EffBT: An Efficient Behavior Tree Reactive Synthesis and Execution Framework

Ziji Wu[1,2], Yu Huang[1,2], Peishan Huang[1,2], Shanghua Wen[2], Minglong Li[2,✉], Ji Wang[1,2,✉]
[1]State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China
[2]College of Computer Science and Technology, National University of Defense Technology, Changsha, China
{wuziji, huangyu, huang_ps, liminglong10, wj}@nudt.edu.cn, shanghua_w@126.com

***Abstract*—Behavior Trees (BTs), originated from the control of Non-Player-Characters (NPCs), have been widely embraced in robotics and software engineering communities due to their modularity, reactivity, and other beneficial characteristics. It is highly desirable to synthesize BTs automatically. The consequent challenges are to ensure the generated BTs semantically correct, well-structured, and efficiently executable. To address these challenges, in this paper, we present a novel reactive synthesis method for BTs, namely EffBT, to generate correct and efficient controllers from formal specifications in GR(1) automatically. The idea is to construct BTs soundly from the intermediate strategies derived during the algorithm of GR(1) realizability check. Additionally, we introduce pruning strategies and use of *Parallel* nodes to improve BT execution, while none of the priors explored before. We prove the soundness of the EffBT method, and the experimental results demonstrate its effectiveness in various scenarios and datasets.**

***Index Terms*—Behavior Trees, Reactive Synthesis, GR(1), Efficient Execution**

I. Introduction

Behavior Trees (BTs), initially developed for controlling Non-Player Characters (NPCs), have gained significant popularity in robotics due to their advantages in modularity, reactivity, and other features. Moreover, they are also a powerful tool in software engineering, particularly in domains where complex decision-making and control logic are required. They play a key role in organizing, controlling, and executing complex behaviors in a modular and maintainable way, and provide the programming framework for component-based system and software design. BTs excel in environments requiring reactive systems, task prioritization, and failure management. As robotic and software systems become more and more complex, the most desirable goal for developers is to automatically design BTs. In the past decades, researchers have worked a lot to search for solutions. The existing methods, including evolution algorithms [1][2], reinforcement learning [3], task and motion planning [4], Large Language Models [5][6], and learning from demonstrations [7][8], have been used to generate BTs for solving a variety of tasks. Compared with these informal methods mentioned above, reactive synthesis, which automatically obtains correct-by-construction representations (e.g., automaton) from formal specifications (e.g., Linear Temporal Logic (LTL)), has also been used in BTs' synthesis. It has a significant advantage in that its outcomes are rigorously aligned with the provided formal specifications.

There are two main categories in terms of BTs' reactive synthesis. The first, such as TAMP [4], utilizes a model-checking-based strategy on general LTL and suggests synthesizing BTs from an emptiness-checking procedure over product automata, which is the production of the Büchi automata (converted from the negation of LTL specifications) and a transition system (describes the robot and environment). If the specification is satisfiable, the procedure identifies a counter-example path (i.e., a state transition path), and then constructs BTs from it. Nevertheless, the conversion from LTL to Büchi automata exhibits exponential time complexity, implying that the conversion time grows exponentially with the size of the formula. Consequently, the approaches relying on this strategy inevitably face the challenge of high computational complexity (at least EXPTIME). The second category, based on the Fragment of LTL (F-LTL), obtains BTs from F-LTL specifications rather than general LTL to reduce the complexity. In previous studies [9][10], authors restrict formulas to specific LTL patterns and then derive transition strategies by calculating the value functions, requirement functions, and constraint functions for each formula. Following that, the final BTs can be constructed by encoding and assembling the results of these functions.

In addition, two indirect methods are identified for constructing BTs from GR(1) in our paper, termed NaiveBT and RawBT, respectively. Both of them construct BTs from the generated Fair Discrete Systems after GR(1) realizability checking. We utilize them for comparison and point out their drawbacks in the structure and execution of BTs through experimental and theoretical analysis. Aside from this, an extra alternative method involves transforming the specification into an automaton and then obtaining BTs from it. However, BTs are action-based structures 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 state transitions in automata.

In this work, we propose a novel GR(1)-based behavior tree synthesis framework, termed EffBT. Given a GR(1) specification, our method regards it as a game structure that encapsulates the system and environment dynamics along with a GR(1) formula that describes the objectives. Then, EffBT checks its realizability and stores transition strategies if

✉ Corresponding author

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00225

realizable [11]. We intend to directly extract BTs from these intermediates. However, it is not straightforward due to the following two challenges: constructing semantically correct and well-structured BTs, and improving their execution efficiency. For the first challenge, we decompose and restructure the intermediate strategies, then modularly construct corresponding subtrees and combine them. For the second challenge, we propose pruning strategies and the use of *Parallel* nodes to improve BT execution, an approach not previously explored. Subsequently, we prove the soundness of the EffBT method.

The EffBT also falls into the second category of synthesizing from LTL fragments like the priors [9][10]. However, our method differs significantly from those in not only the specification language (aka GR(1) vs specific formula patterns) but also the approach to construct BTs and their optimization. First, our method adopts GR(1) realizability check as the unified procedure of transition calculating rather than performing individual function calculations for each formula pattern, which makes intermediate results more regular. Then, benefiting from this, we introduce a more structured BT construction approach compared to priors. In our approach, each subtree features a similar sub-structure but is responsible for addressing distinct subgoals. This design not only aligns with the expertise gained from manually designing behavior trees in large quantities, but also ensures that the BTs exhibit superior modularity, better readability, and are much more convenient for debugging. Furthermore, to enhance the execution performance of generated BTs, we investigate and propose the design principle of using *Parallel* nodes.

To our knowledge, few priors focused on structural optimization for improving the execution efficiency of BTs, and none of them utilizes *Parallel* nodes. In the design of BTs, pursuing a concise and modular tree structure is a goal worth striving for: Firstly, a concise structure with fewer nodes and shorter paths helps the efficiency of the decision-making process, which is crucial for hard real-time systems whose overheads should be reduced as required. Secondly, modular BTs are generally easier to understand and reuse, which is essential for team collaboration and knowledge transfer. It is deserved to construct BTs correctly and keep the results concise and modular simultaneously.

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

- We propose an algorithm for the direct construction of well-structured BTs from the intermediates of the GR(1), and we further prove the soundness of our method.
- We optimize the execution efficiency of generated BTs by employing pruning strategies and incorporating *Parallel* nodes when construction.
- We substantiate the effectiveness of our method through experimental evaluations compared to NaiveBT, RawBT, as well as two representative GR(1) synthesis methods Spectra [12] and JITS [13], conducted across various tasks and datasets.

The remainder of this paper is organized as follows. Sec. II provides necessary backgrounds on BTs and GR(1) synthesis.

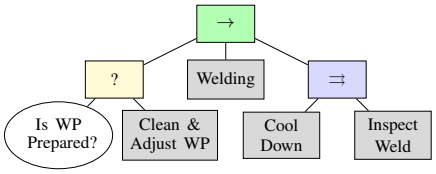


Fig. 1: A simplified Work Pieces (WP) Welding BT.

Sec. III presents the NaiveBT and the RawBT methods. Sec. IV gives the problem formulation and introduces our method in detail, along with soundness proof and complexity analysis. Sec. V contains the experimental research questions, configurations, results, and analysis. Finally, Sec. VI recalls prior studies, and Sec. VII concludes this paper.

II. Preliminaries

A. Behavior Trees

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

The leaf nodes (also known as *Execution* nodes) include *Condition* (as the ellipse nodes in Fig. 1) and *Action* nodes (as the gray box node in Fig. 1). *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, *Failure* when actions failed, else return *Running*.

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. 1, resp.), whose statuses depend on their type and statuses of their children. When *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 *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 succeed, *Failure* if $n-m+1$ children fail, and *Running* in other cases.

