EffBT: An Effective Behavior Tree Reactive Synthesis and Execution Framework

Anonymous Author(s)*

ABSTRACT

Behavior Trees originate from the control of Non-Player-Characters (NPCs) and have been widely embraced in robotics because of their advantages in modularity, reactivity, and other features. Research on synthesizing BTs automatically has been strongly promoted. Reactive synthesis is a method that obtains correct-by-construct controllers from formal specifications. However, model-checking-based reactive synthesis suffers an EXPTIME complexity and Fragment-LTL-based (F-LTL) synthesis needs to calculate several individual functions for each formula. Hence, we propose EffBT, an effective behavior tree synthesis and execution framework, which also adopts the synthesis based on F-LTL, but we unify function calculations into GR(1) realizability check. Benefiting from this, we could optimize the execution efficiency of obtained BTs at the design phase by efficient structure design and the adoption of *Parallel*, while none of the prior works focused on this. We proved the soundness of EffBT and later experiments also demonstrated its effectiveness.

CCS CONCEPTS

• **General and reference** → **General conference proceedings**; • **Software and its engineering** → **Formal methods**; • **Computing methodologies** → *Robotic planning*.

KEYWORDS

Behavior Tree, Reactive Synthesis, GR(1), Effective Execution

ACM Reference Format:

Anonymous Author(s). 2024. EffBT: An Effective Behavior Tree Reactive Synthesis and Execution Framework. In *Proceedings of 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Behavior Trees (BTs) originate from the video game industry as the controller of Non-Player-Characters (NPCs) and have been widely used in robotics because of their advantages in modularity, reactivity, and other features. As the robotic system becomes more and more complex, the most desirable goal for developers is to generate BTs automatically. In the past decades, researchers have worked a lot to search for solutions, lots of methods, including evolution algorithms [12][16], reinforcement learning [8], task and motion planning [14], Large Language Models [5][13], and learning from demonstrations [17] [9], have been used to generate BTs for solving a variety of tasks. In addition, reactive synthesis, which obtains correct-by-construct representations (e.g., automaton) from formal specifications (e.g., Linear Temporal Logic (LTL)) automatically, has also been used in BTs' synthesis. Compared with those informal methods, reactive synthesis has the grant advantage in that its result follows the given formal specifications rigorously.

There are two main categories in terms of reactive synthesis for BTs. The first, such as TAMP[14], utilizes a model-checking-based strategy and suggests synthesizing BTs from an emptiness-checking procedure over product automata, which is the production of the Büchi automata (converted from LTL specifications) and a transition system (describes the robot and environment). If the specification is realizable, the procedure identifies a counter-example path (i.e., state transitions), and then constructs BTs from this path. Nevertheless, the conversion from LTL to Büchi automata exhibits exponential time complexity, implying that the running time increases exponentially with the size of the formula. Consequently, BT synthesis approaches relying on this strategy inherently face the challenge of high computational complexity (at least EXPTIME).

To reduce complexity, researchers propose to obtain BTs from fragments of LTL instead of the general LTL, forming the second strategy. In prior works [6][19], the authors construct formulas on LTL fragments and then obtain transition relations by calculating value functions, requirement functions, and constraint functions for each formula. Subsequently, they construct BTs from those functions. Our approach also embraces the second strategy but diverges significantly from previous efforts. Specifically, we calculate transition relations through a unified process, namely the realizability check of GR(1) [2], whereas earlier methods necessitate individual function calculations for each formula. Benefiting from this, we could design efficient structures and optimize the execution efficiency of the resulting BTs, such as adopting *Parallel* nodes, yet none of the prior approaches have considered this aspect.

In this work, we propose EffBT, an efficient behavior tree synthesis and execution framework. Given the game structure GS and GR(1) specifications φ as input, EffBT generates an executable BT that satisfies predefined objectives, in which GS encapsulates the dynamic properties of systems and environments while φ describes their objects. Our framework comprises two primary procedures, Step 1 and Step 2′, as depicted in Fig. 1 (follows the **green arrow**). In the first step, we check the realizability of given specifications and construct two intermediate arrays mX[][][] and mY[][], which are similar to the operations in [2]. This checking process is formulated as a two-player game (GS, φ) and then solved by μ-calculus. Following this, Step 2′ employs a construction algorithm we propose to generate BTs from the computed mX[][][] and mY[][] arrays. Besides, we further refine the structure of subtrees and incorporate *Parallel* nodes based on the propositions in Sec. 3.2. This modification significantly enhances BT execution efficiency by reducing decision-making time.

ASE '24, Oct 27–Nov 01, 2024, California, US

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

By the way, tracing the **brown arrow** in Fig. 1, there is an alternative method to synthesize BTs from GR(1) as the following steps (referred as the NaiveBT method in this paper): 1. conduct realizability check over (GS, φ) and get mX[][][] and mY[][]; 2. construct the Fair Discrete System (FDS) from the intermediate results [2]; 3. since FDS is stored as a Binary Decision Digraph (BDD), transform it to the If-Then-Else (ITE) structure [3]; 4. at the last step, a BT could be constructed easily because BTs have sufficient capacity to represent the If-Then-Else logic. However, empirical studies in Sec. 2.3 reveal numerous shortcomings associated with this NaiveBT approach.

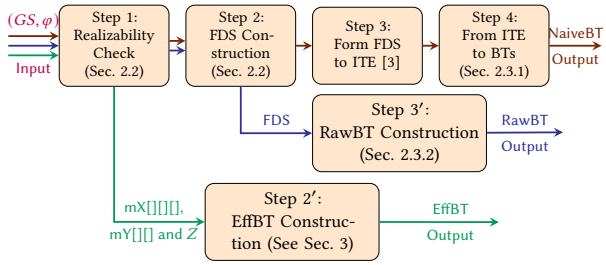


Figure 1: Procedure Overview of the NaiveBT (Brown Arrow), RawBT (Blue Arrow), and EffBT (Green Arrow) Methods

Aside from the NaiveBT, another method for synthesizing BTs from GR(1) specifications involves initially transforming the specifications into an automaton, followed by generating BTs from this automaton. Nevertheless, BTs are inherently action-based while automata are state-based, making it challenging to identify meaningful actions during the construction. This difficulty arises due to the complexity of states and transitions within the automaton.

The primary contributions of this work could be summarized as follows:

- We propose EffBT, an efficient behavior tree synthesis and execution framework that incorporates a unified realizability check and transition calculation process.
- We concentrate on optimizing the execution efficiency of synthesized BTs through efficient structure design and adoption of *Parallel* nodes.
- We have proven the soundness of our method and substantiated its effectiveness through experimental evaluations conducted on several distinct scenarios.

The remainder of this paper is organized as follows: Sec. 2 gives preliminary knowledge of Behavior Trees and GR(1) synthesis. Meanwhile, it introduces the NaiveBT and RawBT methods. Then, Sec. 3 presents our EffBT method in detail, complemented by soundness proof and complexity analysis. Research questions, corresponding experiment configurations, and their outcomes are detailed in Sec. 4. Finally, in Sec. 5 we recall related works and conclude this paper in Sec. 6.

2 PRELIMINARIES AND RUNNING EXAMPLE

2.1 Behavior Trees

Behavior Trees (BTs) are rooted directed trees whose running starts from the root, a *tick* signal is transited to its children following the depth-first sequence. Each node in BTs has three types of status, *Running*, *Success*, and *Failure*, and returns one of the statuses to its parent when being ticked.

The leaf nodes (also known as *Execution* nodes) include *Condition* nodes (as the ellipse nodes in Fig. 2) and *Action* nodes (as the gray box node in Fig. 2). *Condition* nodes have no *Running* status and return *Success* only when the condition stands. *Action* nodes start execution when they are been ticked, return *Success* when actions performs successfully, return *Failure* when actions failed, else return *Running* to their parents.

