Multi-modal Sketch-Based Behavior Tree Synthesis

WENMENG ZHANG, National University of Defense Technology, China
ZHENBANG CHEN*, National University of Defense Technology, China
WEIJIANG HONG, National University of Defense Technology, China

Behavior trees (BTs) are widely adopted in the field of agent control, particularly in robotics, due to their modularity and reactivity. However, constructing a BT that meets the desired expectations is time-consuming and challenging, especially for non-experts. This paper presents BtBot, a multi-modal sketch-based behavior tree synthesis technique. Given a natural language task description and a set of positive and negative examples, BtBot automatically generates a BT program that aligns with the natural language description and meets the requirements of the examples. Inside BtBot, an LLM is employed to understand the task's natural language description and generate a sketch of the task execution. Then, BtBot searches the sketch to synthesize a candidate BT program consistent with the user-provided positive and negative examples. When the sketch is proven to be incapable of generating the target BT, BtBot provides a multi-step repairing method that modifies the control nodes and structure of the sketch to search for the desired BT. We have implemented BtBot in a prototype and evaluated it on a benchmark of 70 tasks across multiple scenarios. The experimental results indicate that BtBot outperforms the existing BT synthesis techniques in effectiveness and efficiency. In addition, two user studies have been conducted to demonstrate the usefulness of BtBot.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Behavior Tree, Sketch

ACM Reference Format:
Wenmeng Zhang, Zhenbang Chen, and Weijiang Hong. 2025. Multi-modal Sketch-Based Behavior Tree Synthesis . *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 400 (October 2025), 28 pages. https://doi.org/10.1145/3763178

1 Introduction

Behavior trees (BTs) provide a highly modular and hierarchical structure that is natural for modeling control and decision-making. A behavior tree (BT) is a tree in which the leaf nodes are *action* nodes or *condition* nodes and non-leaf nodes are control nodes. Due to their superior reactivity, modularity, readability and scalability, BTs play an important role in controlling agent behavior [Marcotte and Hamilton 2017; Marzinotto et al. 2014]. Initially, BTs were used to control the behavior of non-player characters (NPCs) in video games [Fu et al. 2016; Marcotte and Hamilton 2017; Sekhavat 2017]. Nowadays, BTs are widely used in various domains for behavior control and enabling complex decision-making, such as robotics [Colledanchise and Ögren 2018; Ögren and Sprague 2022] and intelligent driving [Olsson 2016]. However, when the task complexity increases, the BTs also

*Zhenbang Chen is the corresponding author.

Authors' Contact Information: Wenmeng Zhang, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, wenmengzhang@nudt.edu.cn; Zhenbang Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, zbchen@nudt.edu.cn; Weijiang Hong, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, hongweijiang17@nudt.edu.cn.

ACM 2475-1421/2025/10-ART400
https://doi.org/10.1145/3763178

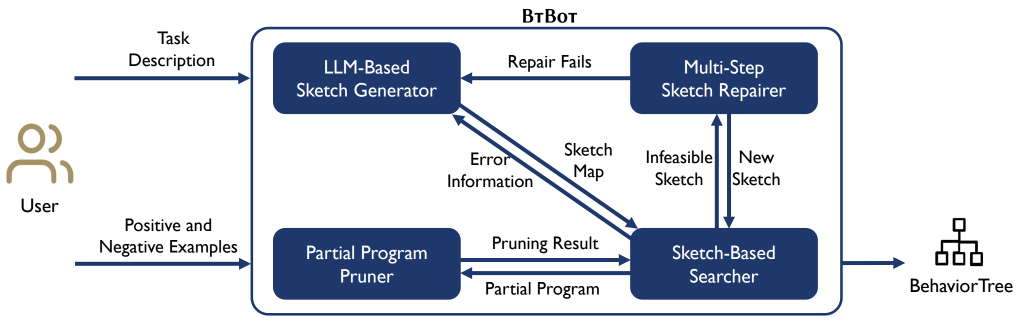


Fig. 1. Overview of BtBot.

become more and more complex, which makes developing a BT satisfying requirements a highly challenging task [Scheide et al. 2021].

Program synthesis [Gulwani et al. 2017] provides a method for automatically generating a program satisfying the requirements (or specifications). Existing synthesis methods for BTs [Cai et al. 2021; Chen et al. 2024; Hong et al. 2023; Izzo et al. 2024] vary in specification forms and search methods. Some existing approaches synthesize BTs from scratch and employ heuristic search algorithms (*e.g.*, genetic algorithm [Ångström 2022] and Monte Carlo tree search [Hong et al. 2023]) for the target BTs satisfying different kinds of specifications, such as those based on linear temporal logic (LTL [Pnueli 1977]) [Biggar and Zamani 2020] or dynamic logic (DL [Harel et al. 2001]) [Paxton et al. 2019]. These approaches enjoy the generality and automation of synthesis but suffer from efficiency problems. Additionally, there are planning-based learning algorithms [Cai et al. 2021] for automatically generating BTs, which need detailed specifications of the actions for BT inference. Planning-based methods are efficient by sacrificing generality and need more user engagement. Furthermore, approaches [Izzo et al. 2024; Vemprala et al. 2024] leveraging fine-tuned LLMs for BT generation can accept natural language specifications and generate BTs efficiently, but the generated results may be affected by LLM hallucinations. Until now, the abstraction and its algorithms for achieving the balance between efficiency and user engagement for synthesizing desired BTs are still challenging.

Our insight into BT synthesis is that natural language descriptions of requirements are more suitable for BT developers. Besides, the action and condition sequences generated by BTs serve as a more natural synthesis specification for developers, providing a key abstraction for BT synthesis. Based on this insight, we aim to provide a synthesis method that accepts the requirement description in natural language and the specification in action and condition sequences and outputs a BT that fulfills the requirement and is consistent with the specification. For instance, even a developer with little knowledge of BTs can control a household robot to complete a charging task by generating a BT through a simple task description, *e.g.*, *"If the battery is low, navigate to the charging station and start charging"* along with the desired behavior sequence ⟨batteryLow, goStation, Charge⟩.