Usually, during the execution of BTs, a *blackboard* is built for variable sharing (e.g., sensor information) among nodes, which operates a dictionary with keys and values, and allows all the nodes in BTs to read from and write to it.

B. The GR(1) Synthesis

GR(1) synthesis is originally devised to solve the problem of automatically generating digital designs from linear-time specifications [11]. Specifically, it takes GR(1) specifications as

input and yields correct-by-construction controllers (e.g., Fair Discrete System (FDS)). The GR(1) specification ψ defined on atomic propositions of environment and system variables ($\mathcal{X}$ and $\mathcal{Y}$, resp.) contains initial assumptions and guarantees which restrict initial states (θ_e and θ_s, resp.), safety assumptions and guarantees that describe the transition from current to next state (ρ_e and ρ_s, resp.), and justice assumptions and guarantees that require an assertion holds infinitely many times during a computation (J^e and J^s, resp.), in which a *state* is an assignment to $\mathcal{X} \cup \mathcal{Y}$, assumptions denote possible states of the environment, and guarantees describe the system's reactions to those states. $\varphi = \bigwedge_{i\in\{1,\cdots,m\}} \mathcal{GF} J_i^e \rightarrow \bigwedge_{j\in\{1,\cdots,n\}} \mathcal{GF} J_j^s$ is called GR(1) formula which is a fragment of LTL that suites for describing most reactive systems. Prior works [11][13] are recommended for further reading about the GR(1) specification and synthesis.

1) Realizability Check of GR(1): This procedure checks whether the given GR(1) specification ψ is realizable, and this problem is formulated as an equivalent two-player game between system and environment [11]. The game structure GS is a tuple $\langle \mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s \rangle$ constructed from the GR(1) specification, where $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$, θ_e, θ_s are initial assumptions and guarantees, ρ_e, ρ_s are transition assumptions and guarantees. Then, the given GR(1) specification is realizable if and only if the system wins the game, and the system winning condition of this game is GR(1) formula φ, i.e., regardless of how the environment changes, the system can find a way to continue the game, and the path of the game conforms to φ. System-winning set W_s (or z) is a set of states that, starting from these states, system-winning strategies always exist. Hence, the specification is realizable iff W_s is not an empty set. Then, to determine the system-winning set W_s, we should solve its equivalent problem, namely calculating the semantics of the following μ-calculus formula over game structure GS achieved through a three-level fixed point calculation:

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

where $\langle\!\rangle\varphi$ denotes the controlled predecessor, from the states in which the system can force the environment to visit a state in φ. Symbols μ and ν denote the minimum and the maximum fix-point operator, respectively. Symbols X, Y, and Z are relation variables that are assigned as *true*, *false* and *true* initially. During the calculation, two crucial intermediate arrays that entail system transition strategies, mX$[j][r][i]$ and mY$[j][r]$, are stored. Here, $j \in \{1,\cdots,n\}$, $r \in \{1,\cdots,k\}$, and $i \in \{1,\cdots,m\}$, where the variables n, k, and m denote the counts of system guarantees, the iteration times for calculating μY, and the counts of environment assumptions, respectively.

mX[][][] is the fix-point calculation results of the innermost safety game, i.e., the maximum fix-point νX. Intuitively, from the states in mX$[j][r][i]$, the system could either move one step closer to reaching $J_j^s \wedge \langle\!\rangle Z$ states within r transitions to satisfy the jth system guarantee, or keep forcing the environment to violate the ith environmental assumption J_i^e which violates the left part of the GR(1) formula.

mY[][] stores the fix-point calculation results of the inner least reachability game, i.e., the minimum fix-point μY, which assures that the game can reach $J_j^s \wedge \langle\!\rangle Z_{j\oplus 1}$ states finally. From the states in mY$[j][r]$, the system could either move to $J_j^s \wedge \langle\!\rangle Z$ states within r steps that satisfy the jth system guarantee or violate at least one environment assumption J_i^e for some i.

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

2) FDS Construction in GR(1): Following the original process of GR(1) synthesis, static Fair Discrete System controllers defined as $\langle \mathcal{V}_\mathcal{D}, \theta_D, \rho \rangle$ will be constructed from mX[][][] and mY[][] immediately after the realizability check, in which $\mathcal{V}_\mathcal{D} = \mathcal{V} \cup jx$, $\theta_D = \theta_e \rightarrow (\theta_s \wedge (jx = 1) \wedge z)$ that restricts initial states. The extra variable set jx represents the index of some justice goal the system is aiming to accomplish.

Intuitively, the transition ρ in FDS has three parts ρ_1, ρ_2, and ρ_3, each tailored to handling a separate case. When the current is a z state, the first part ρ_1 deals and marks jx 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 but still stays in mX$[j][r][i]$ states. Their formal representations are as follows, where the transition ρ in FDS equals $\rho_1 \vee \rho_2 \vee \rho_3$.

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

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

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

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

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

III. The NaiveBT and the RawBT Methods

In this section, we briefly introduce the NaiveBT and the RawBT methods. We also explain their inherent limitations through theoretical analysis and later experiments.

A. The NaiveBT Method

Tracing the **Brown Arrow** in Fig. 2, the following steps outline the procedure of the NaiveBT method for synthesizing BTs from GR(1): Step 1 and Step 2 conduct realizability check over (GS, φ) and construct FDS, similar to the original GR(1) synthesis; Then, Step 3 transforms the FDS into an If-Then-Else (ITE) structure [14] as it is stored as a BDD; Finally, Step 4 constructs a BT which is straightforward due to the sufficient capacity of BTs to cover the ITE logic. The first three steps have already been proposed and implemented in [11]

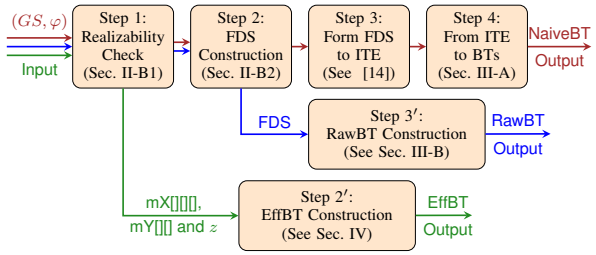


Fig. 2: Procedure Overview of the NaiveBT (**Brown Arrow**), RawBT (**Blue Arrow**), and EffBT (**Green Arrow**) Methods

and [14]. Limited to the pages, the detailed introduction of Step 4 about how to construct BTs from ITE by matching regular patterns and applying transition rules is in the supplementary materials [15].

B. The RawBT Method

Following the **Blue Arrow** in Fig. 2, the RawBT is also a FDS-based BTs' synthesis method but without transforming to ITEs. In detail, it obtains BTs from the transition strategy ρ in FDS in a top-down manner (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 *Selector* as root. Regarding the innermost conjunction formulas, we identify and separate the action and condition according to whether the predication describes the current or the next 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 node *Condition*$((jx=j)\wedge z\wedge J_j^2\wedge\rho)$ and the node *Action*$(z'_{|\mathcal{X}} \wedge (jx' = j + 1))$, where $z'_{|\mathcal{X}}$ is a systematic z' state that only preserves those predicates of system variables. We then combine them with *Sequence* and mount the subtree under the corresponding *Selector* node in ρ-subtrees. However, after observing the execution of such BTs, we introduce the following finding. The detailed proof of this property is available in [15].

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

Each BT produced by RawBT has three subtrees that correspond to ρ_1, ρ_2, and ρ_3 transitions in FDS. As per Prop. 1, state transitions predominantly happen in reaching justice goals, but the parts of reaching a single goal are distributed among those subtrees. Consequently, during execution, BTs must frequently tick and switch to find the proper subtree in different subtrees, which is a process that wastes lots of time in decision-making. In our approach, we addressed this limitation by introducing a novel structural design and utilizing *Parallel* nodes.

IV. Methodology

We propose EffBT, a method for constructing BTs from GR(1) that solves the aforementioned challenges and overcomes the limitations inherent in NaiveBT and RawBT.