The non-leaf nodes (also known as *ControlFlow* nodes) include the *Sequence*, the *Selector*, and the *Parallel* nodes (as the marks with an $\rightarrow$ in a green box, an ? in a yellow box, and an $\rightrightarrows$ in a blue box shown in Fig. 2, resp.), whose statuses depend on their type and statuses of their children. When the *Sequence* is ticked, it ticks its children from left to right until any child returns *Failure* then the *Sequence* fail, or all of the children return *Success* in that case *Sequence* succeeds also, or otherwise it returns *Running* (i.e., any one child is running). When the *Selector* is ticked, *tick* signals are routed to its children in the same manner with *Sequence* until any child returns *Success* and the *Selector* succeeds, or all of the children failed which leading to the *Selector* fails, or otherwise return *Running*. Note that *Sequence/Selector* nodes will not pass *tick* signals to the next children (if any) when the previous child returns *Failure/Success* or *Running*. In terms of the *Parallel*, let n and m denote the total number of its children and the threshold defined by users, it ticks all its children simultaneously and returns *Success* if m children return *Success*, return *Failure* if $n - m + 1$ children return *Failure*, and return *Running* in other cases. In this paper, we only apply the *Parallel* node with $m = 1$, which ticks its children in parallel and returns *Success* once a child succeeds.

Usually, when BTs are executed, a *blackboard* is built for sharing variables (e.g., sensor's information) between nodes, which is a dictionary with keys and values, all the nodes in BTs have permission to read from and write to the blackboard.

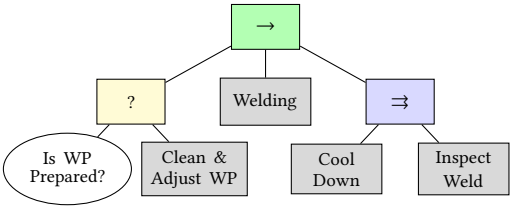


Figure 2: A simplified Work Pieces (WP) Welding BT.

2.2 The GR(1) Synthesis

GR(1) synthesis is devised to solve the problem of automatically synthesizing digital designs from linear-time specifications [2]. This approach takes as input GR(1) formulas and yields correct-by-construction controllers, i.e., Fair Discrete System (FDS). The GR(1) formula is the fragment of LTL with a predefined implicational structure like $asm \rightarrow gar$, where assumptions (asm) denote possible environmental states while guarantees (gar) describe the system's reactions to those states. Specifically, it contains initial assumptions and guarantees over initial states (θ^e and θ^s resp.), safety assumptions and guarantees relating the transition from current to next state (ρ^e and ρ^s resp.), and justice assumptions and guarantees requiring that an assertion holds infinitely many times

during a computation (J^e and J^s resp.). The realizability check of GR(1) verifies the following LTL formula:

$$(\theta^e \wedge \mathcal{G}\rho^e \wedge \bigwedge_{i\in\{1..m\}} \mathcal{GF}J_i^e) \rightarrow (\theta^s \wedge \mathcal{G}\rho^s \wedge \bigwedge_{j\in\{1..n\}} \mathcal{GF}J_j^s) \quad (1)$$

2.2.1 Realizability Check of GR(1). Compared to priors, checking realizability and calculating transition policies in our work are unified into the *GR(1) Realizability Check* process rather than individual function calculations for each formula. This process takes as input fragments of LTL and produces a winning set W_s and two intermediate variables mX[][][] and mY[][].

This problem is formulated as a two-player game between the system and environment [2], where the Formula. 1 will be decomposed as a game structure GS and corresponding specification φ. In detail, the initial and transition assertions of the system and environment in GS are θ^s, θ^e and ρ^s, ρ^e respectively, and φ is the formula $\bigwedge_{i\in\{1..m\}} \mathcal{GF}J_i^e \rightarrow \bigwedge_{i\in\{1..n\}} \mathcal{GF}J_j^s$. The specification is realizable if and only if (iff) the system wins in the game. The system wins if a strategy allows it to satisfy all justice guarantees infinitely often while maintaining initial and safety guarantees, or forcing the environment to violate any of its justice assumptions, or its initial or safety assumptions. The system-winning set W_s stores the states from which there exist system-winning strategies. Following that, the GR(1) realizability check is solved by a three-level μ-calculus and stores two intermediate arrays mX[][][] and mY[][], which entails transition relations. Details of this procedure are presented in Sec. 3.1 and the works [2][15] are recommended for further reading about the GR(1) formula and synthesis.

2.2.2 FDS Construction in GR(1). Following the original process of GR(1) synthesis, static FDS controllers will be constructed from mX[][][] and mY[][] immediately after the realizability check. Without the loss of generality, later in this paper, we use the symbol z to represent the system-winning set W_s.

Intuitively, the transition ρ in FDS has three parts ρ_1, ρ_2, and ρ_3, each tailored to handling a separate case. Let the extra variable jx represent the index of a justice goal that the system attempts to accomplish currently. When the current state is a z state, the first part ρ_1 deals with it, and jx marks to the next $jx \oplus 1$, and the next state is from z (e.g., a random state in it). The second part ρ_2 takes efforts when the current state is an mY$[j][r]$ state, then the system moves to mY$[j][r-1]$ that is closer to satisfying J_j^s. When the current is a mX$[j][r][i]$ state, the strategy ρ_3 works and still stays in mX$[j][r][i]$ states. Their formal representations are as follows, where the transition $\rho = \rho_1 \vee \rho_2 \vee \rho_3$.

$$\rho_1 = \bigvee_{j\in\{1..n\}} (jx = j) \wedge z \wedge J_j^s \wedge \rho_e \wedge \rho_s \wedge z' \wedge (jx' = j+1)$$

$$\rho_2 = \bigvee_{j\in[1..n]} (jx = jx' = j) \wedge \rho_2(j)$$

$$\rho_2(j) = \bigvee_{r>1} \mathrm{mY}[j][r] \wedge \neg\mathrm{mY}[j][<r] \wedge \rho_e \wedge \rho_s \wedge \mathrm{mY}'[j][<r]$$

$$\rho_3 = \bigvee_{j\in[1..n]} (jx = jx' = j) \wedge \rho_3(j)$$

$$\rho_3(j) = \bigvee_{r>1} \bigvee_{i\in[1..m]} \mathrm{mX}[j][r][i] \wedge \neg\mathrm{mX}[j][<(r,i)] \wedge \neg J_i^e \wedge \rho_e \wedge \rho_s \wedge \mathrm{mX}'[j][r][i]$$

2.3 Running Example: the NaiveBT and the RawBT Methods

In this section, we briefly review the NaiveBT and the RawBT methods and explain why don't we adopt them for BT synthesis, as we have proved that both of them have inherent efficiency limitations through theoretical analysis (Sec. 2.3.2) and experiments (Sec. 4.3.1).

2.3.1 The NaiveBT Method. The procedure of this method has been introduced before and summarized in Fig. 1, in which the first three steps were proposed in [2] and [3] and already have been realized. Hence, we only introduce the last step about how to construct BTs from ITE by applying regular transition rules here.

As shown in Table. 1, we summarized four regular patterns (RPs) that emerged in the context of ITE and designed corresponding transformation rules for each pattern, where P and P' denote the predicate that describes the current and next states, while BTs are denoted as texts, *Seq/Sel* $\{\ldots\}$ denotes the *Sequence/Selector* node, C^P denotes the *Condition* node with checking condition P, and $A^{P'}$ denotes the *Action* node with action moving to state P'.

For single imply patterns RP1 and RP2 (e.g., "robot at (0,0)" $\rightarrow$ "robot′ at (0,1)"), we construct an equivalent BT by using *Sequence* as root and the *Condition* node ("is the robot at (0,0)") and the *Action* node ("move robot to (0,1)") as its children. Note that RP1 and RP2 correspond to the identical form of SubBT, we have a more detailed explanation on our website [1]. RP3 is the Conjunction Rule, meaning that we use a *Selector* node to represent the conjunction. RP4 is the Default Rule, which means that if a system variable R doesn't occur in an implied sequence (e.g., "robot$_0$ at 0" $\rightarrow$ "robot$_1$ at 0" $\rightarrow$ "robot$_0'$ at 1", while the system variable robot$_1'$ doesn't in this sequence), its value could be random in a legal set $\{r_1, \ldots, r_n\}$.

By applying RP1 to RP4 on ITE recursively, we could finish Step 4 and construct a result BT from ITE. However, later experiments have evidenced that the NaiveBT method performs poorly no matter on synthesis efficiency or the quality of generated BTs.