Our intention naturally falls into multi-modal program synthesis [Chen et al. 2020; Rahmani et al. 2021], which needs to consider two kinds of specifications, *i.e.*, natural language task description and sequence examples. Inspired by programming by example (PBE) [Feser et al. 2015; Gulwani and Jain 2017] and sketch-based program synthesis techniques [Solar-Lezama 2008, 2009], we propose a sketch-based multi-modal BT synthesis technique called BtBot. The key idea of BtBot is to generate a sketch from the natural language task description and efficiently synthesize the target BT based on the sketch with respect to the sequence examples. We leverage an LLM for sketch generation to tackle the challenge of generating sketches from natural language descriptions. Figure 1 shows the workflow of BtBot. First, the user provides a natural language task description along with specifications of positive and negative examples. Next, the LLM generates a sketch based on the task description. Subsequently, BtBot performs an enumerative search on the sketch to find a complete BT consistent with the positive and negative example specifications. During the search process, BtBot prunes partially completed BTs to improve search efficiency. If generating a target BT from the sketch is impossible, BtBot enters the repair phase. BtBot employs a multi-step repair process in which many new candidate sketches will be generated to search for the target BT. If it is still impossible to find a target BT, the LLM is invoked again to generate a new sketch. The process is repeated until a target BT is found or timeout.

Due to LLM hallucinations and the vast program search space, we need to tackle two technical problems inside BtBot: (1) the efficiency of sketch-based searching of the target BT and (2) the satisfiability and feasibility of the sketch generated by LLMs. For the first problem, we propose a partial BT pruning method according to the positive and negative examples, *i.e.*, all the positive examples should be possible to be generated, and none of the negative examples should be generated. Specifically, we propose over-approximation and under-approximation methods for the partial BT and check their consistency with the positive and negative examples. For the second problem, we propose a multi-step repair method for gradually fixing the sketch. By modifying the assignments of control nodes, we repair errors in the sketch's control nodes caused by the LLM's misinterpretation of the task's logical relationships. Additionally, by expanding the structure of the sketch, we repair errors caused by missing information due to the LLM's incomplete understanding of the task. If the target BT cannot be found, the LLM is called to generate another sketch.

We have implemented BtBot in a prototype tool and evaluated it on a benchmark of 70 tasks in four scenarios. The experimental results show that BtBot can synthesize the BTs for 96.3% benchmark tasks. Moreover, compared to state-of-the-art BT synthesis methods [Hong et al. 2023; Izzo et al. 2024], including techniques based on fine-tuning LLMs and traditional search-based BT synthesis approaches, BtBot demonstrates superior performance in terms of both success rate and efficiency. Finally, we conducted two user studies on BtBot, one using BtBot once to solve tasks, and the other using BtBot to solve tasks iteratively based on feedback. The results showed that using BtBot helped users successfully solve more tasks, with all participants in both user studies successfully solving more than 80% of the tasks. Most participants believed that building BTs using BtBot was easier and more user-friendly than building BTs manually.

In summary, the contributions of this paper are as follows.

- We propose a sketch-based multi-modal synthesis technique for generating BTs from the task description in natural language and examples. To the best of our knowledge, we are the first to utilize PBE and sketch-based program synthesis techniques for BT constructing tasks.
- We propose a new sketch-based synthesis engine BtBot for BT synthesis that (1) leverages LLM for sketch generation, (2) applies an over-approximation and under-approximation method for partial BT to prune the search space, and (3) designs a multi-step repair method for fixing the sketch in case of infeasibility.
- We have implemented BtBot and evaluated it on 70 representative benchmark tasks. The results demonstrate the effectiveness and efficiency of BtBot, successfully synthesizing BTs for 96.3% of the benchmark tasks. Compared with the state-of-the-art baseline [Hong et al. 2023; Izzo et al. 2024] and LLM-based BT synthesis approaches, BtBot achieves a higher success rate in solving benchmark tasks.
- We have also conducted two user studies to evaluate BtBot's benefits to expert users and non-expert users. The results indicate that compared with manual construction, BtBot improves the success rate of constructing BTs, and 83% of participants find it easier and more convenient to

construct BTs using BtBot once. All participants believe that iteratively using BtBot to build BTs based on feedback is more effective and helpful for revising BTs.

2 Overview

In this section, we introduce the relevant background knowledge of behavior trees (BTs), followed by a motivating example to present the proposed technique.

2.1 Behavior Tree

BTs are primarily composed of control nodes and execution nodes. Execution nodes, which form the tree's leaf nodes, are further categorized into *action* nodes and *condition* nodes. Action nodes can return one of three possible states: *success*, *failure*, and *running*, whereas condition nodes can only return *success* and *failure*. Control nodes, which serve as non-leaf nodes in a BT, must have at least two child nodes and a specific type [Colledanchise and Ögren 2018], such as sequence nodes, fallback nodes, and parallel nodes, to name a few. In this paper, we primarily focus on sequence nodes and fallback nodes. The sequence and fallback nodes execute their child nodes from left to right in order. A sequence node returns success only if all of its child nodes return success and immediately returns failure/running if any child node returns failure/running. A fallback node returns success/running if any child node returns success/running and returns failure if all child nodes return failure; if a child node returns failure, the fallback node continues to execute the next node.

The BT in Figure 2 controls a robot performing an automatic charging task. The robot first attempts to execute the charging task in the left subtree; if the battery's level is not low or the charging task fails, it performs the other task represented by the right subtree (doOtherTask). In more detail, the charging task first checks whether the current battery level is low. If the battery is low (batteryLow), the robot will autonomously navigate to the charging station (goStation) and start to charge (Charge).

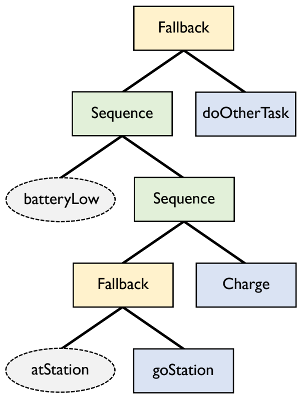


Fig. 2. The behavior tree for autonomous charging task.

2.2 Illustration of BtBot

Next, we illustrate BtBot step by step to generate a BT for the autonomous charging task.

Description	*When the robot is in a low battery state, it should check whether it is at the charging station or move to the charging station and finally start charging; otherwise, continue performing other tasks.*
Positive Examples	⟨batteryLow, goStation, Charge⟩, ⟨batteryLow, atStation, Charge⟩, ⟨batteryLow, doOtherTask⟩, ⟨doOtherTask⟩
Negative Examples	⟨shutOff⟩

Fig. 3. Autonomous charging task. Task description, positive behaviors, and negative behaviors for the task.