A. Problem Formulation

Recall that $\mathcal{X}$ and $\mathcal{Y}$ denote the environment and system variables, respectively. The set of all variables $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$. The set of all states is denoted by S, where each state $s \in S$ is an assignment to $\mathcal{V}$. The GR(1) specification ψ can be decomposed into a game structure GS and a GR(1) formula φ. Subsequently, we present the discrete definition of BTs.

Definition (Behavior Tree)**.** *A BT is a tuple $\mathcal{T}=\langle f, r\rangle$, where f : gud $\rightarrow$ dst describes its effects in system states. Here, gud : $2^{\mathcal{V}\cup\mathcal{X}'}$ and dst : $2^{\mathcal{Y}'}$ denotes the transition guards and the destination system states, resp. The mapping $r : 2^{\mathcal{V}\cup\mathcal{V}'} \rightarrow \{\mathcal{S}, \mathcal{R}, \mathcal{F}\}$ represents the return status.*

An execution of a BT over GS is an infinite state sequence $\sigma = s_0, s_1, \ldots$, where s_0 is the initial state. A transition from s_i to s_{i+1} is legal iff $s_i \cup s_{i+1|\mathcal{X}} \vDash$ *gud* and $s_{i+1|\mathcal{Y}} \vDash$ *dst*, where $s_{i+1|\mathcal{X}}$ denotes the environment state only contains $\mathcal{X}$ and excluding system variables $\mathcal{Y}$. A BT satisfies a GR(1) specification iff all of its executions satisfy ψ, denoted as $BT \vDash \psi$. Given these preliminaries, the BT synthesis problem is formally defined as follows:

Problem. *Given a GR(1) specification ψ, the problem is to automatically construct a BT $\mathcal{T}$ that satisfies ψ, i.e., $\mathcal{T} \vDash \psi$.*

B. The BT Construction Algorithm in EffBT

1) Overview: As depicted in Fig. 2 (following the **Green Arrow**), our framework consists of two main procedures. The first, as detailed in Sec.II-B, checks the realizability of GR(1) specifications. In this section, we focus on presenting our algorithm (Step 2′), which constructs correct BTs from the results of Step 1. Moreover, it refines the structure of subtrees to mitigate the shortcomings in RawBT and overcome the first challenge.

In addition to basic *ControlFlow* nodes, as summarized in Table. I, leaf nodes in the result BTs contain three types, including *Condition* node Cond, and two *Action* nodes CalAct and MovAct. Note that an extra Action Library is required in MovAct, as well as for the $A^{P'}$ in NaiveBT, and the *Action*() in RawBT, when the goal is to control actual movements, e.g., a robot arm in reality. This is because our method is incapable of synthesizing primitive controllers (e.g., PID controllers), which is a common abstraction in reactive synthesis. Intuitively, the Action Library stores the action name and variables as key and its actual primitive action controller as value, which will be dynamically matched and loaded when execution.

TABLE I: Predefined Condition and Action Types

Name (Abbr.)	Description
Cond	It assesses whether its inner condition are satisfied, returns *Success* if true, or *Failure* otherwise.
CalAct	This action conducts pure computation 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.

The pseudo-code of BTs' construction algorithm in EffBT is shown in Alg. 1 and Alg. 2. This procedure takes as input

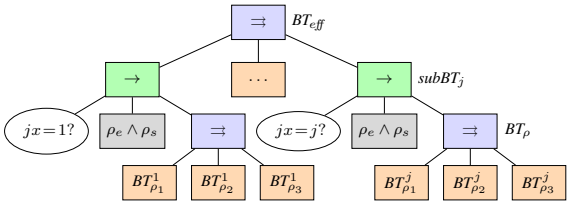
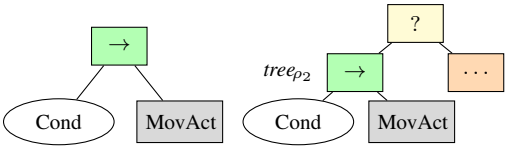


Fig. 3: Overview of the Structure of the BT from EffBT

the intermediates and results (mX[][][], mY[][], and z) from GR(1) realizability check, along with an auxiliary variable set jx which identifies sub-strategies. Then it generates the result behavior tree BT_{eff} that satisfies φ whose root applies the *Parallel* (c.f., line 1 in Alg. 1). Figure. 3 demonstrates an overview of the structure of BT_{eff}, it has n structurally similar *Sequence* subtrees and the jth one referred 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. A single subtree $subBT_j$ is composed of three parts: a Cond node to judge whether the current goal is the jth justice guarantee; a CalAct node to ensure both systems and environments satisfy the transition rules $\rho_e \wedge \rho_s$ (c.f., line 4 in Alg. 1); and a *Parallel* subtree BT_ρ with three children that represent the ρ_1, ρ_2, and ρ_3 transitions. Note that the ρ_x transitions discussed here only correspond to a single justice guarantee that differs from those in Sec. II-B2. To distinguish, we denote them as BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3}, each of them differs in structures and meanings. By decomposing the original transitions, we can categorize ρ_x transitions associated with the jth justice goal into a single subtree and avoid the decision-making problem in the RawBT method.

Besides, an internal blackboard is designed to store running statuses (e.g., the jth subtree currently active, system and environment states) and intermediate calculation results from CalAct, which is transparent to users. During the execution, the environment and system states are continuously perceived and logged onto this blackboard. Based on this information, BTs determine the subsequent actions to be taken.

2) Construction of BT_{ρ_1} and BT_{ρ_2}: As shown in Fig. 4(a), the jth ρ_1 subtree BT_{ρ_1} is constructed as *Sequence* with Cond and MovAct as its children (c.f., lines 19 to 20 in Alg. 1). The instanced Cond checks whether the current state satisfies the jth justice goal, if true, then the instanced MovAct node takes action to the $j \oplus 1$ goal.



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

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

As presented in Fig. 4(b), the BT_{ρ_2} is constructed as a *Selector* with many *Sequence* subtrees (referred as $tree_{\rho_2}$), each subtree has the same structure with BT_{ρ_1} but differs in contents (c.f., lines 8 to 18 in Alg. 1). Intuitively, the Cond under the rth $tree_{\rho_2}$ checks whether current state satisfies mY[j][r], if

Algorithm 1 BT Construction of the EffBT Method

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

1: $BT_{eff} \leftarrow$ Parallel Node
2: **for all** $j \in \{1, \cdots, n\}$ **do**
3: $subBT_j \leftarrow$ Sequence Node;
4: $subBT_j$.addChildren({Cond($jx = j?$), CalAct($\rho_e \wedge \rho_s$)})
5: $BT_\rho \leftarrow$ Parallel Node
/*Construction of the jth sub-tree $BT^j_{\rho_3}$ */
6: $BT^j_{\rho_3} \leftarrow$ ConstructTree$_3$(mX[][][], z, j)
7: BT_ρ.addChildIfNotEmpty($BT^j_{\rho_3}$) ▷ Prune (Case 2).
/*Construction of the jth sub-tree $BT^j_{\rho_2}$ */
8: $BT^j_{\rho_2} \leftarrow$ Parallel Node; $low \leftarrow$ mY[j][0]
9: **for all** $r \in \{2, \cdots, r_j\}$ **do**
10: $cond \leftarrow$ mY[j][r] $\wedge \neg low \wedge low'_{|\mathcal{X}}$
11: **if** $cond$ **then** ▷ Prune (Case 1).
12: $tree_{\rho_2} \leftarrow$ Sequence.addChild(Cond($cond$))
13: $tree_{\rho_2}$.addChild(MovAct($low'_{|\mathcal{Y}} \wedge jx' = j$))
14: BT_{ρ_2}.addChild($tree_{\rho_2}$)
15: **end if**
16: $low \leftarrow low \vee$ mY[j][r]
17: **end for**
18: BT_ρ.addChildIfNotEmpty($BT^j_{\rho_2}$) ▷ Prune (Case 2).
/*Construction of the jth sub-tree $BT^j_{\rho_1}$ */
19: $BT^j_{\rho_1} \leftarrow$Sequence.addChild(Cond($jx = j \wedge z \wedge J^s_j \wedge z'_{|\mathcal{X}}$))
20: $BT^j_{\rho_1}$.addChild(MovAct($z'_{|\mathcal{Y}} \wedge jx' = j + 1$))
21: **if** $subBT_j$.hasNoChild() **then** ▷ Prune (Case 2).
22: $subBT_j \leftarrow BT_{\rho_1}$
23: **else if** $n = 1$ **then** ▷ Prune (Case 2).
24: $BT_{eff} \leftarrow subBT_1$
25: **else**
26: $subBT_j$.addChild(BT_ρ)
27: BT_{eff}.addChild($subBT_j$)
28: **end if**
29: **end for**

true, then the MovAct forces the system to mY[j][$r-1$] states that one step closer to the jth goal and finally reach the goal.