Table 1: Regular Patterns of the NaiveBT method

Name	Regular Pattern	SubBT
RP1	$P \rightarrow P'$;	*Seq* $\{C^p, A^{P'}\}$
RP2	$P' \rightarrow P$;	*Seq* $\{C^p, A^{P'}\}$
RP3	$P \rightarrow P' \wedge P'$, or $P' \rightarrow P \wedge P$;	*Sel* $\{BT^1_{sub}, BT^2_{sub}\}$
RP4	$P^*(\rightarrow P^*)^*$ and the system variable R not occur in it.	*Seq*^{s} $\{A^{R'=r_1}, \ldots, A^{R'=r_n}\}$

2.3.2 The RawBT Method. Following the **Blue Arrow** in Fig. 1, this method is based on the FDS from GR(1) synthesis. Specifically, it obtains BTs by translating the transition ρ in FDS to BTs (Step 3′). For ρ_1, ρ_2 and ρ_3 in the transition, we formulate their corresponding subtrees by substituting the outermost disjunctions with *Selector* nodes, and conjunct these ρ-subtrees by the *Selector* as root. Regarding the innermost conjunction formulas, we identify and separate the action and condition nodes according to whether the predication describes the current state or the state following it. For example, the minimum part of ρ_1 is $(jx = j) \wedge z \wedge J_j^2 \wedge \rho \wedge z' \wedge (jx' = j+1)$, which corresponds to the condition node $Cond((jx = j) \wedge z \wedge J_j^2 \wedge \rho)$ and the action node $Act(z' \wedge (jx' = j+1))$. Then, we combine them

with a Sequence node and mount the subtree under the Selector. However, after observing the execution of such BTs, we propose the following finding as a proposition and prove it.

Proposition 1. *State transitions in a single sub-strategy (i.e., to reach a particular justice goal) exceed those between different sub-strategies significantly.*

Proof. The system transits from the current sub-strategy to another one iff the current state satisfies one justice guarantee J_j^s (i.e., when the state is in $S_t = \bigcup_{j\in[1..n]}\{s|s \models J_j^s\}$, then the system transits to another sub-strategy). Let $S_{nt} = S/S_t$ denote the set of states that couldn't happen transition over sub-strategies. Obviously, Prop. 1 holds if $|S_t| \ll |S_{nt}|$. On the other hand, if $|S_t| \gg |S_{nt}|$, only one or fewer sub-strategies could handle, thus, transitions in one sub-strategy still more often than those over sub-strategies, Prop. 1 holds. □

Each subtree within the BTs produced by RawBT corresponds to the ρ_1, ρ_2, and ρ_3 transitions. As per Prop. 1, systems predominantly aim to reach a single justice goal. Consequently, during execution, these BTs must frequently tick and switch to find the proper jth subtree in ρ_x (ρ_1, ρ_2, or ρ_3) subtrees, a process which wastes lots of time in decision-making. Although using ControlFlow nodes with memory could release the situation, they hurt the reactivity of BTs. In our methodology, we fixed this drawback by structure optimization and the adoption of *Parallel* nodes.

3 METHODOLOGIES

To overcome those constraints inherent in NaiveBT and RawBT while preserving the reactivity, we propose EffBT, an efficient BT synthesis framework that preserves the modularity and reactivity of BTs and adopts *Parallel* nodes to optimize the execution efficiency of generated BTs. It comprises two phases Step 1 and Step 2′ as shown in Fig. 1. The first step takes as input a game structure and specification as (GS, φ), then calculates two intermediate arrays mX[][][], mY[][], and the winning set W_s (or z) by GR(1) realizability check. Subsequently, Step 2′ constructs a result BT that satisfies φ from them. It's not trivial to construct such BTs since we should both keep the semantic correctness of BTs while optimizing their execution efficiency.

3.1 Realizability Check

Instead of calculating individual value functions, requirement functions, and constraint functions for each formula, our approach unifies them by calculating two intermediate arrays mX[][][] and mY[][] following the realizability check procedure that occurred in GR(1) synthesis [2]. The preliminaries have told that, realizability checking takes as input game structure GS and specification φ, and this problem has been formulated as a two-player game between the system and the environment. Let $\langle\!\langle X$ denote controlled predecessors, from the states in $\langle\!\langle X$ the system can force the environment to visit a state in X. Besides, let μ and ν denote the minimum and the maximum fix-point operator respectively. Thus, to check the realizability and calculate the system-winning set W_s , we should solve its equivalent problem, namely calculate the semantics of the following μ-calculus formula over the game structure:

$$W_s = \nu Z(\bigwedge_{j\in\{1..n\}} \mu Y(\bigvee_{i\in\{1..m\}} \nu X(J_j^s \wedge \langle\!\langle Z \vee \langle\!\langle Y \vee \neg J_i^e \wedge \langle\!\langle X))) \quad (2)$$

Subsequently, this equivalent problem is solved by a three-level fixed point calculation νZ, μY, and νX, where X, Y, and Z are relation variables (equal to *true*, *false* and *true* resp. in the beginning). During the calculation, two crucial intermediate arrays, mX$[j][r][i]$ and mY$[j][r]$, are stored, wherein the union of mX$[j][r][i]$ over i (i.e., $\bigcup_{i\in\{1..m\}}$mX$[j][r][i]$), is equivalent to mY$[j][r]$. Here, $j \in \{1..n\}$, $r \in \{1..k\}$, and $i \in \{1..m\}$, among which the variables n, k, and m represent the counts of system guarantees, the iterations for the minimum fixed point μY, and the counts of environment assumptions respectively.

3.1.1 mX[j][r][i]. It's the fix-point calculation results of the innermost safety game in Eq. (2), i.e., the maximum fixed point calculation νX. Intuitively, from the states in mX$[j][r][i]$, the system could either move one step closer to reaching $J_j^s \wedge \langle\!\langle Z$ states within r transitions or keep forcing the environment to violate the ith environmental assumption J_i^e.

3.1.2 mY[j][r]. This variable stores the fix-point calculation results of the inner least reachability game, i.e., the minimum fixed point calculation μY. From the states in mY$[j][r]$, the system could either move to $J_j^s \wedge \langle\!\langle Z$ states within r steps that satisfy the jth system guarantee or violate at least one environment assumption J_i^e for some i.

In addition, the outermost maximum fix-point νZ ensures that once reaching J_j^s states, the system proceeds to visit states satisfying $J_{j\oplus 1}^s$ and keeps the looping.

3.2 Overview of the EffBT Construction

The pseudo-codes of the BT constriction algorithm in EffBT are presented in Alg. 1 and Alg. 2. This algorithm takes as input the intermediates and results (mX[][][], mY[][], and Z) from GR(1) realizability check, along with an auxiliary variable jx which identifies sub-strategies. Then it generates a result behavior tree BT_{eff} that satisfies φ whose root applies *Parallel* node (c.f. line 1 in Alg. 1). Figure. 3 demonstrates a structural overview of the synthesized BT_{eff}, it has n structurally similar *Sequence* subtrees and the jth one named as $subBT_j$. Intuitively, each subtree corresponds to a sub-strategy that controls the system to satisfy one particular justice guarantee (goal) and we construct n subtrees for all goals.

Guided by the design principles outlined in Prop. 2, we adopt the *Parallel* (m=1) as an alternative to *Selector* to enhance the execution speed of obtained BTs in this paper, such as the *Parallel* root of BT_{eff} in Fig. 3. To our knowledge, none of the prior works optimized the execution efficiency of synthesized BTs during the design phase and none of them utilizes *Parallel* nodes.

Proposition 2. *A BT could adopt the Parallel nodes (m=1) as root iff it satisfies the following two conditions: 1. it's excepted to succeed when any one of its children succeeds; 2. there is no resource competition or temporal dependency among its children.*

The *Parallel* Node (m=1) has similar semantics with *Selector*, as both of them succeed once any one child succeeds. Hence, the first condition in Prop. 2 indicates that being able to use *Selector* is a

necessary condition for using *Parallel* (m=1). However, unlike *Selector* which ticks its children in sequence, the *Parallel* (m=1) ticks its children simultaneously, thus, we supplement the second condition to make sure its children can execute separately and avoid possible conflicts. In terms of performance, the children of *Parallel* execute at the same time and each of them separates according to Prop. 2. Obviously, it would be faster than *Selector* that ticks its children one by one. Later experiments have verified the effectiveness of *Parallel* design in EffBT.