Positive and Negative Examples. Inspired by the work [Robertson and Watson 2015], we define behavior sequences that the user expects the BT to generate as positive examples, and those that must be prohibited as negative examples. A behavior sequence comprises the actions and conditions from initiating a task to a specified time point. For instance, users can provide a task description and a set of positive and negative examples for the autonomous charging task, as illustrated in Figure 3. *It is important to note that user can provide only a subset of the expected key behavior sequences to generate a BT consistent with the task description. Such under-specifications are allowed.*

Expected Output. Our objective is to automate the generation of BTs by extracting essential information from the natural language descriptions of tasks. Figure 2 shows the target BT for the charging scenario. The construction of the BT necessitates the extraction of relevant information from the task description, which is subsequently organized into the BT structure according to the specified control logic. When the task description is incomplete or ambiguous, positive examples can enhance the description and generate a BT that aligns more closely with user expectations. Positive and negative examples are important for generating BTs and the validation process, ensuring that the generated BT adheres to user specifications. Specifically, the BT must cover all the positive examples while avoiding any negative examples. For the task in Figure 3, we expect the generated BT in Figure 2 and the BT program to be as follows, where the nodes within parentheses belong to a subtree, and the symbol '$\fatsemi$'/'$\triangleright$' represents a 'sequence'/'fallback' node.

$$(\mathsf{batteryLow} \fatsemi (\mathsf{atStation} \triangleright \mathsf{goStation}) \fatsemi \mathsf{Charge}) \triangleright \mathsf{doOtherTask}$$

Synthesis Challenges. The inherent ambiguity of natural language presents challenges [Jia et al. 2025] for automatically generating BTs. The descriptions often lack precision and may be subject to multiple interpretations. For complex tasks, natural language is inadequate in specifying the tasks [Willems et al. 2016]. Furthermore, as task complexity increases, the corresponding BT becomes more complex, *i.e.*, both the number of nodes and the structural complexity increase, making it challenging to coordinate and even understand the subtasks of the BT. Hence, synthesizing a BT that meets the given requirements from scratch becomes challenging due to the lack of specification and the task's complexity.

BtBot Approach. To address these challenges, we leverage an LLM to comprehend the task description and generate a BT sketch [Solar-Lezama 2008, 2009]. This process involves decomposing complex tasks into simpler subtasks and explicitly defining the logical relationships among them, thereby enhancing the LLM's capacity to understand and extract relevant task information. Subsequently, we introduce abstraction techniques for partial BTs, which prune the search space and boost the search process. Finally, we propose a sketch repair technique for infeasible sketches, which utilizes the failed positive examples as guidance to refine the sketch, ultimately generating a BT that adheres to the positive and negative examples requirements.

Sketch Generation Method. The LLM generates a sketch with control nodes that describes the task execution process. Given natural language's inherent ambiguity and vagueness, the LLM may not understand the task description accurately. To mitigate this issue, we propose a task-specific language pattern [de Boer et al. 2024; Liu et al. 2024] tailored for robot control tasks, which aids the LLM in better understanding the task requirements. For instance, based on the task description presented in Figure 3, the LLM generates the following sketch for the motivating example, where $\square_c$ and $\square_a$ represent condition and action leaf nodes, respectively.

$$(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$$

Program Search. Based on the sketch [Solar-Lezama 2008, 2009] generated by the LLM [Chen et al. 2023], BtBot searches for the target program, which is an enumerative search process. BtBot completes each hole in the sketch with different candidates to generate multiple candidate BTs, aiming to find the BTs consistent with the given specifications. To speed up the search process, we introduce an abstraction technique [Wang et al. 2017] for partial programs [Barnaby et al. 2023]. This technique improves efficiency by pruning the search space and avoiding the satisfaction checking of all BT programs. For the auto-charging task in Figure 3, given the sketch $(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$, we generate a partially filled BT $(\mathsf{batteryLow} \fatsemi (\mathsf{batteryLow} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$. Since it is not possible to generate the example $\langle \mathsf{batteryLow}, \mathsf{atStation}, \mathsf{Charge} \rangle$ from this partial BT, we stop completing

$$\begin{array}{rcll}
a & \in & \mathbb{A}^* & := \{\ldots\} \\
c & \in & \mathbb{C}^* & := \{\ldots\} \\
\phi & \in & \mathbb{E} & := c \mid \neg c \\
\mathcal{P}, Q & \in & \mathbb{P} & := a \mid \phi \mid \mathcal{P} \triangleright Q \mid \mathcal{P} \fatsemi Q \mid (\mathcal{P})
\end{array}$$

Fig. 4. The syntax of BT programs. $\mathbb{A}$ and $\mathbb{C}$ represent the action and condition node sets, respectively. The superscript $*$ means that $\mathbb{A}$ and $\mathbb{C}$ are customized by users. $\triangleright$ and $\fatsemi$ denote fallback and sequence nodes.

this partial program and instead explore other potential candidate programs. Using this sketch-based enumerative search method, BtBot can generate a BT that meets the positive and negative examples, as shown in Figure 2.

Sketch Repair. If BtBot cannot find a target BT from the sketch, it starts to repair the sketch. We propose a multi-step sketch repair technique [Yaghmazadeh et al. 2017]. BtBot first modifies the control nodes within the sketch and then expands the sketch structure based on failed positive examples [Feng et al. 2018; Le et al. 2017]. To illustrate the repair technique, we suppose that the sketch of the task in Figure 3 generated by the LLM is $(\square_c \fatsemi (\square_c \fatsemi \square_a) \fatsemi \square_a) \triangleright \square_a$, based on which BtBot fails to produce a behavior tree consistent with the specifications given by the positive and negative examples in Figure 3. In the first step, BtBot attempts to repair the sketch by modifying the control nodes, leading to the following feasible sketch.

$$(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$$

Based on this sketch, BtBot can generate a BT consistent with the specifications. However, if the sketch generated by LLM is $(\square_c \fatsemi \square_a \fatsemi \square_a) \triangleright \square_a$, neither the enumeration search nor the modification of the control nodes can find the target BT. In this case, BtBot modifies the sketch's structure according to the failed positive example $\langle \text{batteryLow}, \text{atStation}, \text{Charge} \rangle$, resulting in the following partial BT based on which the target BT program can be synthesized.

$$(\square_c \fatsemi (\text{atStation} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$$

If the multi-step repair technique fails, BtBot will re-engage the LLM to generate a new sketch.

