construct the corresponded CSP model $\mathcal{M}$ in a bottom-up manner as shown in Algorithm 1. Considering that each node with its children in a BT is corresponded to a sub-BT, we use a mapping to store the corresponding CSP model for each node/sub-BT (line 1). During the modelling process, we first construct CSP model for each leaf node *w.r.t.* $\mathcal{F}$ (line 2-5), then focus on the structure building for the control node (line 6-14). For each control node whose CSP model is not defined but its child nodes are defined (line 8), we compose these child nodes together based on the type of control nodes to construct the control node's CSP model (line 9-13). Finally, we select the root's CSP model to return (line 15-17). The corresponding CSP model for the BT in Figure 1 is presented in Figure 4. We next describe the Function Embedding and the Structure Building for Figure 4 in detail.

Algorithm 1: Modelling($\mathcal{B}$, $\mathcal{F}$)

```
Require: The candidate behavior tree $\mathcal{B}$ and the function information of action
    nodes $\mathcal{F} = \{(require_1, action_1, result_1), ..., (require_n, action_n, result_n)\}$.
Ensure: The corresponding CSP model $\mathcal{M}$ for $\mathcal{B}$.
 1: $\mathcal{E} \leftarrow \{\}$      // Mapping each node to a CSP model
 2: // Function Embedding
 3: for each leaf node $n$ in $\mathcal{B}$ do
 4:     $\mathcal{E}[n] \leftarrow$ functionEmbedding($n, \mathcal{F}$)
 5: end for
 6: // Structure Building
 7: while exist undefined control node in $\mathcal{B}$ w.r.t. $\mathcal{E}$ do
 8:     $(n_1, n_2, n) \leftarrow$ select($\mathcal{B}, \mathcal{E}$) // $n_1$,$n_2$ are defined, while the parent $n$ is not.
 9:     if $n$ is $\rightarrow$ then
10:         $\mathcal{E}[n] \leftarrow$ composeSequence($n_1, n_2$)
11:     else
12:         $\mathcal{E}[n] \leftarrow$ composeFallback($n_1, n_2$)
13:     end if
14: end while
15: // The Final Model w.r.t. Root
16: $\mathcal{M} \leftarrow \mathcal{E}$[selectRoot($\mathcal{B}$)]
17: return $\mathcal{M}$
```

```
 1 var a1, a2;       // a1/a2 represents whether GotoPos/Pickup returns success
 2 var c1, c2;       // c1/c2 represents whether Picked/AtPos returns success
 3 var n1=0, n2=0;   // n1/n2 records the number of running times an action keeps
 4 #define N 2;      // N times running must eventually leads to success
 5 
 6 BT = ([c1==0] picked_f -> ([c2==0] atpos_f ->
 7           (([n1<N] gotopos_r {n1++;a1=0;c2=0} -> BT)
 8            []
 9            ([n1==0] gotopos_f {n1=0;a1=0;c2=0} -> BT)
10            []
11            ([n1<=N] gotopos_s {n1=0;a1=1;c2=1} ->
12                     // ---------- the modelling for terminals ---------- //
13                     ([n2<N&&c1==0&&c2==1] pickup_r {n2++;a2=0;c1=0} -> BT
14                      []
15                      [n2==0&&(c1==1||c2==0)] pickup_f {n2=0;a2=0;c1=0} -> BT
16                      []
17                      [n2<=N&&c1==0&&c2==1] pickup_s {n2=0;a2=1;c1=1} -> BT)
18                     // -------- the modelling for non-terminals -------- //
19                     // Unknown -> BT
20           )))[]
21           ([c2==1] atpos_s ->
22                     ([n2<N&&c1==0&&c2==1] pickup_r {n2++;a2=0;c1=0} -> BT
23                      []
24                      [n2==0&&(c1==1||c2==0)] pickup_f {n2=0;a2=0;c1=0} -> BT
25                      []
26                      [n2<=N&&c1==0&&c2==1] pickup_s {n2=0;a2=1;c1=1} -> BT)
27                     // Unknown -> BT
28      ))[]
29      ([c1==1] picked_s -> BT);
30 
31 P = (set_c1_0{c1=0} -> set_c2_0{c2=0} -> Skip); BT;
32 
33 #assert P |= F pickup_s; // check whether the cube is finally picked up?
```

Fig. 4. The CSP model for the BT example in Figure 1, wherein `[]` represents the external choice □ and `//` represents comments. (1) if `picked` holds, the process is re-executed (line 29); (2) if `picked` doesn't hold but `atpos` holds, `pickup` is executed (line 22-26); (3) if both `picked` and `atpos` don't hold, `gotopos` and `pickup` are executed (line 7-17).