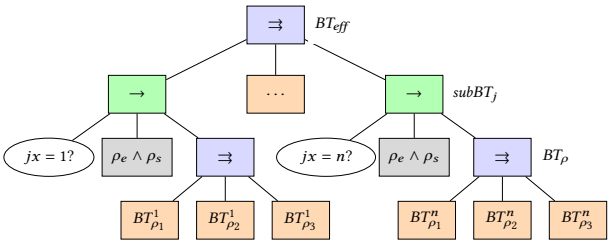


Figure 3: Structure Overview of the BT from EffBT Method

In addition to basic ControlFlow nodes, another four predefined leaf nodes are used during the construction of BTs, including two *Condition* nodes BDDCond and JthCond, and two *Action* nodes CalAct and MovAct. Their meanings are summarized in Table. 2. Note that an extra Action Library is needed in the MovAct node when it needs to control actual movements (e.g., a robot arm in reality) because our method cannot synthesize primitive controllers (e.g., PID controllers).

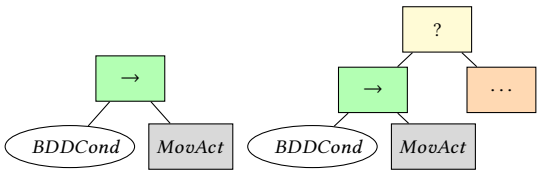
Table 2: Predefined Condition and Action Nodes

Name (Abbr.)	Description
BDDCond	This node assesses whether the current state satisfies the BDD condition, returns *Success* if true, or *Failure* otherwise.
JthCond	This node judges whether the current goal is the jth justice guarantee, yields *Success* if true, or *Failure* otherwise.
CalAct	This action conducts pure calculations only and has no actual movement, return *Failure* when its result is *false* or invalid.
MovAct	This action matches the primitive function in Action Library and conducts action in physical or simulated environments.

3.3 Construction of Subtrees

As demonstrated in Fig. 3, a single subtree $subBT_j$ is composed of three parts: a JthCond node to judge whether the current goal is the jth justice guarantee; a CalAct node to make sure systems and environments satisfy the transition rules $\rho_e \wedge \rho_s$ (c.f. line 4 in Alg. 1); and another *Parallel* subtree BT_ρ with three children to describe the ρ_1, ρ_2, and ρ_3 transitions. Note that ρ transitions here only correspond to a single justice guarantee that differs from those in Sec. 2.2.2. To distinguish, we denote them as BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3}, each of them differs in structure and meanings. The detailed construction procedures are shown in Alg. 1 and Alg. 2. Through the design of decomposed ρ transitions, we categorize transitions for each goal into the same subtree and avoid the decision-making problem in the RawBT method.

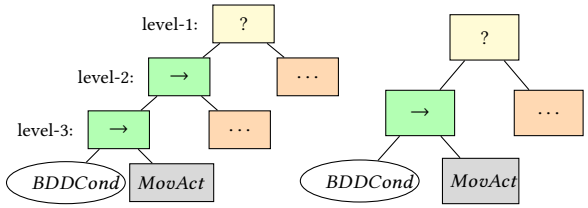
3.3.1 Construction of BT_{ρ_1} and BT_{ρ_2}. As presented in Fig. 4(a), the jth ρ_1 subtree BT_{ρ_1} is constructed as *Sequence* with BDDCond and MovAct as its children (c.f. line 19 and line 20 in Alg. 1). The instanced BDDCond node checks whether the current state satisfies the jth justice goal, if true, next the instanced MovAct node takes action to the $j \oplus 1$ goal. As shown in Fig. 4(b), the BT_{ρ_2} is a *Selector* node with *Sequence* subtrees (denoted as $tree_{\rho_2}$), each subtree has the same structure with BT_{ρ_1} but differs in contents (c.f., line 8 to line 18). Intuitively, the BDDCond under the rth $tree_{\rho_2}$ checks whether the current state satisfies $\mathrm{mY}[j][r]$, if true, then the MovAct node forces the system to $\mathrm{mY}[j][r-1]$ states that one step closer to the jth goal and finally reach the goal.



(a) Structure of BT_{ρ_1} **(b) Structure of BT_{ρ_2}**

Figure 4: The Inner Structure of BT_{ρ_1} and BT_{ρ_2}

3.3.2 Construction of BT_{ρ_3}. The construction of BT_{ρ_3} is presented in Alg. 2, and its structure is shown in Fig. 5(a). It's constructed as a *Selector* with two-level nested *Sequence* subtrees for each r and i (c.f., line 3 to line 20 in Alg. 2). Moreover, for the lowermost subtree, which is also structurally similar to BT_{ρ_1}, its instanced BDDCond node checks whether the current state satisfies $\mathrm{mX}[j][r][i]$, if true, the MovAct force the system stays to keep violating the ith environment assumption.



(a) Structure of BT_{ρ_3} **(b) Pruned (Example)**

Figure 5: The Inner and Pruned Structure of BT_{ρ_3}

3.3.3 Pruning. Note that we prune useless nodes or subtrees during the construction in two cases. The first, when constructing subtrees of BT_{ρ_2} and BT_{ρ_3} (i.e., $tree_{\rho_2}$ and $tree_{\rho_3}$), if the condition is *false* (c.f. line 11 in Alg. 1 and line 7 in Alg. 2), the subtree will be pruned as it will never be executed. The second condition occurs if the ControlFlow node has only one child, we remove the ControlFlow node and reload its child to its parent directly. As shown in Fig. 5(a), if the *Selector* in level-2 only has one child, it will be pruned (c.f., line 8 to line 11 in Alg. 2) and the result is shown in Fig. 5(b). Besides, the second strategy is also applied in the construction of BT_{ρ_1}, BT_{ρ_2}, and the whole tree (c.f., line 7, line 18, line 21 to line 28 in Alg. 1).

3.4 Execution of the BT

In this part, we demonstrate how synthesized BTs are executed. Firstly, an internal blackboard, which is transparent to users, is designed to store running statuses (e.g., the currently active jth subtree, system state, environment state) and intermediate computation results such as BDD calculation results from CalAct nodes.

Algorithm 1 BT Construction of the EffBT Method

Input: mX[][][], mY[][], Z, jx
Output: BT_{eff} — the root node of the result BT

$BT_{eff} \leftarrow$ Parallel Node
for all $j \in \{1, ..., n\}$ **do**
  $subBT_j \leftarrow$ Sequence Node;
  $subBT_j$.addChildren({JthCond($jx = j$?), CalAct($\rho_e \wedge \rho_s$)})
  $BT_\rho \leftarrow$ Parallel Node
  /*Construction of the sub-tree $BT_{\rho_3\_j}$*/
  $BT_{\rho_3\_j} \leftarrow$ ConstructTree$_3$(mX[][][], Z, j)
  BT_ρ.addChildIfNotEmpty($BT_{\rho_3\_j}$)
  /*Construction of the sub-tree $BT_{\rho_2\_j}$*/
  $BT_{\rho_2\_j} \leftarrow$ Parallel Node; $low \leftarrow$ mY[j][0]
  for all $r \in \{2, ..., r_j\}$ **do**
    $cond \leftarrow$ mY[j][r] $\wedge \neg low$
    if $cond \neq$ false **then** ▷ Prune.
      $tree_{\rho_2} \leftarrow$ Sequence.addChild(BDDCond($cond$))
      $tree_{\rho_2}$.addChild(MovAct($low' \wedge jx' = j$))
      BT_{ρ_2}.addChild($tree_{\rho_2}$)
    end if
    $low \leftarrow low \vee$ mY[j][r]
  end for
  BT_ρ.addChildIfNotEmpty($BT_{\rho_2\_j}$)
  /*Construction of the sub-tree $BT_{\rho_1\_j}$*/
  $BT_{\rho_1_j} \leftarrow$Sequence.addChild(BDDCond($jx = j \wedge Z \wedge J_j^s$))
  $BT_{\rho_1\_j}$.addChild(MovAct($Z' \wedge jx' = j + 1$))
  if $subBT_j$.hasNoChild() **then** ▷ Prune.
    $subBT_j \leftarrow BT_{\rho_1}$
  else if $n = 1$ **then**
    $BT_{eff} \leftarrow subBT_1$
  else
    $subBT_j$.addChild(BT_ρ)
    BT_{eff}.addChild($subBT_j$)
  end if
end for

When executing, environment and system states are perceived and recorded on this blackboard, Meanwhile, BTs decide the next steps according to those pieces of information.