3 Behavior Tree Program

This section will describe the syntax and semantics of the domain-specific language (DSL) for BTs. Then, we will formally define the synthesis problem for BTs.

3.1 DSL Syntax

Given an action set $\mathbb{A}$ and a condition set $\mathbb{C}$, Figure 4 shows the syntax of the DSL for BTs. There are two basic cases for BTs: each action a is a single-node BT, and a condition c or its negation $\neg c$ also forms a single-node BT. The actions typically represent skills that control the robot's behavior by invoking an internal API. Conditions represent an abstraction of the robot's state at a particular point during the task execution, such as its position ('atStation'). Then, there are two control operators for composition, *i.e.*, $\fatsemi$ and $\triangleright$, which represent sequence and fallback compositions, respectively. Besides, we introduce $(\mathcal{P})$ to alleviate the ambiguity in composition. Please note that the sets $\mathbb{A}$ and $\mathbb{C}$ need to be customized for different scenarios by users. For example, for the BT in Figure 2, $\mathbb{A}$ is {goStation, Charge, doOtherTask, shutOff}, and $\mathbb{C}$ is {batteryLow, atStation}.

3.2 DSL Semantics

The execution of a BT is usually tick-driven [Biggar et al. 2021]. The BT's root node receives the tick first, and then each node will pass the received tick to its child nodes according to the node's type. When a leaf node, *i.e.*, an action or a condition node, receives the tick, the real job will be

$$
\begin{array}{rcl}
[\![a]\!] & := & \{\langle\rangle, \langle a\rangle, \langle a, \checkmark\rangle, \langle !\rangle, \langle ?\rangle\} \\
[\![c]\!] & := & \{\langle\rangle, \langle c\rangle, \langle c, \checkmark\rangle, \langle !\rangle\} \\
[\![\neg c]\!] & := & \{\langle\rangle, \langle \neg c\rangle, \langle \neg c, \checkmark\rangle, \langle !\rangle\} \\
[\![\mathcal{P} \triangleright Q]\!] & := & ([\![\mathcal{P}]\!] \setminus \{t \cdot ! \mid t \cdot ! \in [\![\mathcal{P}]\!]\}) \cup \{t_1 \cdot t_2 \mid t_1 \cdot ! \in [\![\mathcal{P}]\!] \wedge t_2 \in [\![Q]\!]\} \\
[\![\mathcal{P} \fatsemi Q]\!] & := & ([\![\mathcal{P}]\!] \setminus \{t \cdot \checkmark \mid t \cdot \checkmark \in [\![\mathcal{P}]\!]\}) \cup \{t_1 \cdot t_2 \mid t_1 \cdot \checkmark \in [\![\mathcal{P}]\!] \wedge t_2 \in [\![Q]\!]\} \\
[\![(\mathcal{P})]\!] & := & [\![\mathcal{P}]\!]
\end{array}
$$

Fig. 5. The trace-based denotational semantics for BT DSL. The label (✓,!, ?) represents the BT's status.

carried out, and the status (or result) will be returned to the parent node, which returns the status to its parent according to the statuses of its child nodes. Hence, driven by different ticks, the execution of a BT may have different results, and completing a task may need multiple ticks. In this paper, we concentrate on the behavior of one tick because the behavior of BT inside a tick is more natural for human understanding. More specifically, we define the semantics of a BT as the possible behaviors inside one tick.

We use $\Sigma = \mathbb{A} \cup \mathbb{E}$ as the *alphabet* set, and Σ^* denotes the set of finite sequences (called *traces*) over Σ. For two traces s and t in Σ^*, we use $s \cdot t$ to represent the concatenation of s and t. Besides, we use $\checkmark$, !, and ? to represent the success, failure, and running statuses of BT execution, and $\Omega = \{\checkmark, !, ?\}$ as the status set. For brevity, we use $s \cdot \omega$ to represent $s \cdot \langle\omega\rangle$, where $\omega \in \Omega$. We call the sequences in $\{s \cdot \omega \mid s \in \Sigma^* \wedge \omega \in \Omega\}$ *terminated traces.* Then, the semantics of a BT $\mathcal{P}$ is given by the following function, where $\Sigma_\Omega^\circledast = \Sigma^* \cup \{s \cdot \omega \mid s \in \Sigma^* \wedge \omega \in \Omega\}$ is the set of possible traces composed by the alphabets in Σ and the statuses in Ω, including the terminated traces.

$$[\![\mathcal{P}]\!] : \mathbb{P} \rightarrow 2^{\Sigma_\Omega^\circledast} \tag{1}$$

According to the execution logic of the BT nodes in Section 2.1, Figure 5 shows the DSL semantics.

An action node can result in one of three states: success, failure, or running. Hence, its semantics contain three cases, *i.e.*, $\langle a, \checkmark\rangle$, $\langle !\rangle$, and $\langle ?\rangle$. Correspondingly, a condition node can result in two cases, *i.e.*, $\langle c, \checkmark\rangle$, $\langle !\rangle$. The right subtree is only executed for the BT rooted by a fallback node if the left subtree returns a failure. All the traces that lead to a failure in the left subtree will serve as the prefixes to those produced by the right subtree. The left and right subtrees of a fallback node are not interchangeable—switching them results in a BT with different semantic traces. For example, the BT in Figure 2 is capable of generating the trace ⟨batteryLow, doOtherTask⟩. However, if the left and right subtrees of the root node are swapped, the resulting BT would no longer be able to generate this trace. Conversely, when a sequence node is the root node of the BT, the successful execution of the left subtree is a necessary condition for the execution of the right subtree. Here, the semantics of a BT is *prefix-closed* because we want the semantics to incorporate the intermediate execution information.

3.3 Problem Statement

This section formally defines the BT synthesis problem.