Function Embedding First, we present CSP model without the function embedding for each leaf node. Note that, a single leaf node can be also regarded as a BT. For example, the model for a condition node $\texttt{cond}_\texttt{i}$ is like

$$\texttt{BT} = (\texttt{cond}_\texttt{i}\texttt{\_f} \to \texttt{BT}) \, \Box \, (\texttt{cond}_\texttt{i}\texttt{\_s} \to \texttt{BT})$$

It captures the two possible return statuses of $\texttt{cond}_\texttt{i}$ by the external choice operator $\Box$, wherein $\texttt{cond}_\texttt{i}\texttt{\_f}$ and $\texttt{cond}_\texttt{i}\texttt{\_s}$ are the events that mean the node returns failure and success, respectively. Besides, this node will be infinitely ticked at a particular frequency, which has been depicted as the recursion structure. The action node $\texttt{act}_\texttt{i}$ can be constructed in the similar way by adding $\texttt{act}_\texttt{i}\texttt{\_r}$ for the status running as shown below.

$$\texttt{BT} = (\texttt{act}_\texttt{i}\texttt{\_r} \to \texttt{BT}) \, \Box \, (\texttt{act}_\texttt{i}\texttt{\_f} \to \texttt{BT}) \, \Box \, (\texttt{act}_\texttt{i}\texttt{\_s} \to \texttt{BT})$$

Second, we consider to add the function embedding into the model. The key point is using a flag ($\texttt{c}_\texttt{i}$ for $\texttt{cond}_\texttt{i}$ and $\texttt{a}_\texttt{i}$ for $\texttt{act}_\texttt{i}$) to indicate which status the execution node returns (`1` for success and `0` for others), which will be further used to guide other nodes' executions. The function embedding of events in CSP model is presented in the following form

$$\underbrace{[\ldots\ \ldots\ \ldots]}_{\texttt{Guard}} \ \texttt{Event} \ \underbrace{\{\ldots\ \ldots\ \ldots\}}_{\texttt{Program}}$$

, wherein `Guard` is a boolean expression to represent the pre-condition needed for `Event` to take, and `Program` is the detailed description for the effect of `Event` taken. For each condition node $\texttt{cond}_\texttt{i}$, its related events will be depicted by

$$[\texttt{c}_\texttt{i}\texttt{==0}] \ \texttt{cond}_\texttt{i}\texttt{\_f} \ \{\ \} \qquad [\texttt{c}_\texttt{i}\texttt{==1}] \ \texttt{cond}_\texttt{i}\texttt{\_s} \ \{\ \}$$

The value of $\texttt{c}_\texttt{i}$ depends on the environment initialization as shown in line 31 of Figure 4 that assumes there is no condition holds initially. Besides, the value of $\texttt{c}_\texttt{i}$ can be also altered by the execution of action nodes as shown below.

For illustrations, we assume the requirement for $\texttt{act}_\texttt{i}$ is $\texttt{cond}_\texttt{j} \wedge \neg\texttt{cond}_\texttt{k}$, and the result for $\texttt{act}_\texttt{i}$ is $\texttt{cond}_\texttt{p}$, its related events will be depicted by

$$[\texttt{c}_\texttt{j}\texttt{==1 \&\& c}_\texttt{k}\texttt{==0}] \ \texttt{act}_\texttt{i}\texttt{\_r} \ \{\texttt{c}_\texttt{p}\texttt{=0; a}_\texttt{i}\texttt{=0}\}$$

$$[\texttt{c}_\texttt{j}\texttt{==0 || c}_\texttt{k}\texttt{==1}] \ \texttt{act}_\texttt{i}\texttt{\_f} \ \{\texttt{c}_\texttt{p}\texttt{=0; a}_\texttt{i}\texttt{=0}\}$$

$$[\texttt{c}_\texttt{j}\texttt{==1 \&\& c}_\texttt{k}\texttt{==0}] \ \texttt{act}_\texttt{i}\texttt{\_s} \ \{\texttt{c}_\texttt{p}\texttt{=1; a}_\texttt{i}\texttt{=1}\}$$

Besides, we require `N` times running of an action eventually leads to the status success and use $\texttt{n}_\texttt{i}$ to record the number of running times an action has kept. Therefore, the events above will be additionally decorated by

$$[(\texttt{c}_\texttt{j}\texttt{==1 \&\& c}_\texttt{k}\texttt{==0}) \ \texttt{\&\&} \ \texttt{n}_\texttt{i} < \texttt{N}] \ \texttt{act}_\texttt{i}\texttt{\_r} \ \{(\texttt{c}_\texttt{p}\texttt{=0; a}_\texttt{i}\texttt{=0}) \ \texttt{; n}_\texttt{i}\texttt{++}\}$$

$$[(\texttt{c}_\texttt{j}\texttt{==0 || c}_\texttt{k}\texttt{==1}) \ \texttt{\&\&} \ \texttt{n}_\texttt{i}\texttt{==0}] \ \texttt{act}_\texttt{i}\texttt{\_f} \ \{(\texttt{c}_\texttt{p}\texttt{=0; a}_\texttt{i}\texttt{=0})\}$$

$$[(\texttt{c}_\texttt{j}\texttt{==1 \&\& c}_\texttt{k}\texttt{==0}) \ \texttt{\&\&} \ \texttt{n}_\texttt{i} \leq \texttt{N}] \ \texttt{act}_\texttt{i}\texttt{\_s} \ \{(\texttt{c}_\texttt{p}\texttt{=1; a}_\texttt{i}\texttt{=1}) \ \texttt{; n}_\texttt{i}\texttt{=0}\}$$

After the function embedding for each leaf node, we next focus on the structure building for control nodes.