3) 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 is constructed as a *Selector* with two-level nested *Sequence* subtrees for each r and i (c.f., lines 3 to 21 in Alg. 2). Moreover, for the lowermost subtree, which is also structurally similar to BT_{ρ_1}, its instanced Cond node checks whether the current state satisfies mX[j][r][i], if true, the MovAct force the system stays to keep violating the ith environment assumption.

C. The Optimizations to EffBT

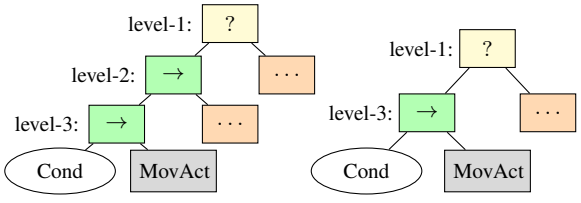
The execution of BTs can unfold in two stages: decision-make and action-take. The *tick* signal propagates through nodes to select the next action during the decision-make phase. Then a leaf node executes the chosen action during the action-take phase, such as moving or grabbing in simulated or physical environments. In this part, we propose pruning strategies and incorporate *Parallel* nodes to improve the execution speed of BTs, specifically, by decreasing decision-making time.

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

Input: mX[][][], z, j
Output: $BT^j_{\rho_3}$ — the root node of this tree

1: $BT^j_{\rho_3} \leftarrow$ Selector Node
2: $low \leftarrow$ false
3: **for all** $r \in \{1, \cdots, r_j\}$ **do**
4: $tree_{\rho_3} \leftarrow$ Sequence Node
5: **for all** $i \in \{1, \cdots, m\}$ **do**
6: $cond \leftarrow$ mX[j][r][i] $\wedge \neg low \wedge$ mX$'_{|\mathcal{Y}}$[j][r][i]
7: $act \leftarrow$ mX$'_{|\mathcal{X}}$[j][r][i] $\wedge jx' = j$
8: **if** $cond$ **then** ▷ Prune (Case 1).
9: **if** $m = 1$ **then** ▷ Prune (Case 2).
10: $tree_{\rho_3}$.addChild(Cond($cond$))
11: $tree_{\rho_3}$.addChild(MovAct(act))
12: **else**
13: $sub_{\rho_3} \leftarrow$ Sequence.addChild(Cond($cond$))
14: sub_{ρ_3}.addChild(MovAct(act))
15: $tree_{\rho_3}$.addChild(sub_{ρ_3})
16: **end if**
17: **end if**
18: $low \leftarrow low \vee$ mX[j][r][i]
19: **end for**
20: BT_{ρ_3}.addChild($tree_{\rho_3}$)
21: **end for**
22: **return** BT_{ρ_3}

1) Pruning: 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, line 8 in Alg. 2), the subtree will be pruned as it will never be executed. The second case occurs if a *ControlFlow* node has only one child, we remove it and reload its child to its parent directly. As shown in Fig. 5(a), if the *Sequence* in level-2 has just one child, it will be pruned (c.f., lines 8 to 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 and lines 21 to 28 in Alg. 1).



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

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

2) Adoption of Parallel Nodes: Parallelism is always being pursued in computing even if it can be simulated by interleaving. Because it would be a better choice to keep the parallel nature in both the problem and solution. Besides, with the increasing application of multi-core processors, the parallel is much desired and feasible in many domains such as robotics.

We adopt the *Parallel* (m=1) as an alternative to *Selector*. They share similar semantics, with success being determined by the success of any child node. However, *Parallel* executes its children in parallel, which would be faster than *Selector* whose children execute one by one, and later experiments have verified the effectiveness. To our knowledge, none of the priors focused on the efficiency of BTs' execution and none of them utilizes *Parallel* nodes. It is nontrivial since potential conflicts need to be solved. Thus, we identify possible conflict situations during BTs' construction and summarize them as principles in Prop. 2. To use *Parallel* unambiguously, any BT violating the rules should not use *Parallel* as root, more details are presented in the additional materials [15].

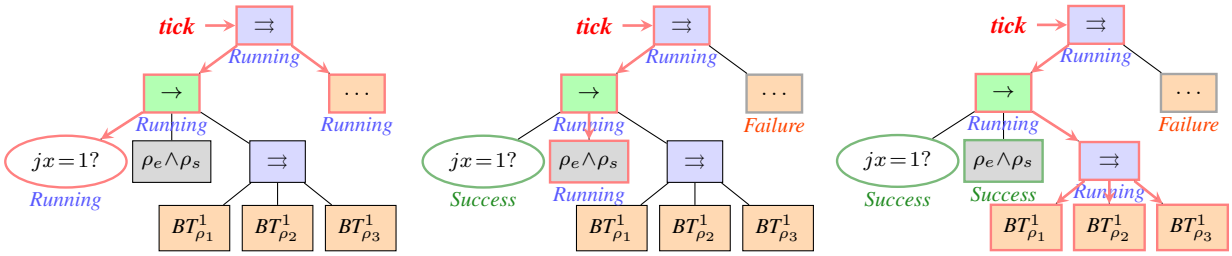
Proposition 2. *A BT could adopt Parallel (m=1) as a new root if satisfies the following conditions: 1. its root is an Selector; 2. no blackboard written conflict or simultaneous condition and action among its children.*

Owing to the regularity of GR(1) formulas, our algorithm, particularly the modularized and unified subtree construction and the usage of the auxiliary variable jx, could synthesize BTs with superior structure and is capable of employing *Parallel* nodes, while priors have difficulties blamed on their structure design. For example, the BT in [9] is a *Sequence* as root with several *Sequence* subtrees (BT_n, $n \in \{2, \cdots, 6\}$), and the deepest subtrees (e.g., BT_{2i}) uses *Selector* as root with both *Condition* and *Action* nodes as children. However, replacing the *Selector* with a *Parallel* would lead to simultaneous condition check and action conduct when executing BT, which is not reasonable obviously.

By the way, it is also feasible to use *Parallel* in the RawBT method (termed RawBT-P) as a substitute for the *Sequence* between the three ρ-transition subtrees, which has a limited effect on the structure of BTs, except for the nodes' replacement. As for the execution performance of BTs, we argue that the adoption of *Parallel* in RawBT could bring an improvement since they tick their children simultaneously, which effectively shortens the decision-making time. However, it is still unable to match the performance of EffBT. As per Prop. 1, state transitions in a single sub-strategy (i.e., to reach a particular justice goal) significantly exceed those between different sub-strategies. Hence, RawBT-P encounters identical challenges as RawBT, i.e., it also requires frequent ticking and switching between the three ρ-transitions to determine the appropriate one among different subtrees. Overall, the final efficiency result would be: EffBT $>$ RawBT-P $>$ RawBT.