Definition 3.1. **(Specification)** The synthesis specification Ψ is a tuple $(\mathcal{E}^+, \mathcal{E}^-)$, where $\mathcal{E}^+ \subseteq \Sigma^*$ is the set of positive examples, and $\mathcal{E}^- \subseteq \Sigma^*$ is the set of negative examples. Each example is a sequence of alphabets.

Definition 3.2. **(Consistency)** Given a BT $\mathcal{P}$ and a specification $\Psi = (\mathcal{E}^+, \mathcal{E}^-)$, $\mathcal{P}$ is consistent with Ψ, denoted by $\mathcal{P} \models \Psi$, iff all positive examples must be generated by $\mathcal{P}$ and no negative example could be generated.

$$\mathcal{E}^+ \subseteq [\![\mathcal{P}]\!] \wedge \mathcal{E}^- \cap [\![\mathcal{P}]\!] = \emptyset \tag{2}$$

Definition 3.3. **(Problem)** Given a specification Ψ and a natural language description of a task $\mathcal{D}$, the synthesis problem of BT is to produce a program $\mathcal{P} \in \mathbb{P}$ such that $\mathcal{P} \models \Psi$ and thus makes $\mathcal{P}$ consistent with $\mathcal{D}$.

4 Synthesis Algorithm

This section presents the algorithms for the BT synthesis problem. We begin with an overview of the top-level algorithm and then describe its key components.

4.1 Sketch Language and Partial Program

The BT synthesis algorithm relies on a specific representation of partial programs. A partial program's syntax includes the syntax in Figure 4 by extending the program with holes, abbreviated as follows, where $\square_c$ and $\square_a$ represent the holes for condition and action nodes, respectively.

$$\mathcal{P}, \mathcal{Q} \quad \in \quad \mathbb{P} \quad := \quad a \mid \phi \mid \square_c \mid \square_a \mid \mathcal{P} \triangleright \mathcal{Q} \mid \mathcal{P} \fatsemi \mathcal{Q} \mid (\mathcal{P}) \tag{3}$$

Consequently, we extend the original denotational semantics by giving the semantics of holes.

$$\begin{aligned} [\![\square_c]\!] &:= \{\langle\rangle, \langle\square_c\rangle, \langle\square_c, \checkmark\rangle, \langle !\rangle\} \\ [\![\square_a]\!] &:= \{\langle\rangle, \langle\square_a\rangle, \langle\square_a, \checkmark\rangle, \langle !\rangle, \langle ?\rangle\} \end{aligned} \tag{4}$$

To facilitate the presentation, we define a partial program as an abstract syntax tree defined as follows. For brevity, we extend the definition of Σ and its related definitions to incorporate holes, *i.e.*, $\{\square_c, \square_a\} \subseteq \Sigma$.

Definition 4.1. **(Partial Program)** A partial program $\mathcal{P}$ is a *binary* tree (V, E, Π, r), where

- V is the set of nodes in $\mathcal{P}$, and $r \in V$ is the root node.
- $E \subseteq V \times V \times \mathbb{N}$ is the set of relations in $\mathcal{P}$, and each element $(v_1, v_2, n) \in E$ means that v_2 is the nth child node of v_1 (*i.e.*, v_1 is the parent node of v_2). Besides, we require that each node can only have two child nodes or none, *i.e.*, the size of each node v's child node set $\{v_1 \mid (v, v_1, n) \in E\}$ (denoted by $\mathsf{Child}(v)$) is 2 or 0.
- $\Pi : V \longrightarrow \Sigma \cup \{\triangleright, \fatsemi\}$ is the labeling function that gives each node its label and satisfies $\forall v \in V \bullet \#\mathsf{Child}(v) = 2 \longrightarrow \Pi(v) \in \{\triangleright, \fatsemi\}$, *i.e.*, the labels of the nodes that have child nodes are control labels.

Given a partial program $\mathcal{P} = (V, E, \Pi, r)$, we give the following definitions.

- We call the nodes with hole labels (*i.e.*, $\{\square_c, \square_a\}$) *hole nodes*, and the nodes with control labels (*i.e.*, $\{\triangleright, \fatsemi\}$) *control nodes.*
- We use $\mathsf{Leaf}(\mathcal{P})$ to denote the leaf node set of $\mathcal{P}$, *i.e.*, $\{v \mid v \in V \wedge \#\mathsf{Child}(v) = 0\}$.
- A partial program is *complete* (denoted by $\mathsf{Complete}(\mathcal{P})$) if none of its nodes has a hole label, *i.e.*, $\forall v \in V \bullet \Pi(v) \notin \{\square_c, \square_a\}$.

Since our synthesis algorithm incrementally fills holes with actions or conditions, we define an operation for updating partial programs to represent the process of filling holes.

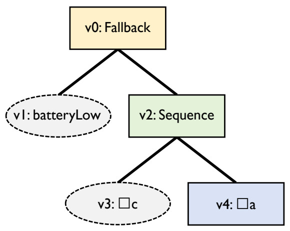


Fig. 6. The partial syntax tree corresponding to partial program batteryLow $\triangleright$ ($\square_c \fatsemi \square_a$).

Definition 4.2. **(Program Update)** Given a partial program $\mathcal{P} = (V, E, \Pi, r)$, one of its nodes v and a label l, we use $\mathcal{P}^{\Pi}[v \triangleleft l]$ to represent the new partial program after updating v's label to l.

We use $\mathcal{P}^{\Pi}[v_1 \triangleleft l_1, ..., v_n \triangleleft l_n]$ as a short hand for the program after n updates to $\mathcal{P}$.