Structure Building We consider more complex BTs that contain control nodes (Fallbacks or Sequences) other than the single node. We construct CSP model in a bottom-up manner by considering the functionality of control nodes described in Section 2.1. We take the BT in Figure 1 as an example for the whole modelling process. We first construct the model for each leaf node as shown below and ignore those details of `Guard` and `Program` for clarity.

$$\begin{aligned}
\texttt{BT}_1 &= ([...]\ \texttt{picked\_f}\ \{...\} \to \texttt{BT}_1) \;\Box\; ([...]\ \texttt{picked\_s}\ \{...\} \to \texttt{BT}_1)\\
\texttt{BT}_2 &= ([...]\ \texttt{atpos\_f}\ \{...\} \to \texttt{BT}_2) \;\Box\; ([...]\ \texttt{atpos\_s}\ \{...\} \to \texttt{BT}_2)\\
\texttt{BT}_3 &= ([...]\ \texttt{gotopos\_r}\ \{...\} \to \texttt{BT}_3) \;\Box\; ([...]\ \texttt{gotopos\_f}\ \{...\} \to \texttt{BT}_3)\\
&\quad \Box\; ([...]\ \texttt{gotopos\_s}\ \{...\} \to \texttt{BT}_3)\\
\texttt{BT}_4 &= ([...]\ \texttt{pickup\_r}\ \{...\} \to \texttt{BT}_4) \;\Box\; ([...]\ \texttt{pickup\_f}\ \{...\} \to \texttt{BT}_4)\\
&\quad \Box\; ([...]\ \texttt{pickup\_s}\ \{...\} \to \texttt{BT}_4)
\end{aligned}$$

Next, we consider the node `?` associated with `AtPos` and `GotoPos`, which requires that the execution of `GotoPos` only happened when `AtPos` doesn't hold. Therefore, we deconstruct $\texttt{BT}_3$ and compose it with $\texttt{BT}_2$ to reconstruct a new model $\texttt{BT}_{23}$. $\texttt{BT}_{23}$ consists with the modelling shown in line 6-11 of Figure 4.

$$\begin{aligned}
\texttt{BT}_{23} = ([...]\ \texttt{atpos\_f}\ \{...\} \to (\ &([...]\ \texttt{gotopos\_r}\ \{...\} \to \texttt{BT}_{23})\\
&\Box\; ([...]\ \texttt{gotopos\_f}\ \{...\} \to \texttt{BT}_{23})\\
&\Box\; ([...]\ \texttt{gotopos\_s}\ \{...\} \to \texttt{BT}_{23}))\\
)\;\Box\; ([...]\ \texttt{atpos\_s}\ \{...\} \to \texttt{BT}_{23})&
\end{aligned}$$

After that, we consider the node $\to$ associated with the left subtree and `Pickup`. It required that the execution of `Pickup` only happened under two cases: (1) `atpos_s` of the left subtree taken (corresponded to line 21-26 of Figure 4); (2) both `atpos_f` and `gotopos_s` of the left subtree taken (corresponded to line 11-17). The model's construction process is similar to the previous one described. Finally, we tackle with the modelling of the node `?` associated with `Picked` and the right subtree based on the functionality of Fallbacks, as shown in Figure 4. Intuitively, we automatically construct CSP model in a bottom-up manner based on the type of control nodes and the event each child nodes takes.

Verification After the scene-customized function embedding for leaf nodes and the structure building for control nodes, we construct a CSP model from the given BT. Then, we can verify the correctness of CSP model *w.r.t.* specifications (line 33, *i.e.* the robot finally picks up a cube) by the verifier PAT [24]. The final verification result shows this CSP model is truly valid *w.r.t.* specifications, which implies the behavior of BT satisfies such specification. The verifier can also provide a counter-example trace if the final verification result shows *invalid*. For example, let the specification be that the robot never picks up a cube, *i.e.*, $\mathcal{G}\ \neg$`pickup_s`, the verifier may return a counter-example trace like `picked_f`; `atpos_f`; `gotopos_s`; `pickup_s`. The counter-example provided by the verifier will be useful in the following search process.

4.2 Grammar-based MCTS

We instantiate the grammar-based search as a Monte Carlo Tree Search (MCTS) process. Starting from the non-terminal symbol `Root` as the initial candidate BT, MCTS consists of four phases:

- selection phase: it selects the most promising candidate BT based on the current exploration.
- expansion phase: it expands the selected BT based on the given grammar shown in Figure 2 to generate more candidate BTs.
- simulation phase: it evaluates the candidate BT based on the feedback provided by the verifier.
- backpropagation phase: it updates the information of BTs that have been explored. After backpropagation phase, a new round of search begins.