As depicted in Fig. 6(a), the root (*Parallel*) receives a *tick* signal in the beginning and then ticks all its children. This signal will be transported to the leftmost child JthCond($jx = j$) in each subtree. When ticked, this condition node queries the inner blackboard for the value of jx, judges whether is currently running at the jth subtree, and only one subtree could return *Success*. Then on the second tick as shown in Fig. 6(b), the only running subtree ticks the next child CalAct($\rho_e \wedge \rho_s$) that calculates the conjunction of the current state, movements of the environment, and $\rho_e \wedge \rho_s$ to judge whether the transitions are legal and writes the intermediate results to the internal blackboard with the key *mediate*.

After this, on the third tick as shown in Fig. 6(c), the last node of the (*Sequence*) subtree is ticked, which then ticks all its children (BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3}) simultaneously. Each of the BT_{ρ_x} subtrees handles different cases as we have introduced in Sec. 3.3 and only one MovAct node will be finally executed. Take the BT_{ρ_1} as an instance, it ticks its child BDDCond to judge whether the current is a z state at first, which calculates conjunctions of the *mediate* in

Algorithm 2 Construction of jth BT_{ρ_3} (*ConstructTree*$_3$() Function)

Input: mX[][][], Z, j
Output: $BT_{\rho_3\_j}$ — the root node of this tree

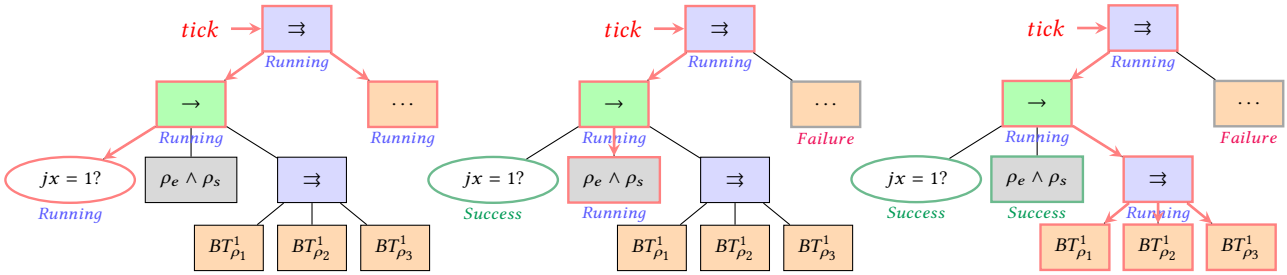
$BT_{\rho_3\_j} \leftarrow$ Selector Node
$low \leftarrow$ false
for all $r \in \{1, ..., r_j\}$ **do**
  $tree_{\rho_3} \leftarrow$ Sequence Node
  for all $i \in \{1, ..., m\}$ **do**
    $cond, act \leftarrow$ mX[j][r][i] $\wedge \neg low$, mX'[j][r][i] $\wedge jx' = j$
    if $cond \neq$ false **then** ▷ Prune.
      if $m = 1$ **then**
        $tree_{\rho_3}$.addChild(BDDCond($cond$))
        $tree_{\rho_3}$.addChild(MovAct(act))
      else
        $sub_{\rho_3} \leftarrow$ Sequence.addChild(BDDCond($cond$))
        sub_{ρ_3}.addChild(MovAct(act))
        $tree_{\rho_3}$.addChild(sub_{ρ_3})
      end if
    end if
    $low \leftarrow low \vee$ mX[j][r][i]
  end for
  BT_{ρ_3}.addChild($tree_{\rho_3}$)
end for
return BT_{ρ_3}

inner blackboard and the BDD in itself, returns *Failure* if calculation result is *false*, or otherwise returns *Success* and updates the *mediate* with the result. If BDDCond succeeds, the MovAct will be executed which takes actions (in Action Library) to make the system state to $z' \wedge jx' = j \oplus 1$. Specifically, it matches an action in the Action Library with to-be-operated system variables and executes that action. Intuitively, the Action Library stores the action name and variables as key and its actual primitive action controller (e.g., PID Controllers for robot arms) as value, which will be matched and loaded for execution dynamically. Note that we stored the auxiliary variable jx in the blackboard and each active subtree updates it to $jx \oplus 1$ upon success, thus, the root (*Parallel*) behaves similarly to ControlFlow nodes with memory, and the entire BT is executed in sequence round-by-round.

3.5 Correctness and Complexity

The EffBT method is sound, implying that whenever given specifications are realizable, it can generate BTs that satisfy specifications strictly. Intuitively, considering that the original GR(1) approach is both sound and complete [2], and the state transitions produced by our BTs constitute a subset thereof, therefore, our method is a sound solution inherently. The formal proof is presented below.

Let *control* denote a state transition that includes a transition guard and its destination, and the *control set* represents a set of all *controls* from one controller (e.g., BT or FDS). Without loss of generality, we define a *control set* calculation function for BT, noted as $ctrl(BT)$. Thus, $C_{eff} = ctrl(BT_{eff})$ denotes the *control set* of BT_{eff}, which also equals $\bigcup_{j \in \{1..n\}} ctrl(subBT_j)$ as BT_{eff} is composed of n subtrees $subBT_j$ and they are separate to each other. For the jth subtree $subBT_j$, its *controls* are composed of three parts from BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3} since only BT_ρ conducts actions and



(a) on the first *tick*. **(b) on the second *tick*.** **(c) on the third *tick*.**

Figure 6: Execution of BTs generated by the EffBT Method

has state transitions, i.e., $ctrl(subBT_j) = \bigcup_{x\in\{1..3\}}\{(jx = j) \wedge (\rho_e \wedge \rho_s) \wedge ctrl(BT_{\rho_x\_j})\}$. Note that conditions in JthCond and CalAct also construct the transition guard. Then, according to the construction algorithm (c.f. Alg. 1 and Alg. 2), we present the *control set* calculation for each $BT_{\rho_x\_j}$ (i.e., $ctrl(BT_{\rho_x\_j}), x \in \{1..3\}$).

- $\mathbf{BT}_{\rho_1\_j}$: As shown in Fig. 4(a), the tree $BT_{\rho_1\_j}$ has the structure of a *Sequence* with *Condition* and *Action* as its children, while this also composes to a *control* with condition and next state shown in line 7 and line 8 (Alg. 1), hence, $ctrl(BT_{(}\rho_1\_j))$ equals $\{(Z \wedge J^s_j) \wedge (Z' \wedge jx' = jx + 1)\}$.
- $\mathbf{BT}_{\rho_2\_j}$: As for the BT_{ρ_2}, its controls are the union of its subtrees ($tree_{\rho_2}$), since we prune the *false* conditions, finally, the control set $ctrl(BT_{\rho_2})$ equals $\bigcup_{r\in\{2..r_j\}}\{mY[j][r] \wedge \neg low \wedge low' \wedge jx' = j | mY[j][r] \wedge \neg low \neq false\}$.
- $\mathbf{BT}_{\rho_3\_j}$: The subtree BT_{ρ_3} has a similar calculate algorithm with BT_{ρ_2}. Hence, its *control set* is represented as $\bigcup_{r\in\{1..r_j\}} \bigcup_{i\in\{1..m\}}\{mX[j][r][i] \wedge \neg low \wedge mX[j][r][i]' \wedge jx' = j | mX[j][r][i] \wedge \neg low \neq false\}$.

Now, the *control set* of BT_{eff} could be calculated inductively by the *ctrl*() function. It's not difficult to find that $\bigcup_{j\in\{1..n\}}\{(jx = j) \wedge (\rho_e \wedge \rho_s) \wedge ctrl(BT_{\rho_x})\}$ subsets to the ρ_x in *trans* after simplification. Furthermore, we could get the conclusion that $C_{seq} \subseteq trans$, i.e., the transitions controlled by our BTs always satisfy the transition in the original GR(1). As the GR(1) method is sound and complete, our method is also inherently sound.

Next, we discuss the complexity of our method. Its procedure consists of GR(1) realizability check and BT construction. The former has a time complexity of $O(mn|\sigma|^2)$, where σ denotes the set of all variables assignments in φ. Then, concerning the latter, it's implemented via a deepest three-level loop (c.f. Alg. 1 and Alg. 2) resulting in complexity of $O(mnk)$. Consequently, the overall computational complexity of our method is the sum of these two components, i.e., $O(mn|\sigma|^2 + mnk)$, or simplified $O(n^4)$.