Algorithm 1 LLMBasedBTSynthesizer($\Psi, \mathbb{A}, \mathbb{C}, \mathcal{D}$)

```
1:  input: A specification $\Psi$, the action set $\mathbb{A}$, the condition set $\mathbb{C}$, and the description $\mathcal{D}$ of BT's
    task in natural language.
2:  output: a BT $\mathcal{P}$ such that $\mathcal{P} \models \Psi$ or $\perp$ when a timeout occurs.
3:  $\mathcal{FS} \leftarrow \emptyset$
4:  while True do
5:      $(\mathcal{S}, \mathcal{M}) \leftarrow$ LLMBasedSketchGen($\mathcal{D}, \mathbb{A}, \mathbb{C}$)            ▷ $\mathcal{M}$ : Leaf($\mathcal{S}$) $\rightarrow \Sigma^*$
6:      if $\mathcal{S}$'s syntax is valid then
7:          $\mathcal{P} \leftarrow$ SketchBasedSearch($\mathcal{S}, \mathcal{M}, \Psi$)
8:          if $\mathcal{P} \neq \perp$ then
9:              return $\mathcal{P}$
10:         $\mathcal{FS} \leftarrow \mathcal{FS} \cup \mathcal{S}$
11:         for all $\mathcal{S}' \in$ SketchSpace($\mathcal{S}$) do
12:             $\mathcal{P}' \leftarrow$ SketchBasedSearch($\mathcal{S}', \mathcal{M}, \Psi$)
13:             if $\mathcal{P}' \neq \perp$ then
14:                 return $\mathcal{P}'$
15:             $\mathcal{FS} \leftarrow \mathcal{FS} \cup \mathcal{S}'$
16:         while $\mathcal{FS} \neq \emptyset$ do
17:             $\mathcal{S}_f \leftarrow \arg\max_{\mathcal{S}_i \in \mathcal{FS} \wedge \text{Match}(\mathcal{E}^+, \mathcal{S}_i) \subset \mathcal{E}^+} \#\text{Match}(\mathcal{E}^+, \mathcal{S}_i)$    ▷ $\text{Match}(\mathcal{E}^+, \mathcal{S}_i) := \{t \mid \mathcal{S}_i \vdash t \wedge t \in \mathcal{E}^+\}$
18:             $\mathcal{FS} \leftarrow \mathcal{FS} \setminus \{\mathcal{S}_f\}$
19:             $\mathcal{P}' \leftarrow$ RepairSketch($\mathcal{S}_f, \mathcal{M}, \Psi$)
20:             if $\mathcal{P}' \neq \perp$ then
21:                 return $\mathcal{P}'$
22: return $\perp$
```

Example 1. Figure 6 depicts the partial program batteryLow $\triangleright (\square_c \,\fatsemi\, \square_a)$ as a tree, with each node annotated by its corresponding label. In this tree, the label of node $v1$ is batteryLow. This partial program can be obtained from the partial program $\mathcal{P} = \square_c \triangleright (\square_c \,\fatsemi\, \square_a)$ by $\mathcal{P}^{\Pi}[v1 \triangleleft \text{batteryLow}]$.

LEMMA 4.1. *If a trace s* ***without any holes*** *belongs to the semantics of a partial BT $\mathcal{S}$ with holes, then s belongs to the semantics of the new BT after updating $\mathcal{S}$'s any hole node.*

Proof. This lemma holds because the semantics is prefix-closed.

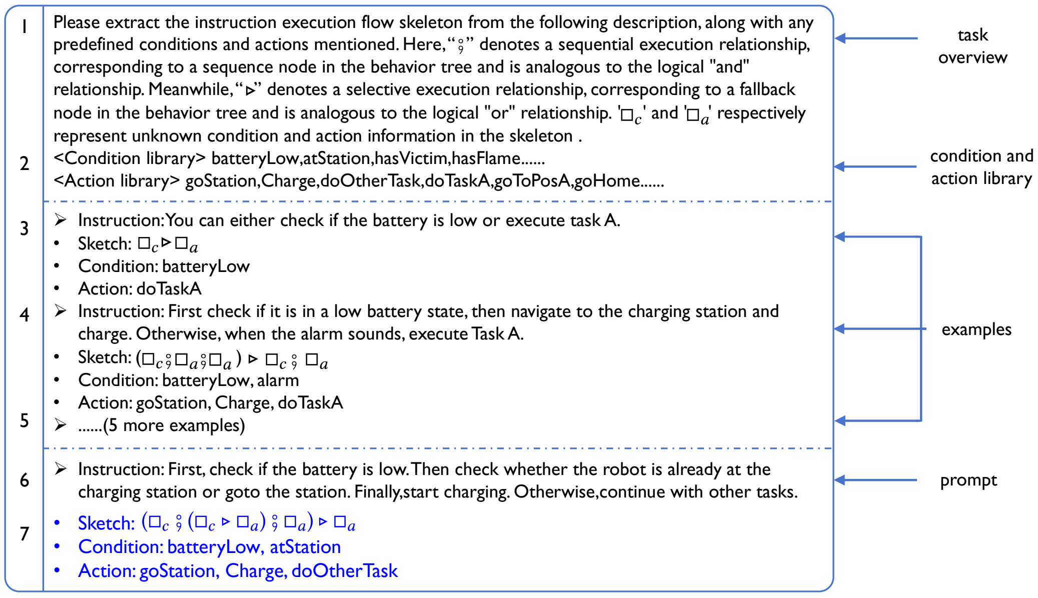
Our algorithm performs the search based on a sketch generated by the LLM. Based on the above definitions, we provide the formal definition of a sketch.

Definition 4.3. **(Sketch)** A sketch $\mathcal{S}$ is a partial program $\mathcal{P} = (V, E, \Pi, r)$ in which all the leaf nodes are condition or action hole nodes, *i.e.*, $\forall v \in \text{Leaf}(\mathcal{P}) \bullet \Pi(v) \in \{\square_c, \square_a\}$.

4.2 Top Level Algorithm

Algorithm 1 gives the top-level algorithm. The inputs are a natural language description of the task $\mathcal{D}$, a specification $\Psi = (\mathcal{E}^+, \mathcal{E}^-)$, and the action and condition sets $\mathbb{A}$ and $\mathbb{C}$. The algorithm returns a BT program if it finds the one that is consistent with the specification Ψ; otherwise, it returns $\perp$, *i.e.*, the synthesis fails due to the timeout or exceeding the maximum limit of LLM calls, omitted for brevity.

At a high level, the algorithm generates a sketch and attempts to complete it. If the sketch cannot produce a BT program consistent with the specifications, the algorithm attempts to repair



1 Please extract the instruction execution flow skeleton from the following description, along with any predefined conditions and actions mentioned. Here, "$⨟$" denotes a sequential execution relationship, corresponding to a sequence node in the behavior tree and is analogous to the logical "and" relationship. Meanwhile, "$\triangleright$" denotes a selective execution relationship, corresponding to a fallback node in the behavior tree and is analogous to the logical "or" relationship. '$\square_c$' and '$\square_a$' respectively represent unknown condition and action information in the skeleton .