The whole process is presented in Figure 5. The search process will repeat until the expected BT has been found or the time is running out.

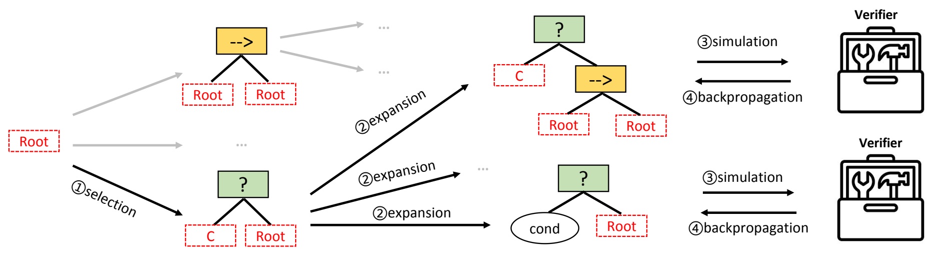

Fig. 5. Grammar-based MCTS.

Instead of applying simulators to do the simulation phase, our method utilizes the verifier to evaluate the BT and provides feedback. The main consideration is that, the feedback provided by robotic simulators is not timely enough since the dynamic interaction and reaction with environments is time-consuming. Compared with robotic simulators, the verifier can provide timely feedback in a static manner without the interaction. Therefore, we invoke the verifier to check the candidate BT and calculate its value as shown in Figure 6.

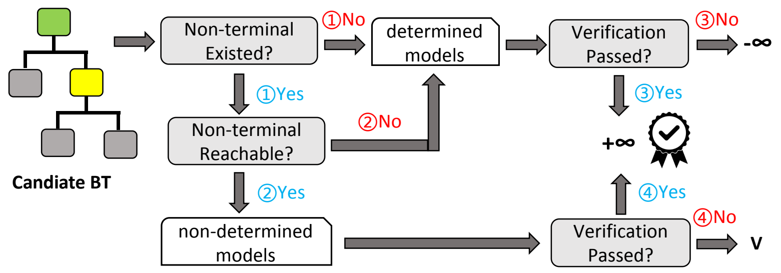

Fig. 6. The evaluation of candidate BTs.

We first classify candidate BTs' models into two categories: determined one and non-determined one, based on the existence and reachability of non-terminals. After that, we invoke the verifier to check the model's correctness *w.r.t.* specifications and assign different values for candidate BTs according to the verification result: (1) the value is -∞ when the determined model failed to pass the verification; (2) the value is calculated as `v` when the non-determined model failed to pass the verification; (3) the value is ∞ when the model passed the verification, which means the expected BT is found. We next describe them in detail.

Type Classification During the grammar-based search process, we may get plenty of candidate BTs with or without non-terminals. For the candidate BT without non-terminals, it's undoubtedly classified into the category of determined models; while for the candidate BT with non-terminals, its category depends on the reachability of non-terminals.

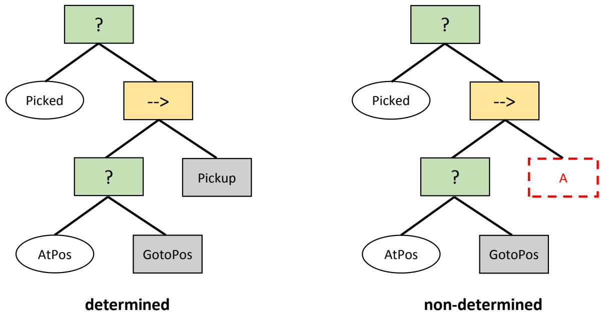

Fig. 7. The candidate BTs.

For example, the BT in Figure 1 might be derived from a candidate BT with non-terminals that only has difference in the rightmost as shown in Figure 7, wherein `A` is a non-terminal symbol. For modelling candidate BTs with non-terminals, we treat all the non-terminal symbols as the symbol `Unknown`, which represents the behavior of this node is unknown and full of possibility. Then, the corresponding CSP model can be obtained by replacing the whole part of `Pickup` in Figure 4 with `Unknown` $\rightarrow$ `BT` (line 19 and line 27 of Figure 4). Intuitively, the reachable of `Unknown` in CSP model, *i.e.*, $\mathcal{G}$ $\neg$ `Unknown` is not valid, means there exist the possibility to satisfy any specifications, even it doesn't yet. Conversely, the unreachable means that the behavior of BT has been determined. Therefore, we classify those candidate BTs with non-terminals into different categories based on the reachability checking.

The verification for determined models For the model without `Unknown`, or the model wherein `Unknown` is unreachable, we invoke the verifier to check whether it satisfies specifications. If the verifier returns *valid*, then we found the expected BT and stopped the search; otherwise, the value of this model's corresponding candidate BT is set as -∞, which implies this candidate BT will never be explored again, and we abandon it to prune the search space.