D. An Example of BT Construction and Execution

A simple example of constructing BTs using EffBT is provided in the supplementary materials [15]. In this part, we demonstrate the execution of synthesized BTs. 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 each subtree and then the leftmost children Cond($jx = j$), $j \in \{1, \cdots, n\}$. When ticked, these nodes query the inner blackboard for the value of jx, check whether is currently running at the jth subtree and only one subtree would return *Running*. Then on the second tick (see Fig. 6(b)), the running subtree ticks its next child CalAct($\rho_e \wedge \rho_s$) that



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

Fig. 6: Execution of BTs generated by the EffBT Method

calculates the conjunction of current state, movements of environment, and $\rho_e \wedge \rho_s$ verify the legality of the transitions and writes the results to blackboard with key *mediate*.

After this, on the third tick (see Fig. 6(c)), the last part of the subtree is ticked, which then ticks all its children (BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3}) simultaneously. Each BT_{ρ_x} handles different cases as we have introduced before and only one MovAct node will be finally executed. Take BT_{ρ_1} as an instance, it ticks its child Cond to judge whether the current is a $z \wedge J_j^s$ state at first, which calculates the conjunction of the *mediate* in inner blackboard and the BDD in itself, returns *Failure* if the result is *false*, or otherwise returns *Success* and updates the value of *mediate* with the result. If succeeded, the MovAct will be executed which matches and takes action (in Action Library) to make the system state to $z' \wedge jx' = j \oplus 1$. Behaviors of the other two subtrees (BT_{ρ_2} and BT_{ρ_3}) are similar. 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 cyclically.

E. Correctness and Complexity

Theorem. *EffBT is sound, whenever a given GR(1) specification ψ is realizable, the final BT_{eff} can be correctly constructed, i.e., $BT_{\mathit{eff}} \vDash \psi$.*

Intuitively, to prove the soundness, we should prove that the dynamics of BT_{eff} is a subset of FDS behaviors generated from the original GR(1), since that the GR(1) synthesis is both sound and complete [11]. Then, we formally present the proofs as outlined below.

Proof. We explain the dynamics of BT_{eff} by extracting its *effect* $f = \mathscr{F}[\![BT_{\mathit{eff}}]\!]$, which equals $\bigvee_{j\in\{1,\cdots,n\}} \mathscr{F}[\![subBT_j]\!]$ because BT_{eff} is composed of n separate subtrees, and $\mathscr{F}[\![.]\!]$ is the *effect* calculate function. The disjunction of two *effects* (e.g., f_i and f_j) yields an new *effect* that defined as $\{gud_i \vee gud_j, dst_i \vee dst_j\}$. Then, considering a single subtree $subBT_j$, its *effect* involves three parts from BT_{ρ_1}, BT_{ρ_2}, and BT_{ρ_3} as only the BT_{ρ_x} subtrees conduct actions and have state transitions, i.e., $\mathscr{F}[\![subBT_j]\!] = \bigvee_{x\in\{1,\cdots,3\}}\{(jx=j) \wedge \rho_e \wedge \rho_s \wedge \mathscr{F}[\![BT^j_{\rho_x}]\!].gud, \mathscr{F}[\![BT^j_{\rho_x}]\!].dst\}$. Note that the conditions in Cond and CalAct are also constructed into the transition guard. Then, following the construction algorithm (c.f., Alg. 1 and Alg. 2), we present the *effect* calculations for each $BT^j_{\rho_x}$ (i.e., $\mathscr{F}[\![BT^j_{\rho_x}]\!]$).

As shown in Fig. 4(a) and line 7 to line 8 in Alg. 1, $BT^j_{\rho_1}$ has the structure of a *Sequence* with Cond and MovAct as its children. This also composes to an *effect* with the condition as *gud* and next state as *dst*, thus, $\mathscr{F}[\![BT^j_{\rho_1}]\!]$ equals $\{z \wedge J_j^s, z' \wedge jx' = jx + 1\}$. As for BT_{ρ_2}, its *effect*s are the disjunction of all its subtrees (i.e., $tree_{\rho_2}$). Besides, due to the pruning of *false* conditions, the final *effect* $\mathscr{F}[\![BT^j_{\rho_2}]\!]$ equals $\bigvee_{r\in\{2,\cdots,r_j\}}\{\mathrm{mY}[j][r] \wedge \neg low, low' \wedge jx' = j\}$, where $\mathrm{mY}[j][r] \wedge \neg low \neq false$. The *effect* of BT_{ρ_3} has a similar theoretical derivation procedure with BT_{ρ_2}, which is represented as $\bigvee_{r\in\{1,\cdots,r_j\}}\bigvee_{i\in\{1,\cdots,m\}}\{\mathrm{mX}[j][r][i] \wedge \neg low,\ \mathrm{mX}[j][r][i]' \wedge jx' = j\}$, where $\mathrm{mX}[j][r][i] \wedge \neg low \neq false$.

Now, we could calculate the *effect* of BT_{eff} inductively by the $\mathscr{F}[\![.]\!]$ function. Subsequently, it is not difficult to find that the *effect* f of BT_{eff} is a subset of the transition ρ in the original FDS from GR(1). Consequently, we could conclude that executions controlled by BT_{eff} always belong to executions in FDS, while FDS $\vDash \psi$ is sound and complete, thus, $BT_{\mathit{eff}} \vDash \psi$ is also sound. □

Let us discuss the complexity of our method. It consists of two procedures: the GR(1) realizability check and the BT construction. The former has a time complexity of $O(mn|\sigma|^2)$, where σ denotes the set of all variables assignments in ψ. The latter is implemented via a deepest three-level loop (c.f., Alg. 1 and Alg. 2), resulting in a complexity of $O(mnk)$. Recall that k is the iteration times for calculating μY, while m and n are the numbers of environmental assumptions and system guarantees, respectively. Consequently, the overall computational complexity of our method is the sum of these two components, i.e., $O(mn|\sigma|^2 + mnk)$, which is a polynomial complexity. Indeed, the time cost of the realizability check in our method can be quite high. However, it is worth noticing that compared to the 2EXPTIME of checking general LTL formulas, GR(1) has reduced its complexity to polynomials while retaining sufficient expressive power. Therefore, the time required for realizability checks is unavoidable and our work mainly focuses on improving the efficiency of constructing behavior trees and the efficiency of BTs' execution.

V. Experimental Evaluation

We implemented our method, NaiveBT, and RawBT on the GR(1) synthesis tool *Spectra*[1]. Each execution node in the BT holds a small BDD, and they are combined into a whole with *ControlFlow* nodes. During execution, we utilize a blackboard implemented as a dictionary and provide basic read-and-write

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

functionality for storing intermediates from BDD calculation. Similar to the *Spectra* tool, both BDD libraries, JTLV and CUDD, are accessible. We opted for CUDD (version 3.0) in all our experiments due to its stability. Experimental materials, including executable JAR package, specification files, and demonstration videos, can be found on Zenodo repository [15].

To evaluate, we have four research questions: **RQ1** How about the performance of the NaiveBT and RawBT methods? **RQ2** Does the EffBT method perform better in synthesis and BT execution? Furthermore, how does the size of specifications affect its efficiency? **RQ3** How about the quality of BTs synthesized from EffBT? **RQ4** Do pruning strategies bring improvements to our method?

A. Scenarios and Datasets

1) The FrozenLake+ Scenarios: Fig. 7 demonstrates the Frozen Lake game in Gymnasium [16]. 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 reach 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.

Fig. 7: The Frozen^{+} scenario with 8×8 grid map

2) AMBA [17] and GenBuf [18]: Both the ARM's AMBA AHB arbiter (AMBA) and IBM's Generalized Buffer (GenBuf) from an IBM tutorial are among the most widely recognized GR(1) specifications for evaluation. In our experiment, the master number is set to 1, 3, 5, 7, and 9 for AMBA, while the request number is confined to 10, 30, 50, 70, and 90 for GenBuf. With the increase in counts of the masters/requests, the complexity of specifications also increases.