2 <Condition library> batteryLow,atStation,hasVictim,hasFlame......
<Action library> goStation,Charge,doOtherTask,doTaskA,goToPosA,goHome......

3
- ➢ Instruction:You can either check if the battery is low or execute task A.
- • Sketch: $\square_c \triangleright \square_a$
- • Condition: batteryLow
- • Action: doTaskA

4
- ➢ Instruction: First check if it is in a low battery state, then navigate to the charging station and charge. Otherwise, when the alarm sounds, execute Task A.
- • Sketch: $(\square_c ⨟ \square_a ⨟ \square_a) \triangleright \square_c ⨟ \square_a$
- • Condition: batteryLow, alarm
- • Action: goStation, Charge, doTaskA

5
- ➢ ......(5 more examples)

6
- ➢ Instruction: First, check if the battery is low.Then check whether the robot is already at the charging station or goto the station. Finally,start charging. Otherwise,continue with other tasks.

7
- • Sketch: $(\square_c ⨟ (\square_c \triangleright \square_a) ⨟ \square_a) \triangleright \square_a$
- • Condition: batteryLow, atStation
- • Action: goStation, Charge, doOtherTask

Fig. 7. Few-shot prompting method for generating BT sketch examples from the task description. The highlighted output content will be in blue.

the sketch. If the repaired sketch can be instantiated successfully, the target BT will be returned. Otherwise, the process is repeated until a BT program consistent with the specifications is found or timeout (omitted for brevity).

More specifically, the algorithm begins by invoking LLMBasedSketchGen (Line 5, *c.f.*, Section 4.3), which queries the LLM to generate a sketch corresponding to the task description. Apart from a sketch, LLM extracts information about actions and conditions from the task description. Using this information, the algorithm provides a list of filling suggestions $\mathcal{M}$ for each hole node. Then, SketchBasedSearch attempts to search for a target BT from the sketch that is consistent with the specification (Line 7). If the search fails, the algorithm repairs the sketch by modifying the labels of control nodes (Lines 11 to 15, *c.f.*, Section 4.5.1) or applying the failure example-based repairing (Lines 16 to 21, *c.f.*, Section 4.5.2). If multi-step repair techniques still fail to obtain the target BT, the LLM is called to regenerate a different sketch to search the target BT.

4.3 Sketch Generation

To improve the sketch generation, BtBot employs few-shot prompting [Gu et al. 2022] and language patterns [de Boer et al. 2024] to enhance the LLM's understanding of the task description.

4.3.1 Few-Shot Prompting. The prompt interacting with the LLM first summarizes the task to be solved by the LLM, along with conditions and actions. It then presents seven example instructions with their expected outputs to clarify the task for the LLM further. Finally, the prompt describes the instruction the LLM needs to address. Figure 7 illustrates the prompt for the automatic charging task in Figure 3, where the content in blue represents the output generated by the LLM in response to the automatic charging task.

4.3.2 Language Pattern. While LLMs can accurately understand simple tasks, their performance often deteriorates on complex tasks. Due to the inherent ambiguity and vagueness of natural

Algorithm 2 SketchBasedSearch($\mathcal{S}, \mathcal{M}, \Psi$)

```
1: input: $\mathcal{S}$ is a sketch, $\mathcal{M} : \mathsf{Leaf}(\mathcal{S}) \to \Sigma^*$ is generated by LLM and gives each leaf node a
   sequence of suggested actions or conditions, and $\Psi = (\mathcal{E}^+, \mathcal{E}^-)$ is the specification.
2: output: A complete BT program $\mathcal{P}$ such that $\mathcal{P} \models \Psi$ or $\perp$ represents no BTs consistent with $\Psi$
   can be completed by $\mathcal{S}$.
3: $\mathcal{W} \leftarrow \{\mathcal{S}\}$
4: while $\mathcal{W} \neq \emptyset$ do
5:     $\mathcal{S}_c \leftarrow \arg\max_{\mathcal{S}_i \in \mathcal{W}} \sum_{v_i \in \mathsf{Leaf}(\mathcal{S}_i)} \mathsf{Reward}(v_i, \Pi_i, \mathcal{M}, \mathcal{E}^+, \mathcal{E}^-)$    ▷ $\mathcal{S}_i = (V_i, E_i, \Pi_i, r_i)$
6:     $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathcal{S}_c\}$
7:     if $\mathsf{Complete}(\mathcal{S}_c) \wedge \mathcal{S}_c \models \Psi$ then
8:         return $\mathcal{S}_c$
9:     if $\mathcal{S}_c \vdash \mathcal{E}^+ \wedge [\![\mathcal{S}_c]\!] \cap \mathcal{E}^- = \emptyset$ then
10:        $v \leftarrow \mathsf{GetNextHoleLeafNode}(\mathcal{S}_c)$    ▷ Get the first unfilled hole in depth-first manner
11:        $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{S}_c^{\Pi}[v \triangleleft e] \mid e \in \Sigma_{\Pi(v)}\}$    ▷ $\Sigma_{\Pi(v)}$ is $\mathbb{A}$ or $\mathbb{E}$ when $\Pi(v)$ is $\square_a$ or $\square_c$
12: return $\perp$
```

language [Jia et al. 2025], flattened descriptions of complex tasks make it difficult to convey intent clearly. This ambiguity, combined with LLMs' limited understanding of complex tasks [Yuan et al. 2024], frequently leads to incorrect sketch generation. We designed a BT domain-specific language pattern [Liu et al. 2024; Muñoz-Ortiz et al. 2024] to enhance LLMs' ability to understand task descriptions. Users may *optionally* employ this pattern to express complex tasks more precisely, thereby improving LLMs' understanding of the tasks.

Inspired by decomposed prompting [Khot et al. 2023; Ma et al. 2024], complex tasks are manually decomposed by the user into multiple subtasks [Zhang et al. 2021]. Then, the user specifies the logical relationships between these subtasks (tasks to be executed conditionally or in sequence). The pattern example for the automatic charging task in Figure 3 is as follows.