4 EXPERIMENTAL EVALUATION

We implemented our method and two running examples (NaiveBT, RawBT) on the GR(1) synthesis tool *Spectra*[1]. Supporting materials, including executable JAR package, specification files, and demonstration videos, can be found on our website [1]. To evaluate, we have four research questions: **RQ1** How about the performance of the NaiveBT and the RawBT methods? **RQ2** Does the EffBT method perform better in synthesis? **RQ3** How about the quality of synthesized BTs from EffBT? **RQ4** Do pruning strategies bring improvements to our method?

[1]https://github.com/SpectraSynthesizer

4.1 Scenarios

4.1.1 The FrozenLake+ Scenario. Figure. 7 demonstrates the Frozen Lake game in Gymnasium [21]. Initially, a player is located at the upper left corner (1,1) of a frozen lake grid world with a goal located at the far end of the world (e.g., the lower right corner (8,8) of the map). Several ice holes are distributed in the lake and the player needs to visit the goal while avoiding falling into them. Following that, we expand the initial settings to the FrozenLake+ (also denoted as Frozen$^+$) by adding a beast who chases the player continuously and moves half as fast as the player. In the beginning, the beast is located at a random position next to a random goal (in Fig. 7, the beast is located at position (7,8) next to the only goal). Now, the player needs to visit all goals infinitely often while avoiding being caught and falling. Various map sizes (including 8×8, 16×16, 24×24, and 32×32) of this scenario are generated, in which the occurrence probabilities of ice holes and goals are set to 0.125 and 0.03125 respectively when generating.

Figure 7: The Frozen$^+$ scenario with 8×8 grid map

4.1.2 The Cinderella Scenario. In this scenario, Cinderella battles with her stepmother to maintain the safety of a water bucket system, which has N water buckets arranged in a circle, and each has a maximum capacity of B water units. At the start, all buckets are empty. In each turn, the stepmother pours A units of water into the buckets of her choice, and Cinderella empties C adjacent buckets. The objective of Cinderella is to maintain the emptiness of all buckets. Specifications in Spectra Language are from [18]. For

this scenario, we fix two variables: $A = 5$ and $C = 2$. This means that the stepmother can pour up to five units of water into buckets during her turn and then Cinderella empties two adjacent buckets. Subsequently, we establish four different configurations of this scenario with predefined numbers of bucket and capacity, denoted as set $(N, B) \in \{(6,12), (7,14), (8,16), (9,18)\}$. Besides, robotic simulations of this scenario on Ubuntu 22.04 with the *Mujoco* [20] and *robosuite* [22] platforms can be found on supporting materials [1].

4.1.3 SYNTECH Scenarios. We also examine our methods on various tasks and scenarios from *SYNTECH*[2], including ConvoyCar, RobotArm, Humanoid Robot, Self-Parking Smart Cars, Job Scheduler, and Tower of Hanoi. The number of disks in Hanoi is twelve.

4.2 Experimental Setups and Evaluation Metrics

All experiments are performed on a Windows 10 computer equipped with an AMD Ryzen 9 5950X CPU. The maximum allowed time for synthesis is two hours, and the maximum allowed memory is 2048 MB. Processes will be forcibly terminated if they exceed the allotted time or memory. To minimize the impact of JVM, all methods are executed on a single processor five times under each scenario and each setting, and the final results of evaluated metrics are the averages among them. Subsequently, our experiments consist of two parts along with an ablation study.

In the first part, we intend to assess the synthesis efficiency of our EffBT method compared with the RawBT, NaiveBT, and original GR(1) approaches. Note that we don't make comparisons with prior works about generating BTs in a correct-by-construction way [6][19] since neither of their implementations nor their specifications are publicly accessible. We execute all these methods under the aforementioned scenarios with their respective settings. Throughout the execution, we record three evaluation metrics. First, the metric *#Nodes* denotes the sum of nodes when checking realizability which indicates the scale of problems. Then, the metric $Time_{RC}$ records the time spent on the realizability check. The last metric *Construction Time* records the duration from the time of starting synthesis to its completion, excluding the time taken for realizability checks, i.e., the construction time of GR(1), RawBT, and EffBT methods is the time spent on Step 2, Step 3′, and Step 2′ in Fig. 1 respectively.

Then in the second part, we aim to evaluate the quality of synthesized BTs from aspects of structure and execution. The metric *BT Size* refers to the total number of nodes of generated BTs, inclusive of both ControlFlow and Execution nodes. As an important advantage of BTs, modularity makes sure that BTs are easy to read and modular reuse, hence, we apply an additional metric *Balance* to estimate, which consists of two parts: *Structural Balance* and *Node Balance*. The former is defined as $\frac{N_{min}}{N_{max}}$, in which N_{min} and N_{max} represent the minimum and maximum node counts among subtrees under the root. The latter is defined as $\exp(-|N_{exec}/N_{ctrl} - 2|)$, in which N_{exec} and N_{ctrl} denote the sum of Execution nodes and ControlFlow nodes, respectively. The metric *Node Balance* excepts the proportion of Execution nodes and ControlFlow nodes close to 2 since following the manual design principles in [7], the BT with one control node with two leaves is much easier to read and reuse. Both the node and structural balance denote better if closer to 1 and the metric *Balance* is the sum of them.

[2]https://smlab.cs.tau.ac.il/syntech/spectra/index.html

Table 3: Step-by-Step Evaluation Results of the NaiveBT Method under the Frozen$^+$ 8×8 and 16×16 Scenario

Task	Frozen$^+$ 8 × 8	Frozen$^+$ 16×16
#Nodes (×10k)	0.25	6.37
Step 1: Realizability Check (sec)	0.05	2.10
Step 2: FDS Construction (sec)	0.04	13.38
Step 3: ITE Construction (sec)	0.52	125.26
Step 4: NaiveBT Synthesis (sec)	0.67	144.45
Total Time (sec)	1.28	285.19
BT Size	104392	215346

Apart from these, we also give attention to the execution efficiency of generated BTs, while few prior studies focus on it. Execution of BTs can unfold in dual stages: decision-make and action-take. The *tick* signal propagates through nodes to select the next action in the decision phase. Then the action-taking phase follows, where an action node executes that chosen action, such as moving or grabbing in simulated or physical environments. We execute synthesized results (RawBT and EffBT) five times and calculate their average in simulated scenarios with each setting, during each time we measure the total decision-making time of five thousand steps. Note that we don't include the time for action-taking since it differs and depends on the Action Library and scenarios. For each execution, the environment and system randomly select its next action from a set of available actions. For example, the beast in the Frozen$^+$ scenario chooses to move up or left to chase the player randomly if both of them are available, and the player games too.

Moreover, to answer RQ4, we conduct an ablation study by disabling all pruning strategies in EffBT, which forms the *EffBT-NP* (or *EffBT#*) method. Then, we evaluate it and record metrics following the same procedure in the aforementioned two parts.

4.3 Synthesis Efficiency Results

4.3.1 Results of the NaiveBT Method. Table. 3 demonstrates step-by-step evaluation results of the NaiveBT method under two Frozen$^+$ settings. Based on the results, it is found that the majority of the synthesis time was spent on ITE construction (Step 3) and NaiveBT generation (Step 4), which took much longer time than the Realizability Check (Step 1) and FDS Construction (Step 2). The inefficiency of these two steps lowers the overall method performance. Besides, we didn't present more outcomes of the NaiveBT method since we had already tested it under other scenarios, while it always ran out of time or memory and could only synthesize BTs successfully under the settings in Table. 3. Moreover, both the generated BTs have a size of over 100,000, which is extremely large and results in low execution efficiency. **Answer to RQ1 (NaiveBT):** The NaiveBT method indeed suffers inefficiency problems as it consumes a large amount of time and memory, yet yields huge and low-quality BTs.