3) SYNTECH Datasets [19][20]: We opted for the SYNTECH datasets for thorough evaluations of our method. It is a comprehensive suite of specifications that covers diverse ranges of tasks and owns several iterations. We selected five distinct versions, namely SYNTECH 15, 17, 19, 20, and 21, encompassing over 300 task specifications in GR(1).

B. Experimental Setups and Evaluation Metrics

All experiments are performed on a Ubuntu 22.04 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 in each scenario and each setting, and the final results of evaluated metrics are the averages among them. Our experiments are divided into two main parts, along with an ablation study.

In the first part, we intend to assess the synthesis efficiency and the simulated execution performance of the synthesized outcomes. We will compare our EffBT method with RawBT, NaiveBT, and two representative GR(1) synthesis methods: the Static [12] method (denoted as Spectra) and the Just-in-time [13] method (denoted as JITS), respectively. Note that we only make the qualitative comparison to the closely related works [9][10] (c.f., Sec. I) due to neither of their runnable artifacts nor their task specifications are publicly accessible.

We execute all these methods across the aforementioned scenarios and datasets using identical settings. Throughout the construction, we record three evaluation metrics. First, the metric *#Nodes* denotes the sum of nodes when checking realizability, which indicates the scale of the 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 the Spectra, RawBT, and EffBT methods is the time spent on Step 2, Step 3′, and Step 2′ in Fig. 2 respectively. Note that JITS is a dynamic method that has no construction phase thus without construction time correspondingly.

Apart from synthesis efficiency, we then give attention to the execution performance of the generated results (BTs, Static FDS, or JITS Controllers), while few previous studies focused on this aspect. We execute those synthesized controllers five times in simulated scenarios and calculate their average. Each time, we measure a total decision-making time of five thousand steps. In each execution, the environment and system randomly select their next actions from a set of available options. For instance, in the Frozen^{+} scenario, the beast randomly chooses to move either up or left to chase the player, if both options are available, and the player then games too. Notably, the execution time for JITS only accounts for the step time and excludes the initial loading time. Additionally, we omit the physical action-taking time (if any, occurs in physical simulations or real-world scenarios). This timing involves low-level implementations of actual actions (e.g., the movement of a robot arm), which are not relevant to our method, while we preserve the action-taking time used in the BDD-level updates.

Then in the second part, we evaluate the quality of the synthesized BTs from the aspects of their structures. The metric *BT Size* refers to the total number of nodes in the generated BTs, inclusive of both *ControlFlow* and *Execution* nodes. As an important advantage of BTs, modularity ensures that BTs are easy to read and reusable, hence, we introduce an additional metric to estimate this property, called *Balance*, 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 the subtrees under the root. The latter is defined as $\exp\left(-|N_{exec}/N_{ctrl} - 2|\right)$, in which N_{exec} and N_{ctrl} denote the sum of *Execution* nodes and *ControlFlow* nodes, respectively. The metric *Node Balance* measures the ratio of *Execution* nodes to *ControlFlow* nodes, ideally aiming for a value close to 2. This guideline follows the manual design principles outlined in [21], which suggests that a BT with one control node and two branches is much easier to read and reuse. Both node balance and structural balance are considered better when their values are closer to 1. The metric *Balance* is defined as the sum of them.

Moreover, to address RQ4, we conduct an ablation study by disabling all pruning strategies in EffBT, forming the *EffBT#* method. We then evaluate this method and record the metrics following the same procedure as in the two main parts above. In addition, we conducted the entire experiment on *RawBT*$^+$ (the RawBT method with pruning strategies). The results of this experiment can be found in [15].

C. Synthesis Efficiency and Execution Performance Results

1) Results of the NaiveBT Method: Table. II demonstrates step-by-step evaluation results of the NaiveBT method under two Frozen$^+$ settings. As can be seen in the table, 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 did not 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. II. 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 is not efficient enough, which consumes a lot of time and memory while yielding huge and low-quality BTs.

2) Results of Other Methods (Synthesis): Table. III summarizes the evaluated results of the synthesis procedure for various methods, including Spectra, RawBT, EffBT#, and the EffBT, excluding JITS due to its dynamic nature without a construction phase. In this table, the time is measured in *seconds*, and the **bolded** indicates the best result among other methods. The mark *OOT* represents a process that exceeded the maximum allowed time, *N/A* denotes that there are no available values, possibly caused by out of time during the construction phase. Additionally, for each metric evaluated on the SYNTECH datasets, we represent its value as the average of all tasks within each dataset. Additionally, as illustrated in Table. III, the construction time of RawBT is not restricted by the static FDS construction (Spectra) since we did not follow the expected procedure which obtains FDS first (Step 2) before creating the RawBT (Step 3′). Instead, the BTs were constructed directly from Step 1, but followed the construction of FDS as mentioned in Step 3′ during implementation.

TABLE II: 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

The results indicate that our method achieves the shortest construction time across all tasks compared to RawBT and Spectra, with average speedups of 73.8% and 84.7%, respectively. Whether or not EffBT employs pruning strategies, its performance consistently surpasses that of Spectra. Hence, the improvements in synthesis efficiency are mainly attributed to the decomposition of large BDDs. Furthermore, as task specifications become increasingly complex, the construction time of EffBT increases at a much slower rate than that of the other methods. This allows our method to tackle more challenging tasks, while Spectra and JITS struggle. For instance, the Spectra tool timed out in scenarios such as Frozen$^+$ 32×32, AMBA_9, and GenBuf_30 to GenBuf_90, the JITS tool also timed out in the Frozen$^+$ 32×32 scenario (leading to a N/A when executing). In contrast, our method produced the vast majority of results within minutes and outperformed other methods in terms of construction speed.

3) Results of Other Methods (Execution): The simulated execution results of the outcomes produced by Spectra, JITS, RawBT, EffBT, and EffBT# are incorporated in Table. III. In this table, the execution time is an average of the sum of five thousand decision-making steps, measured in *seconds*. We did not execute the BTs synthesized by NaiveBT because their sizes are extremely large (over 100,000).

When comparing the results of EffBT with Spectra and JITS, it is evident that our method significantly reduces decision-making time, showing average performance improvements of 49.1% and 84.5% during randomly simulated executions. Additionally, the execution of EffBT is minimally impacted by the complexity of specifications. This can be attributed to the well-organized small BDDs by using BTs, as well as the pruning of unnecessary BDDs. In scenarios like Frozen$^+$, AMBA, and GenBuf, the step time of Spectra and JITS increases as task complexity rises, whereas EffBT maintains consistent execution times at a smaller scale. Then, the execution results for RawBT show that it has the weakest decision-making process when compared to the EffBT and EffBT# methods. The disparity becomes even more evident

TABLE III: The Synthesis and Execution Results of Spectra, JITS, RawBT, EffBT#, and our EffBT Methods