The verification for non-determined models For the model wherein `Unknown` is reachable, we also invoke the verifier to check the model's correctness *w.r.t.* specification φ. If the verifier returns *valid*, then we found the expected BT and stopped the search; otherwise, we got a counter-example like $\texttt{event}_1$; $\texttt{event}_2$; $\texttt{...}$; $\texttt{event}_N$. Next we evaluate the value in the following four aspects, wherein the first two focus on the BT itself, the last two focus on the specification.

- $\mathbf{V}_1$: we evaluated the ratio of terminal symbols in the candidate BT to investigate the model's determinacy.
- $\mathbf{V}_2$: we evaluated the expansion way that results in the current candidate BT to investigate its influence.
- $\mathbf{V}_3$: we evaluated the relevance between the specification and the candidate BT from the perspective of literal comparison.
- $\mathbf{V}_4$: we evaluated the relevance between the specification and the candidate BT from the perspective of verification result.

Given the action nodes {`GotoPos`, `Pickup`}, the condition nodes {`Picked`, `AtPos`}, the function descriptions shown in Table 1, and the specification $\varphi = \mathcal{F}$ `picked_s`, we take the candidate BT with non-terminals in Figure 7 as an example. We depict those values one by one: $\mathbf{V}_1$ is calculated as the ratio of the already existing terminals to all symbols in the BT like $\frac{3}{4} = 0.75$, while $\mathbf{V}_2$ evaluates the influence of this BT's expansion manners in the expansion phase, which can be classified into three different cases:

- **key terminal expansion**: the BT is expanded based on a non-terminal to terminal rules, wherein the new terminal symbol is related to the existing terminals *w.r.t.* nodes' function. For example, the case that (`? AtPos GotoPos`) is derived from (`? AtPos A`), wherein `AtPos` is entangled in the function of `GotoPos` as shown in Table 1.
- **non-terminal expansion**: the BT is expanded based on a non-terminal to non-terminal rules like `Root` $\rightarrow$ `A`.
- **other terminal expansion**: the BT is expanded based on a non-terminal to terminal rules that the new symbol is not related to the existing terminals *w.r.t.* nodes' function.

We prefer **key terminal expansion**, followed by **non-terminal expansion**, and finally **other terminal expansion**. The $\mathbf{V}_2$ value of three cases is set as 0.9, 0.6, 0.3, respectively.

As for the other two values, the literal relevance value $\mathbf{V}_3$ is calculated as the proportion of the current occurring terminals in φ like $\frac{1}{1} = 1$ (here φ only contains `Picked` and `Picked` exists); while the verification relevance value $\mathbf{V}_4$ measures the complexity of counter-example. It's worth noting that the longer the counter-example trace generated, the closer the behavior of the candidate BT may be likely to φ. For convenience, we project the length of counter-example ($\texttt{event}_1$; $\texttt{event}_2$; $\texttt{...}$; $\texttt{event}_N$) to a value by the formula $\mathbf{V}_4 = \ln N/(\ln N + 2)$. The final value $\mathbf{V}$ is the sum of those four values. Based on the value feedback, we continue advance the later backpropagation phase and search in a new round.

Optimizations To improve the efficiency, we optimize the search process in two aspects. The first one is to make MCTS parallelizing. Note that, for the candidate BTs collected by the expansion phase, we usually do the simulation phase for each candidate BT at a time. However, the simulation phase can be parallelizing. Here we take a leaf parallelization method [23], which invokes multiple threads to deal with those candidate BTs generated by the expansion phase in parallel, then collects all simulation values to propagate backwards through the tree by one single thread. This parallelization can effectively reduce the time required for the simulation phase. The second one is to utilize the nodes' function to make an early checking for the candidate BTs before invoking the verifier. For example, the candidate BT (`? Picked` ($\rightarrow$ `AtPos GotoPos`) `Root`) can be pruned in advance although the non-terminal `Root` is reachable. The reason is that the success of `AtPos` relies on `GotoPos` under the function descriptions shown in Table 1. However, in this case, the failure of `AtPos` will skip the execution of `AtPos`, which makes the status of `AtPos` to be always failure. Therefore, we can deduct that there exists a redundant structure in view of BT's execution and we can prune it for the simplicity of the expected synthesized BT. This pruning can effectively reduce the search space without invoking the verifier.

5 Demonstration

We have implemented our method as a prototype in Python and applied it on several representative robotic missions. To demonstrate its effectiveness and efficiency, we have conducted the following experiments: (1) the comparison experiment between our framework (`MCTS with verifier`) and the framework instantiated as MCTS with a simple simulator used by [16] (`MCTS with simulator`); (2) the ablation experiment for the value designed in Section 4.2.

5.1 Experimental Setup

We collect several representative robotic missions which are shown in Table 3, wherein the first column shows the names of missions and the second column gives a short description for missions. The detailed information about those missions is provided in the website[4]. The time threshold for synthesis is 1 hour. All the experiments were carried out on a machine with an Intel Core i9 processor (3.6 GHz, 8 cores) and 8GB of RAM, and the operating system is Ubuntu 22.04.