4.3.2 Results of Other Methods. Table. 4 summarizes the evaluation results of the synthesis procedure of the original GR(1), RawBT, EffBT#, and our EffBT methods. The time unit in this table is *seconds*, the mark *OOT* represents the process running out of the maximum allowed time, and the bolded is the best result among others. We recognize the superior one of two identical results in this table

Table 4: The Synthesis Results of GR(1), RawBT, EffBT#, and our EffBT Methods

Task		#Nodes (×10k)	Time$_{RC}$ (sec)	Construction Time (sec)			
				GR(1)	RawBT	EffBT	EffBT#
Frozen$^+$	8 × 8	0.25	0.05	0.03	0.01	0.01	**0.01**
	16 × 16	6.37	2.20	13.71	4.21	**2.78**	2.98
	24 × 24	49.22	31.19	280.62	417.26	**37.20**	47.62
	32 × 32	118.12	131.96	OOT	2355.65	**575.90**	1015.64
Cinderella	N = 6, B = 12	0.59	0.35	1.46	**0.01**	0.01	0.01
	N = 7, B = 14	1.10	1.80	3.43	0.02	**0.02**	0.02
	N = 8, B = 16	2.54	19.03	35.84	0.11	**0.08**	0.08
	N = 9, B = 18	9.41	265.46	55.45	0.76	**0.58**	0.59
SYNTECH	ConvoyCar	0.58	0.01	0.02	0.02	0.004	**0.003**
	RobotArm	5.01	1.81	7.92	7.15	**1.82**	1.86
	Humanoid	0.42	0.04	0.03	0.06	**0.02**	0.02
	SelfParkingCar	0.71	0.48	0.71	0.10	0.01	**0.01**
	JobScheduler	1.98	0.69	56.06	888.65	**3.02**	3.41
	Tower of Hanoi	48.14	18.19	48.12	98.68	**37.64**	38.60

according to original data in supporting materials [1]. By the way, as shown in Table. 4, the construction time of RawBT doesn't be restricted by GR(1) since we didn't follow the expected procedure that obtains FDS first (Step 2) and then gets RawBT (Step 3′) when implementing. Specifically, we construct BTs from Step 1 but follow the construction of FDS as mentioned in Step 3′.

The results show that EffBT synthesizes much faster than the RawBT and GR(1) methods under every scenario and setting with average speedups of 53.3% and 75.5%, respectively. For instance, in the larger Frozen$^+$ 32×32 scenario, the GR(1) approach timed out and RawBT took almost one hour, whereas our method achieved results within approximately ten minutes although the problem size escalated. **Answer to RQ2:** Our method performs better in the synthesis procedure, which has significant improvement in time consumption compared to GR(1) and RawBT.

4.4 Behavior Tree Quality Results

Table. 5 presents quality evaluation results of the BTs synthesized by the RawBT, EffBT, and EffBT# methods. We analyze it from the following two aspects including structure and execution.

4.4.1 Structural Results. Considering the size of BTs, Table. 5 shows that EffBT achieves a size reduction of more than half compared to the RawBT method, and the average reduction comes to 56.4%. Then, regarding the BT balance, we find that although RawBT has similar or better node balances with EffBT, it's pretty weak in structural balance, meaning that the sizes of subtrees under those BTs vary significantly. Take the Tower of Hanoi as an example, the structural balance of RawBT is 0.00 (tiny but not equal to zero exactly) since the minimum and maximum sizes among subtrees are 5 and 12298, respectively, leading to its actual balance result being to 0.0004. In comparison, our method fixes this drawback and has a better structural balance benefiting from the design of subtrees, while maintaining the node balance. Therefore, in summary, our method owns better balance and modularity compared to RawBT.

Moreover, as shown in Table. 5, BTs have the same size and tree balance under all settings of the Cinderella scenario. To explain, we draw it out as shown in Fig. 8. Its original root (*Parallel*) has been pruned according to the second pruning strategy since it has only one justice goal. Thus, its root is the *Sequence* node from *subBT*. Then, as all conditions in BT_{ρ_2} and BT_{ρ_3} are *false*, the root of ρ transitions is also pruned making the current root come from BT_{ρ_1}. Finally, the synthesized result has such the same structure. Although they are the same in structure, the inner content of each node differs.

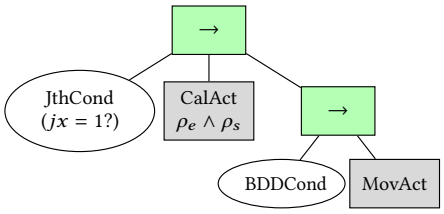


Figure 8: The BT Synthesized in the Cinderella Scenarios

4.4.2 Execution Results. The last part of Table. 5 demonstrates the execution results of RawBT, EffBT, and EffBT# methods. We didn't execute the BTs synthesized by NaiveBT because their sizes are extremely large (over 100,000). In this table, execution time is the average of the sum of five thousand decision-making steps and its unit is *seconds*.

From the results, we notice that RawBT has the worst decision-making procedure among these three methods, and the disparity becomes even more pronounced when the runtime increases. The results confirm our perspective on RawBT, i.e., the structure of RawBT will bring extra useless ticks inherently and waste execution time. Combing the synthesis efficiency results in Sec. 4.3.2 and structural results of RawBT in Sec. 4.4.1, we could **Answer to RQ1 (RawBT):** the RawBT method limited by its structure and suffers problems not only on longer synthesis time consumption but also on lower BT quality and the worst execution performance.

Considering the results of EffBT in Table. 5, it effectively reduces the execution time of decision-making, especially in scenarios like Cinderella in which the execution time is much longer. Moreover, our method has the shortest execution time among other methods in every scenario and has an average performance improvement of 30.8% compared to RawBT. Combining the structural results of

Table 5: Behavior Tree Quality Results of RawBT, EffBT#, and Our EffBT Methods

Task		BT Size			Balance (Structural + Nodes')			Execution (sec)		
		RawBT	EffBT	EffBT#	RawBT	EffBT	EffBT#	RawBT	EffBT	EffBT#
Frozen$^+$	8 × 8	89	**41**	95	0.80 (0.12+0.68)	**1.77** (**1.00**+**0.77**)	1.45 (1.00+0.45)	0.06	**0.05**	0.08
	16 × 16	839	**408**	939	0.93 (0.07+**0.86**)	**1.63** (**0.82**+0.81)	1.29 (0.81+0.48)	0.26	**0.19**	0.30
	24 × 24	3464	**1698**	3924	0.96 (0.05+**0.91**)	**1.62** (**0.76**+0.86)	1.24 (0.75+0.49)	0.67	**0.44**	0.60
	32 × 32	7861	**3871**	8966	0.97 (0.04+**0.93**)	**1.65** (**0.76**+0.89)	1.24 (0.75+0.49)	1.28	**0.78**	0.91
Cinderella	N = 6, B = 12	23	**6**	18	0.84 (0.50+0.34)	2.00 (1.00+1.00)	1.30 (1.00+0.30)	5.62	**3.98**	4.59
	N = 7, B = 14	23	**6**	18	0.84 (0.50+0.34)	2.00 (1.00+1.00)	1.30 (1.00+0.30)	321.09	**195.73**	264.82
	N = 8, B = 16	23	**6**	18	0.84 (0.50+0.34)	2.00 (1.00+1.00)	1.30 (1.00+0.30)	36.22	**23.81**	30.23
	N = 9, B = 18	23	**6**	18	0.84 (0.50+0.34)	2.00 (1.00+1.00)	1.30 (1.00+0.30)	948.33	**570.72**	720.20
SYNTECH	ConvoyCar	47	**20**	46	0.75 (0.23+0.52)	**1.61** (**1.00**+**0.61**)	1.40 (1.00+0.40)	0.27	**0.22**	0.34
	RobotArm	206	**96**	216	0.87 (0.17+**0.70**)	**1.11** (**0.49**+0.62)	0.91 (0.48+0.43)	6.69	**5.20**	6.29
	Humanoid	161	**110**	150	0.63 (0.08+0.55)	1.33 (0.79+0.54)	**1.40** (**0.84**+**0.56**)	0.88	**0.57**	0.72
	SelfParkingCar	352	**201**	333	0.68 (0.06+**0.62**)	1.11 (0.58+0.53)	**1.29** (**0.68**+0.61)	0.89	**0.70**	0.93
	JobScheduler	1064	**516**	1186	0.94 (0.08+**0.86**)	**1.00** (**0.20**+0.80)	0.66 (0.19+0.47)	13.42	**8.96**	10.52
	Tower of Hanoi	24599	**12296**	28690	1.00 (0.00+1.00)	**2.00** (**1.00**+**1.00**)	1.51 (1.00+0.51)	1.50	**0.98**	1.22

EffBT analyzed in Sec. 4.4.1, we could **Answer to RQ3:** Our method owns high-quality BT synthesis ability. The generated BTs have much smaller sizes because of the pruning strategies, accompanied by much better balance and modularity, leading to more efficient execution.