Task		#Nodes ($\times 10k$)	Time$_{RC}$ (sec)	Construction Time (sec)				Execution Time (sec)				
				Spectra	RawBT	EffBT	EffBT#	Spectra	JITS	RawBT	EffBT	EffBT#
Frozen$^+$	8 × 8	0.25	0.05	0.03	0.03	**0.02**	0.03	0.09	0.07	**0.06**	**0.06**	0.07
	16 × 16	6.37	2.20	13.71	16.13	**11.80**	12.03	0.35	0.21	0.26	**0.05**	0.07
	24 × 24	49.22	31.19	280.62	1089.12	**258.06**	271.94	0.78	0.51	0.67	**0.05**	0.04
	32 × 32	118.12	131.96	OOT	5711.29	**2126.25**	2193.02	N/A	N/A	1.28	**0.03**	0.05
AMBA	AMBA_1	0.19	0.08	0.04	0.02	**0.003**	0.01	**0.03**	0.15	0.04	**0.03**	**0.03**
	AMBA_3	4.09	6.96	31.91	5.64	**0.48**	0.49	**0.03**	0.92	0.05	0.04	0.04
	AMBA_5	14.21	59.11	1511.72	47.38	**0.31**	1.71	0.05	2.03	0.06	**0.03**	0.05
	AMBA_7	27.05	332.83	4370.71	81.46	**0.91**	1.89	0.06	3.25	0.10	**0.02**	0.06
	AMBA_9	65.15	894.92	OOT	187.32	**44.77**	48.04	N/A	6.93	0.25	**0.01**	0.20
GenBuf	GenBuf_10	0.51	0.13	0.23	0.49	**0.04**	0.10	0.34	0.55	0.64	**0.05**	0.34
	GenBuf_30	1.99	1.75	OOT	10.38	**0.74**	1.43	N/A	2.26	0.82	**0.05**	0.49
	GenBuf_50	4.12	6.66	OOT	53.05	**3.18**	8.86	N/A	5.84	1.09	**0.05**	0.59
	GenBuf_70	7.24	17.03	OOT	212.69	**5.35**	15.61	N/A	11.03	0.72	**0.05**	0.49
	GenBuf_90	10.94	42.49	OOT	363.15	**9.70**	29.55	N/A	17.26	0.82	**0.05**	0.66
SYNTECH	SYNTECH 15	0.20	0.06	0.04	0.04	**0.01**	0.02	0.13	0.34	0.19	**0.10**	0.12
	SYNTECH 17	1.42	2.57	1.37	6.39	**0.22**	1.05	0.58	0.91	0.78	**0.18**	0.21
	SYNTECH 19	0.79	0.08	2.20	157.43	**0.96**	1.18	3.38	8.95	3.82	**3.01**	11.33
	SYNTECH 20	0.83	0.49	6.69	2.91	**0.05**	0.13	1.01	1.57	1.31	**0.46**	0.55
	SYNTECH 21	5.11	22.82	13.15	150.97	**1.21**	4.14	1.55	2.96	2.04	**0.22**	0.29

as the runtime increases. This phenomenon confirms our earlier perspective on RawBT, i.e., its structure inherently adds unnecessary ticks, which results in wasted execution time.

Answer to RQ2: Our method outperforms others in both the construction and execution efficiency of synthesized results, showing significant improvements in time consumption compared to RawBT, Spectra, and JITS. Meanwhile, its performance is less affected by the complexity of specifications.

D. Behavior Tree Quality Results

Table. IV presents the quality evaluation results of the BTs synthesized using the RawBT, EffBT, and EffBT# methods. Firstly, considering the size of BTs, our method demonstrates a considerable reduction of nearly half compared to RawBT, with an average reduction comes to 40.3%. Then, regarding the BT balance, we observe that although RawBT has similar or better node balances compared to EffBT, it is lacking in structural balance. This indicates that the sizes of the subtrees under their root nodes vary significantly, which are not well-designed BTs. Taking the Frozen$^+$ 32×32 as an example, the structural balance of RawBT is 0.04, due to the minimum and maximum sizes of the subtrees being 157 and 3,910 respectively, resulting in a 25-fold difference between them, which demonstrates an extreme structural imbalance among the subtrees. In comparison, EffBT has a structural balance of 0.73 (83/113). Our method fixes this drawback and achieves a better structural balance and tree size, which benefits from the design of structurally similar subtrees. The average of the structural balance of RawBT is 0.12, while for the EffBT method, it rises to 0.68. Regarding the balance of nodes, which estimates the balance of the *ControlFlow* nodes and the *Execution* nodes, EffBT performs similarly to RawBT, with average values of 0.57 and 0.58, respectively. Therefore, in summary, our method demonstrates improved balance and modularity compared to RawBT. **Answer to RQ3:** Our method is capable of producing high-quality BTs. The generated BTs are smaller in size owning to our pruning strategies, and exhibit improved balance and modularity thanks to the construction algorithm. These factors contribute to more efficient executions of BTs.

In addition to the quality results of RawBT, combining the previous analysis of it in aspects of synthesis efficiency and simulated execution performance (c.f., Sec. V-C2 and Sec. V-C3), we could conclude, **Answer to RQ1 (RawBT):** The RawBT method has a long construction time compared to ours during BT synthesis but produces BT with reduced quality, particularly in the imbalance among subtrees, which adversely affects the execution performance of these BTs.

E. Ablation Study Results

We investigate the practical effect of our pruning strategies by comparing EffBT# (EffBT without Pruning Strategies) with the EffBT method in terms of construction, simulated execution, and BT structures. Likewise, the experimental results are summarized in Table. III and Table. IV.

Concerning construction time, the findings indicate that in various scenarios, including AMBA, GenBuf, and the larger SYNTECH dataset, the pruning strategy consistently leads to a construct time reduction, with an average decrease of 46.4%. Then, regarding the execution results of pruned and unpruned BTs (c.f., Table. III), the unpruned BTs always incur longer decision-making costs, as the time is wasted on checking meaningless condition nodes and switching between redundant structures. At last, in reference to the quality results presented in Table. IV, it is noteworthy that pruning brings an average size reduction of 56.1% compared to the unpruned version, and it performs consistently well across all tasks. As for the balance of BTs, the results suggest that the impact of pruning is somewhat limited. The overall balance of EffBT and EffBT# remains nearly the same, while the node balance of the pruned BTs (EffBT) is slightly improved.

Answer to RQ4: The pruning strategies are crucial for reducing the size of BTs and accelerating both the construction and BT execution periods. Removing them from our method would significantly degrade performance.

TABLE IV: Behavior Tree Quality Results of RawBT, EffBT#, and our EffBT Methods

Task		BT Size			Balance (Structural + Nodes')		
		RawBT	EffBT	EffBT#	RawBT	EffBT	EffBT#
Frozen$^+$	8 × 8	89	**42**	95	0.80 (0.12+0.68)	**1.69** (**1.00**+**0.69**)	1.45 (**1.00**+0.45)
	16 × 16	839	**408**	939	0.93 (0.07+**0.86**)	**1.63** (**0.82**+0.81)	1.29 (0.81+0.48)
	24 × 24	3464	**1698**	3924	0.96 (0.05+**0.91**)	**1.62** (**0.76**+0.86)	1.24 (0.75+0.49)
	32 × 32	7861	**3871**	8966	0.97 (0.04+**0.93**)	**1.62** (**0.73**+0.89)	1.22 (**0.73**+0.49)
AMBA	AMBA_1	185	**124**	170	0.61 (0.11+0.50)	**1.10** (**0.44**+**0.66**)	0.89 (0.39+0.50)
	AMBA_3	473	**354**	442	0.58 (0.09+0.49)	**0.82** (**0.32**+**0.50**)	0.80 (0.31+0.49)
	AMBA_5	741	**552**	694	0.59 (0.09+**0.50**)	**0.82** (**0.32**+**0.50**)	0.81 (0.31+**0.50**)
	AMBA_7	1009	**750**	946	0.59 (0.09+**0.50**)	**0.82** (**0.32**+**0.50**)	0.81 (0.31+**0.50**)
	AMBA_9	1277	**948**	1198	0.58 (0.09+0.49)	**0.82** (**0.32**+**0.50**)	0.81 (0.31+**0.50**)
GenBuf	GenBuf_10	531	**330**	484	0.60 (0.13+0.47)	**1.22** (**0.74**+**0.48**)	1.16 (0.68+**0.48**)
	GenBuf_30	1471	**910**	1344	0.61 (0.13+**0.48**)	**1.22** (**0.74**+**0.48**)	1.16 (0.68+**0.48**)
	GenBuf_50	2411	**1490**	2204	0.62 (0.13+**0.49**)	**1.21** (**0.74**+0.47)	1.16 (0.68+0.48)
	GenBuf_70	3351	**2070**	3064	0.63 (0.13+**0.50**)	**1.21** (**0.74**+0.47)	1.16 (0.68+0.48)
	GenBuf_90	4291	**2650**	3924	0.64 (0.13+**0.51**)	**1.21** (**0.74**+0.47)	1.16 (0.68+0.48)
SYNTECH	SYNTECH 15	109	**55**	100	0.65 (0.17+**0.48**)	**1.35** (**0.88**+0.47)	1.34 (**0.88**+0.46)
	SYNTECH 17	433	**286**	426	0.74 (0.13+**0.61**)	**1.32** (**0.75**+0.57)	1.28 (**0.75**+0.53)
	SYNTECH 19	453	**249**	431	0.77 (0.26+**0.51**)	**1.36** (**0.90**+0.46)	1.34 (0.89+0.45)
	SYNTECH 20	334	**144**	331	0.75 (0.17+**0.58**)	**1.30** (**0.78**+0.52)	1.29 (**0.78**+0.51)
	SYNTECH 21	315	**161**	338	0.81 (0.19+**0.63**)	**1.46** (**0.85**+0.61)	1.27 (0.84+0.43)