5.2 Comparison Experiment

Case Study We take the mission **Alarm** as an example. The mission requires the robot to react with the unpredictable environment factor `Alarm`. The robot may navigate to the position A (`GotoA`) to complete TaskA (`DoTaskA`) if the alarm occurs or navigate to the position B (`GotoB`) to complete TaskB (`DoTaskB`) otherwise. Given the action nodes {`GotoA`, `GotoB`, `DoTaskA`, `DoTaskB`} and the condition ndoes {`Alarm`, `AtA`, `AtB`, `TaskFinishedA`, `TaskFinishedB`}, the semantics for each action node is provided in Table 2.

[4] https://github.com/FM4BT/Synthesizer4BT

Table 2. Nodes' function for mission Alarm.

Action	Requirement	Result
`GotoA`	\	`AtA`, ¬`AtB`
`GotoB`	\	¬`AtA`, `AtB`
`DoTaskA`	`AtA`, `Alarm`, ¬`TaskFinishedA`	`TaskFinishedA`
`DoTaskB`	`AtB`, ¬`Alarm`, ¬`TaskFinishedB`	`TaskFinishedB`

The specification φ is presented as follows. The first part declares the existence of an alarm and specifies that the robot needs to complete at least one of the two tasks. The second part specifies that whenever the alarm occurs, the robot is forbidden to complete TaskB until the alarm frees. The third one is similar.

$$\begin{aligned} &\mathcal{F}\ (\texttt{Alarm\_s}\ \vee\ \texttt{Alarm\_f})\ \ \wedge\ \ \mathcal{F}\ (\texttt{DoTaskA\_s}\ \vee\ \texttt{DoTaskB\_s}) \\ \wedge\ &\mathcal{G}\ (\texttt{Alarm\_s} \rightarrow ((\neg\texttt{DoTaskB\_s}\ \mathcal{U}\ \texttt{Alarm\_f})\ \vee\ \mathcal{G}\ \neg\texttt{DoTaskB\_s})) \\ \wedge\ &\mathcal{G}\ (\texttt{Alarm\_f} \rightarrow ((\neg\texttt{DoTaskA\_s}\ \mathcal{U}\ \texttt{Alarm\_s})\ \vee\ \mathcal{G}\ \neg\texttt{DoTaskA\_s})) \end{aligned}$$

Our method (`MCTS with verifier`) successfully synthesized the expected BT as shown in the left of Figure 8, while `MCTS with simulator` didn't. After an hour of learning, it obtained the synthesized BT as presented in the right of Figure 8. The result BT failed to complete the mission.

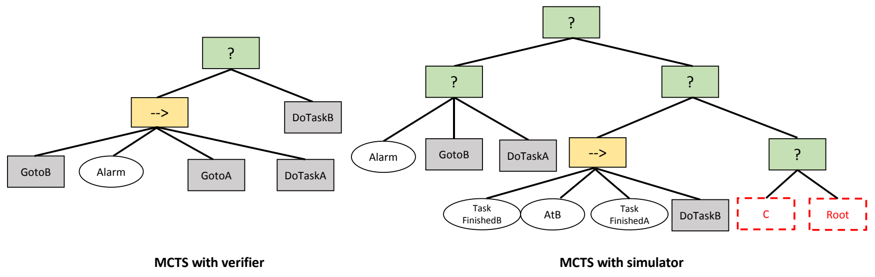

Fig. 8. The synthesis result for the mission **Alarm**.

Overall Results The full experimental results are presented in Table 3. The third column records the time-cost of our method (`MCTS with verifier`) to synthesize the expected BT (☑) for each mission. The fifth column represents that no expected BT is synthesized by `MCTS with simulator` in one hour (✖ 3600s). The latter lacks the evaluation and pruning of non-determined models, as well as the timely feedback from simulators. The detailed synthesis information can be found in the aforementioned website. Note that, compared with the other missions, the significant time overhead increase for the mission **Alarm** is mainly due to the difficulty of determining the position where the node `Alarm` should locate. The forth column represents the number of pruned candidate BTs by the verifier, and the corresponding proportion of these pruned ones in the total verified ones is labeled. There is about 11.6% of the candidate BTs are pruned in average, whose number of leaf nodes is mostly no more than 3. The result implies that we avoid a plenty of meaningless expansions in the early stage.

Table 3. The description and experimental result of missions.

Mission	Description	`MCTS with verifier`		`MCTS with simulator`
		Result	#Pruned	Result
Charge	recharge when the battery is low	✓ 174s	79(15.3%)	✗ 3600s
Patrol$_1$	visit pos$_A$, pos$_B$, pos$_C$ without an order	✓ 124s	57(14.8%)	✗ 3600s
Patrol$_2$	visit pos$_A$, pos$_B$, pos$_C$ in order	✓ 182s	78(14.0%)	✗ 3600s
Pickup	pick up a cube from pos$_A$ and place it at pos$_B$	✓ 1102s	313(9.1%)	✗ 3600s
Alarm	do different tasks depending on the status of alarm	✓ 2535s	353(4.9%)	✗ 3600s

5.3 Ablation Experiment

Besides, we also investigate the rationality of the value design ($\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$ and $\mathbf{V}_4$) in Section 4.2. By disabling the four values individually, we found that the final synthesis time-cost increased to some extent, or even timed out in 16 cases out of 20, as shown in Figure 9. The result implies that the rationality of the value design in guiding the search process.