4.5 Ablation Study Results

From the synthesis results of EffBT# in Table. 4, which disables all pruning strategies, we find that the improvement of pruning on construction time becomes more pronounced as the problem increases. In the same scenario with different settings such as Frozen$^+$, pruning brings 6.7% construction time speedup under 16×16, and the percentage grows to 43.3% under 32×32.

For the quality results of EffBT# in Table. 5, pruning brings an average 56.1% BTs' size reduction compared to EffBT#. Besides, the method without pruning decreases the node balance, which means that ControlFlow nodes and Execution nodes differ in those BTs. Then, considering the execution of unpruned BTs, it's a much slower execution since time was wasted on checking meaningless condition nodes and ticking between redundant structures. **Answer to RQ4:** The two pruning strategies are crucial to limiting BTs' size and speed-upping both construction and BT execution periods. If it is removed from our method, the performance will degrade significantly.

5 RELATED WORKS

Over the past few decades, several methodologies, including Reinforcement Learning[8], Evolutionary Algorithms [12][16], Planning and Search [4][11], Large Languages Models[5], and Learning from Demonstrations [9][10][17], have been employed for the synthesis of BTs and have been applied across various domains such as robotic arms, vehicles, and unmanned aerial vehicles. Compared to these learning-based approaches, reactive synthesis exhibits substantial advantages regarding correctness guarantee and task generalization capabilities.

In the context of reactive synthesis methods used in BTs, two prevailing categories exist: model-checking-based and correct-by-construction-based. The model-checking-based approach such as TAMP[14], represents the interaction between robots and their environments as transition systems and then transforms task specifications (described by LTL) into Büchi automata. Subsequently, it searches for a counter-example path within the product of the transition system and the Büchi automaton, and then an equivalent Behavior Tree is synthesized to embody the identified counter-example path. However, since the complexity of transforming LTL to Büchi is EXPTIME, methods based on model-checking also suffer problems including state space explosion and exponential time complexity.

In terms of generating BTs from correct-by-construction methods, such as [6], which divide a subset of LTL into several specifications, and then calculate the value function, the requirement function, and the constraint function for each specification and construct sub-BTs and finally obtain the complete BT by the *Sequence* node. Later in [19], authors unified the three calculation processes into computing the response region and the safe region only. Our EffBT method could also be seen as one of the correct-by-construction methods, however, it differs from prior research. First, we unify the computation process into a single μ-calcus by using GR(1). Second, as we unified the computation process, we could extend BTs with the *Parallel* node, and pay attention to the execution efficiency of synthesized BTs, which the prior works cannot conduct.

6 CONCLUSION

We propose an effective behavior tree synthesis framework - EffBT, which fills the gap that prior works calculate several functions for each formula by unifying them into GR(1) realizability check. Besides, we optimized the execution efficiency of generated BTs by applying the *Parallel* node, while few prior focused on this aspect. The soundness and completeness of our method have been proved. Experimental results in various scenarios and settings have presented that our method performs better in synthesis speed and synthesis quality.

REFERENCES

[1] [n. d.]. *Supporting Materials.* https://effbt.github.io/

[2] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938. https://doi.org/10.1016/j.jcss.2011.08.007 In Commemoration of Amir Pnueli.

[3] Randal E. Bryant. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.* 24, 3 (sep 1992), 293–318. https://doi.org/10.1145/136035.136043

[4] Zhongxuan Cai, Minglong Li, Wanrong Huang, and Wenjing Yang. 2021. 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, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021.* AAAI Press, 6058–6065. https://doi.org/10.1609/AAAI.V35I7.16755

[5] Yue Cao and C. S. George Lee. 2023. Robot Behavior-Tree-Based Task Generation with Large Language Models. In *Proceedings of the AAAI 2023 Spring Symposium on Challenges Requiring the Combination of Machine Learning and Knowledge Engineering (AAAI-MAKE 2023), Hyatt Regency, San Francisco Airport, California, USA, March 27-29, 2023 (CEUR Workshop Proceedings, Vol. 3433)*, Andreas Martin, Hans-Georg Fill, Aurona Gerber, Knut Hinkelmann, Doug Lenat, Reinhard Stolle, and Frank van Harmelen (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-3433/paper4.pdf

[6] Michele Colledanchise, Richard M. Murray, and Petter Ögren. 2017. 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.* IEEE, 6039–6046. https://doi.org/10.1109/IROS.2017.8206502

[7] Michele Colledanchise and Petter Ögren. 2017. Behavior Trees in Robotics and AI: An Introduction. *CoRR* abs/1709.00084 (2017). arXiv:1709.00084 http://arxiv.org/abs/1709.00084

[8] Rahul Dey and Chris Child. 2013. 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.* IEEE, 1–8. https://doi.org/10.1109/CIG.2013.6633623

[9] Kevin French, Shiyu Wu, Tianyang Pan, Zheming Zhou, and Odest Chadwicke Jenkins. 2019. Learning Behavior Trees From Demonstration. In *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019.* IEEE, 7791–7797. https://doi.org/10.1109/ICRA.2019.8794104

[10] Simona Gugliermo, Erik Schaffernicht, Christos Koniaris, and Federico Pecora. 2023. Learning Behavior Trees From Planning Experts Using Decision Tree and Logic Factorization. *IEEE Robotics Autom. Lett.* 8, 6 (2023), 3534–3541. https://doi.org/10.1109/LRA.2023.3268598

[11] Weijiang Hong, Zhenbang Chen, Minglong Li, Yuhan Li, Peishan Huang, and Ji Wang. 2023. Formal Verification Based Synthesis for Behavior Trees. In *Dependable Software Engineering. Theories, Tools, and Applications - 9th International Symposium, SETTA 2023, Nanjing, China, November 27-29, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14464)*, Holger Hermanns, Jun Sun, and Lei Bu (Eds.). Springer, 72–91. https://doi.org/10.1007/978-981-99-8664-4_5

[12] Matteo Iovino, Jonathan Styrud, Pietro Falco, and Christian Smith. 2021. Learning behavior trees with genetic programming in unpredictable environments. In *2021 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, 4591–4597.

[13] Fu Li, Xueying Wang, Bin Li, Yunlong Wu, Yanzhen Wang, and Xiaodong Yi. 2024. A Study on Training and Developing Large Language Models for Behavior Tree Generation. *CoRR* abs/2401.08089 (2024). https://doi.org/10.48550/ARXIV.2401.08089 arXiv:2401.08089

[14] Shen Li, Daehyung Park, Yoonchang Sung, Julie A. Shah, and Nicholas Roy. 2021. Reactive Task and Motion Planning under Temporal Logic Specifications. In *2021 IEEE International Conference on Robotics and Automation (ICRA)* (Xi'an, China). IEEE Press, 12618–12624. https://doi.org/10.1109/ICRA48506.2021.9561807

[15] Shahar Maoz and Ilia Shevrin. 2020. Just-In-Time Reactive Synthesis. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 635–646. https://doi.org/10.1145/3324884.3416557

[16] Aadesh Neupane and Michael A. Goodrich. 2019. 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*, Sarit Kraus (Ed.). ijcai.org, 513–520. https://doi.org/10.24963/IJCAI.2019/73

[17] Lisa Scherf, Kevin Fröhlich, and Dorothea Koert. 2024. Learning Action Conditions for Automatic Behavior Tree Generation from Human Demonstrations. In *Companion of the 2024 ACM/IEEE International Conference on Human-Robot Interaction* (, Boulder, CO, USA,) *(HRI '24).* Association for Computing Machinery, New York, NY, USA, 950–954. https://doi.org/10.1145/3610978.3640673

[18] Ilia Shevrin and Matan Yossef. [n. d.]. Spectra Example: Cinderella-Stepmother Problem. https://smlab.cs.tau.ac.il/syntech/examples/cinderella/CinderellaStepmother.pdf.

[19] Tadewos G. Tadewos, Abdullah Al Redwan Newaz, and Ali Karimoddini. 2022. Specification-guided behavior tree synthesis and execution for coordination of autonomous systems. *Expert Systems with Applications* 201 (2022), 117022. https://doi.org/10.1016/j.eswa.2022.117022

[20] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

[21] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. 2023. Gymnasium. https://doi.org/10.5281/zenodo.8127026

[22] Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Soroush Nasiriany, and Yifeng Zhu. 2022. robosuite: A Modular Simulation Framework and Benchmark for Robot Learning. arXiv:2009.12293 [cs.RO]