VI. Related Works

Over the past few decades, several methodologies, including Evolutionary Algorithms [1][2], Reinforcement Learning [3], Large Languages Models [5][6], Back-chaining [22], Monte-Carlo Tree Search [23], and Learning from Demonstrations [7][8][24], have been employed for synthesizing BTs and have been applied on various domains such as robot arms and unmanned aerial vehicles. Compared to the learning-based or searching-based methods above, reactive synthesis, which generates controllers that are guaranteed to be correct by construction from formal specifications, exhibits significant advantages in correctness guarantee.

There are two prevailing categories in the context of generating BTs through reactive synthesis: general-LTL-based and F-LTL-based. The former, such as TAMP[4], models the robot-environment interactions as a transition system while transforming task specifications (described by LTL) into Büchi automata simultaneously. Then, it searches for a counter-example path in the product of the transition system and the Büchi automaton. If findable, the final BT is subsequently constructed to equivalently embody the counter-example path. However, due to the EXPTIME complexity of converting LTL to Büchi automata, these methods inevitably suffer problems such as state space explosion and exponential time complexity.

The latter, operated on LTL fragments instead of general LTL, avoids the intractable complexity (at least EXPTIME). Such as [9], it contains six types of LTL patterns which are responsible for different properties, including concepts like '*globally stands*' and '*response to a signal*'. For each LTL pattern, there is a corresponding BT template. The BT synthesis process involves combining those templates following the manner of the provided LTL specifications. However, this approach invariably yields a flat BT structure with subtrees of varying sizes, due to the absence of support for nested formulas and the irregularity of BT patterns. Although each subtree correlated with a pattern carries a specific meaning, the assembled behavior tree remains challenging to comprehend. Similar limitations were also noticed in [10]. Attributed to the regularity of GR(1) intermediates, the subtrees in our method have similar structures but are responsible for different justice goals, avoiding the limitations in the aforementioned methods. This design not only aligns with the experience accumulated from previous manual designs but also adheres to software engineering principles, including modularity, performance, readability, and maintainability.

Additionally, compared to the Static [12] and the Just-in-time GR(1) [13] controller synthesizer, our method decomposes the large BDDs into smaller BDDs and combines them using BTs in a well-structured way while keeping the consistency of semantics.

VII. Conclusion

We propose an efficient framework for the reactive synthesis and execution of BTs, termed EffBT, which fixes the drawback of irregular BTs found in earlier studies stemming from their calculation and construction methods. Our approach unifies them into GR(1) realizability check and proposes an algorithm for constructing highly modularized BTs. As a result, we can enhance the execution efficiency of the generated BTs by utilizing the *Parallel* node, while few focused on this aspect. We proved the soundness of EffBT, and experimental results across various scenarios and datasets demonstrate that our method outperforms others in both synthesis speed and quality.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (Grant No.62032024 and No.62106278).

REFERENCES

[1] M. Iovino, J. Styrud, P. Falco, and C. Smith, "Learning behavior trees with genetic programming in unpredictable environments," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 4591–4597.

[2] A. Neupane and M. A. Goodrich, "Learning swarm behaviors using grammatical evolution and behavior trees," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*. ijcai.org, 2019, pp. 513–520.

[3] R. Dey and C. Child, "QL-BT: enhancing behaviour tree design and implementation with q-learning," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[4] S. Li, D. Park, Y. Sung, J. A. Shah, and N. Roy, "Reactive task and motion planning under temporal logic specifications," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE Press, 2021, p. 12618–12624.

[5] Y. Cao and C. S. G. Lee, "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)*, vol. 3433, 2023.

[6] F. Li, X. Wang, B. Li, Y. Wu, Y. Wang, and X. Yi, "A study on training and developing large language models for behavior tree generation," *CoRR*, vol. abs/2401.08089, 2024.

[7] L. Scherf, K. Fröhlich, and D. Koert, "Learning action conditions for automatic behavior tree generation from human demonstrations," in *Companion of the 2024 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '24. Association for Computing Machinery, 2024, p. 950–954.

[8] K. French, S. Wu, T. Pan, Z. Zhou, and O. C. Jenkins, "Learning behavior trees from demonstration," in *International Conference on Robotics and Automation, ICRA 2019*. IEEE, 2019, pp. 7791–7797.

[9] M. Colledanchise, R. M. Murray, and P. Ögren, "Synthesis of correct-by-construction behavior trees," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017*. IEEE, 2017, pp. 6039–6046.

[10] T. G. Tadewos, A. A. Redwan Newaz, and A. Karimoddini, "Specification-guided behavior tree synthesis and execution for coordination of autonomous systems," *Expert Systems with Applications*, vol. 201, p. 117022, 2022.

[11] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012, in Commemoration of Amir Pnueli.

[12] S. Maoz and J. O. Ringert, "Reactive synthesis with spectra: A tutorial," in *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion*. IEEE, 2021, pp. 320–321.

[13] S. Maoz and I. Shevrin, "Just-in-time reactive synthesis," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 2020, pp. 635–646.

[14] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, p. 293–318, sep 1992.

[15] *EffBT: An Efficient Behavior Tree Reactive Synthesis and Execution Framework (Supplementary Materials)*. Zenodo, Nov. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.14241343

[16] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023.

[17] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Interactive presentation: Automatic hardware synthesis from specifications: a case study," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '07. EDA Consortium, 2007, p. 1188–1193.

[18] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Specify, compile, run: Hardware from PSL," in *Proceedings of the Workshop on Compiler Optimization meets Compiler Verification, COCV@ETAPS 2007*, ser. Electronic Notes in Theoretical Computer Science, S. Glesner, J. Knoop, and R. Drechsler, Eds., vol. 190, no. 4. Elsevier, 2007, pp. 3–16.

[19] D. Ma'ayan and S. Maoz, "Using reactive synthesis: An end-to-end exploratory case study," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 742–754.

[20] R. Shalom and S. Maoz, "Which of my assumptions are unnecessary for realizability and why should i care?" in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 221–232.

[21] M. Colledanchise and P. Ögren, "Behavior trees in robotics and AI: an introduction," *CoRR*, vol. abs/1709.00084, 2017.

[22] Z. Cai, M. Li, W. Huang, and W. Yang, "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 Press, 2021, pp. 6058–6065.

[23] W. Hong, Z. Chen, M. Li, Y. Li, P. Huang, and J. Wang, "Formal verification based synthesis for behavior trees," in *Dependable Software Engineering. Theories, Tools, and Applications - 9th International Symposium, SETTA 2023*, vol. 14464. Springer, 2023, pp. 72–91.

[24] S. Gugliermo, E. Schaffernicht, C. Koniaris, and F. Pecora, "Learning behavior trees from planning experts using decision tree and logic factorization," *IEEE Robotics Autom. Lett.*, vol. 8, no. 6, pp. 3534–3541, 2023.