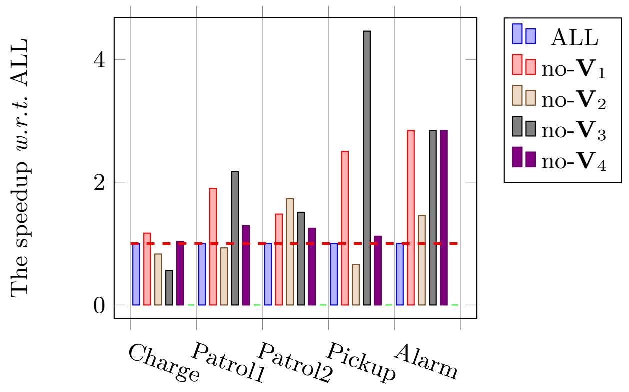

Fig. 9. The speedup *w.r.t.* ALL for each mission and ALL includes all four values.

5.4 Discussion

We are currently primarily focused on BTs composed of only *action*, *condition*, *sequences*, and *fallbacks* nodes. The main bottleneck of our method lies in two aspects. (1) How to design the suitable formal specification to depict the behavior of BTs? This problem can be relieved by utilizing large language models to translate natural language to temporal logics as [9] does. This is not a focal point of our work. (2) How to improve the efficiency of finding the expected BT? This problem can be relieved by designing more effective heuristic search strategy, which is better to customize the heuristic search based on the specific scenario. In this work, we design a general search strategy and demonstrated its effectiveness in the experiment.

6 Related Work

There are many works dedicated to automatically designing and constructing BTs [15]. For example, QL-BT [10] applied reinforcement learning (RL) methods to decide the child nodes' execution order and further optimize early behavior tree prototypes. Banerjee [2] used RL methods to obtain control policies, then converted it into the expected BTs, while Scheide *et al.* [29] utilized Monte Carlo DAG Search to learn BTs based on the formal grammar. Besides, the evolution-inspired learning is another choice for synthesizing BTs [18, 25, 21, 16], which evolves BTs by the generic programming. However, for the above methods, the burden of simulation time for learning and evolving are usually intractable and the synthesized BT just tends to rather than guarantees to meet the specification.

Apart from those learning-based synthesis methods, there also exist some planning-based ones. Colledanchise *et al.* [5] and Cai *et al.* [4] synthesized BTs based on the idea of back chaining, which iteratively extended the action to meet the goal condition. Starting from the formal specifications, Tumova *et al.* [32] constructed an I/O automaton that is the maximally satisfying discrete control strategy *w.r.t.* State/Event Linear Temporal Logic, then converted it into BTs. Colledanchise *et al.* [6] and Tadewos [31] *et al.* taken a divide-and-conquer way to synthesize BTs whose missions are expressed in Fragmented-Linear Temporal Logic. For the method's effectiveness, the expressiveness of those synthesized BTs is usually sacrificed by the limited form of specifications. Compared with that, this work does not impose any restrictions on the specification form.

As regards the verification method for BTs, Biggar *et al.* [3] provided a framework for checking whether the given BT's behavior satisfies LTL specifications. Henn *et al.* [12] utilized Linear Constrained Horn Clauses to verify the BT's safety properties. Serbinowski *et al.* [30] translated BTs into nuXmv models for verification. In this work, we model the behavior of BTs as CSP models and utilize the verifier to check its correctness.

7 Conclusion

In this paper, we proposed a formal verification based synthesis method to automatically construct BTs, which combines Monte Carlo Tree Search with a CSP modelling and verification method. In this method, we innovatively utilized the verifier to complete the simulation phase in MCTS and make the search space pruning based on verification results. The application of our method on several representative robotic missions indicates its promising.

The future work lies in several directions: (1) further exploiting the counter-example traces provided by the verifier, like analyzing the invalidness reason, to guide the search and facilitate the pruning; (2) supporting the modelling for more control node types, like *Parallel*, *Decorator*, *Memorized Sequences*, *Memorized Fallbacks*, and so on; (3) utilizing the concurrence feature of CSP models to verify the robotic mission involved with multi-BTs.

Acknowledgement.

This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

References

1. Baier, C., Katoen, J.: Principles of model checking (2008)
2. Banerjee, B.: Autonomous acquisition of behavior trees for robot control. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018. pp. 3460–3467. IEEE (2018)
3. Biggar, O., Zamani, M.: A framework for formal verification of behavior trees with linear temporal logic. IEEE Robotics Autom. Lett. **5**(2), 2341–2348 (2020)
4. Cai, Z., Li, M., Huang, W., Yang, W.: BT expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event. pp. 6058–6065 (2021)
5. Colledanchise, M., Almeida, D., Ögren, P.: Towards blended reactive planning and acting using behavior trees. In: International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019. pp. 8839–8845. IEEE (2019)
6. Colledanchise, M., Murray, R.M., Ögren, P.: Synthesis of correct-by-construction behavior trees. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017. pp. 6039–6046. IEEE (2017)
7. Colledanchise, M., Ögren, P.: How behavior trees modularize robustness and safety in hybrid systems. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014. pp. 1482–1488. IEEE (2014)
8. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: an introduction. CoRR **abs/1709.00084** (2017)
9. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 383–396. Springer (2023)
10. Dey, R., Child, C.: QL-BT: enhancing behaviour tree design and implementation with q-learning. In: 2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013. pp. 1–8. IEEE (2013)
11. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
12. Henn, T., Völker, M., Kowalewski, S., Trinh, M., Petrovic, O., Brecher, C.: Verification of behavior trees using linear constrained horn clauses. In: 27th Formal Methods for Industrial Critical Systems, FMICS 2022, Warsaw, Poland, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13487, pp. 211–225 (2022)
13. Hoare, C.A.R.: Communicating Sequential Processes (1985)

14. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition, Addison-Wesley (2007)
15. Iovino, M., Scukins, E., Styrud, J., Ögren, P., Smith, C.: A survey of behavior trees in robotics and AI. Robotics Auton. Syst. **154**, 104096 (2022)
16. Iovino, M., Styrud, J., Falco, P., Smith, C.: Learning behavior trees with genetic programming in unpredictable environments. In: IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021. pp. 4591–4597. IEEE (2021)
17. Lan, M., Xu, Y., Lai, S., Chen, B.M.: A modular mission management system for micro aerial vehicles. In: 14th IEEE International Conference on Control and Automation, ICCA 2018, Anchorage, AK, USA, June 12-15, 2018. pp. 293–299. IEEE (2018)
18. Lim, C., Baumgarten, R., Colton, S.: Evolving behaviour trees for the commercial game DEFCON. In: Applications of Evolutionary Computation, Istanbul, Turkey, April 7-9, 2010, Proceedings. Lecture Notes in Computer Science, vol. 6024, pp. 100–110 (2010)
19. Marzinotto, A., Colledanchise, M., Smith, C., Ögren, P.: Towards a unified behavior trees framework for robot control. In: 2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014. pp. 5420–5427 (2014)
20. Mateas, M., Stern, A.: A behavior language for story-based believable agents. IEEE Intell. Syst. **17**(4), 39–47 (2002)
21. Neupane, A., Goodrich, M.A.: Learning swarm behaviors using grammatical evolution and behavior trees. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 513–520 (2019)
22. Nilsson, N.J.: Teleo-reactive programs for agent control. J. Artif. Intell. Res. **1**, 139–158 (1994)
23. Papakonstantinou, G.K., Andronikos, T., Drositis, I.: On parallelization of UET / UET-UCT loops. Neural Parallel Sci. Comput. **9**(3-4), 279–318 (2001)
24. PAT Website: http://pat.comp.nus.edu.sg
25. Perez Liebana, D., Nicolau, M., O'Neill, M., Brabazon, A.: Evolving behaviour trees for the mario AI competition using grammatical evolution. In: Applications of Evolutionary Computation - EvoApplications 2011, Torino, Italy, April 27-29, 2011. Lecture Notes in Computer Science, vol. 6624, pp. 123–132 (2011)
26. Puga, G.F., Gómez-Martín, M.A., Gómez-Martín, P.P., Díaz-Agudo, B., González-Calero, P.A.: Query-enabled behavior trees. IEEE Trans. Comput. Intell. AI Games **1**(4), 298–308 (2009)
27. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
28. Ramírez, M., Papasimeon, M., Lipovetzky, N., Benke, L., Miller, T., Pearce, A.R., Scala, E., Zamani, M.: Integrated hybrid planning and programmed control for real time UAV maneuvering. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018. pp. 1318–1326 (2018)
29. Scheide, E., Best, G., Hollinger, G.A.: Behavior tree learning for robotic task planning through monte carlo DAG search over a formal grammar. In: IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, 2021. pp. 4837–4843 (2021)

30. Serbinowski, B., Johnson, T.T.: Behaverify: Verifying temporal logic specifications for behavior trees. In: 20th Software Engineering and Formal Methods, SEFM 2022, Berlin, Germany, 2022. Lecture Notes in Computer Science, vol. 13550, pp. 307–323 (2022)
31. Tadewos, T.G., Newaz, A.A.R., Karimoddini, A.: Specification-guided behavior tree synthesis and execution for coordination of autonomous systems. Expert Syst. Appl. **201**, 117022 (2022)
32. Tumova, J., Marzinotto, A., Dimarogonas, D.V., Kragic, D.: Maximally satisfying LTL action planning. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014. pp. 1503–1510. IEEE (2014)