The robot selectively executes the following branch tasks:
Task 1: The robot sequentially executes the following sequence tasks:
First, check if the battery is low. Then check whether the robot is already at the charging station or navigate to the charging station. Start charging.
Task 2: *Continue performing other tasks.*

The output of LLM consists of two components: *the sketch text* and *a set of condition and action nodes* extracted from the task description. These two parts are extracted using regular expressions. The sketch text is parsed by a *parser* to obtain the BT sketch $\mathcal{S}$, which is used in the search process. If the sketch text contains syntax errors or LLM outputs gibberish, the parser will report an error and the LLM will be re-prompted. Given the extracted condition and action nodes, the algorithm provides filling suggestions $\mathcal{M}$ for each hole in the sketch based on its type and index.

4.4 Sketch-Based Program Search

The sketch-based algorithm explores different completions of the sketch to find a complete BT program consistent with the specification Ψ. The inputs of Algorithm 2 are a sketch $\mathcal{S}$, the action and condition suggestions for holes $\mathcal{M}$, and a specification Ψ. The algorithm outputs a BT consistent with Ψ if found; otherwise, it outputs $\perp$, *i.e.*, proves that $\mathcal{S}$ is impossible to generate a BT consistent with Ψ, regardless of how the holes in $\mathcal{S}$ are filled.

In more detail, the algorithm implements the search in a worklist manner. At the beginning, the sketch is added to the worklist $\mathcal{W}$ (Line 3). Based on $\mathcal{M}$ and Ψ, the next partial program $\mathcal{S}_c$ to

$$\frac{e \in \mathbb{A} \cup \mathbb{E}}{e \vdash e}\ (\textsc{Basic}) \qquad \frac{a \in \mathbb{A}}{\Box_a \vdash a}\ (\textsc{Action-Type}) \qquad \frac{c \in \mathbb{E}}{\Box_c \vdash c}\ (\textsc{Condition-Type})$$

$$\frac{}{\langle\rangle \vdash \langle\rangle}\ (\textsc{Seq-Empty}) \qquad \frac{\mathsf{Seq}_1 = \langle e_1\rangle \cdot \mathsf{Seq}_1' \quad \mathsf{Seq}_2 = \langle e_2\rangle \cdot \mathsf{Seq}_2' \quad e_1 \vdash e_2 \quad \mathsf{Seq}_1' \vdash \mathsf{Seq}_2'}{\mathsf{Seq}_1 \vdash \mathsf{Seq}_2}\ (\textsc{Seq})$$

$$\frac{\exists s \in [\![\mathcal{S}]\!] \bullet s \vdash \mathsf{Seq}}{\mathcal{S} \vdash \mathsf{Seq}}\ (\textsc{Consist-Seq})$$

$$\frac{\forall \mathsf{Seq} \in \mathcal{E}^+ \bullet \mathcal{S} \vdash \mathsf{Seq} \wedge \mathcal{S}^{\Pi}[\mathsf{Seq}] \vdash \mathcal{E}^+ \setminus \{\mathsf{Seq}\}}{\mathcal{S} \vdash \mathcal{E}^+}\ (\textsc{Consist-Tree})$$

Fig. 8. The over-approximation rules. The symbol $\vdash$ indicates the consistency of over-approximation. The symbol $\bullet$ serves as a separator between quantifiers and predicates.

be filled is selected (Line 5), where we prioritize the partial BTs in $\mathcal{W}$ according to their reward values, which are the sum of the reward values of all leaf nodes. The Reward function at Line 5 for each leaf node is defined as follows, where $\mathsf{Len}(\mathsf{Seq})$ represents the sequence Seq's length, $\mathsf{Seq}[i]$ represents the sequence Seq's ith element and i starts from 0, $\Pi(v) \mathrel{\hat{\in}} \mathcal{E}^+$ means that there is a trace in $\mathcal{E}^+$ that contains $\Pi(v)$, and $\hat{\notin}$ represents the negation of $\hat{\in}$.

$$\mathsf{Reward}(v, \Pi, \mathcal{M}, \mathcal{E}^+, \mathcal{E}^-) := \begin{cases} 0, & \Pi(v) \in \{\Box_c, \Box_a\} \\ \mathsf{Len}(\mathcal{M}(v)) - i + 2, & \Pi(v) = \mathcal{M}(v)[i] \\ 2, & \Pi(v) \mathrel{\hat{\in}} \mathcal{E}^+ \\ -2, & \Pi(v) \mathrel{\hat{\in}} \mathcal{E}^- \wedge \Pi(v) \mathrel{\hat{\notin}} \mathcal{E}^+ \\ 1, & otherwise \end{cases} \tag{5}$$

Reward encourages the generated BT to align closely with the NL description of the task, satisfy all positive examples, and avoid any negative examples. Unique information in negative examples is assigned the lowest priority during the filling process. This mechanism can represent both behavior sequences that the BT must prohibit and actions that the robot should never execute at any time. For example, in the task illustrated in Figure 3, the negative example ⟨shutOff⟩ indicates that shutOff should never occur under any circumstances.

If $\mathcal{S}_c$ is complete and consistent with the specification (Line 7), the algorithm outputs $\mathcal{S}_c$ as the target BT program. Otherwise, the algorithm checks whether it is still possible to find a target BT based on $\mathcal{S}_c$ (Line 9) with respect to the specification's positive and negative examples. Here, the algorithm employs approximation techniques (*c.f.*, Section 4.4) to decide the possibility. If $\mathcal{S}_c$ is possible to generate a target BT, the algorithm uses GetNextHoleLeafNode to select the next unfilled hole in a depth-first manner (Line 10) and fills the hole with actions or conditions (Line 11), *i.e.*, generating new partial BT programs. Finally, if no program consistent with Ψ can be found, $\perp$ is returned, indicating that it is impossible to find a target BT based on $\mathcal{S}$, where we call $\mathcal{S}$ an *infeasible* sketch.

Approximation. Sketch-based search is an enumerative search technique. To improve the efficiency of enumerative search, we propose employing abstractions for pruning partial BT programs. Compared with previous work on BT synthesis [Cai et al. 2021; Chen et al. 2024; French et al. 2019; Gustavsson et al. 2022; Hong et al. 2023; Robertson and Watson 2015], *i.e.*, using a complete BT to determine whether it is consistent with the specifications, we determine the possibility of a partial program in searching the target BT early. If impossible, we will stop filling other holes, *i.e.*, no subsequent BT programs can be consistent with the specifications.

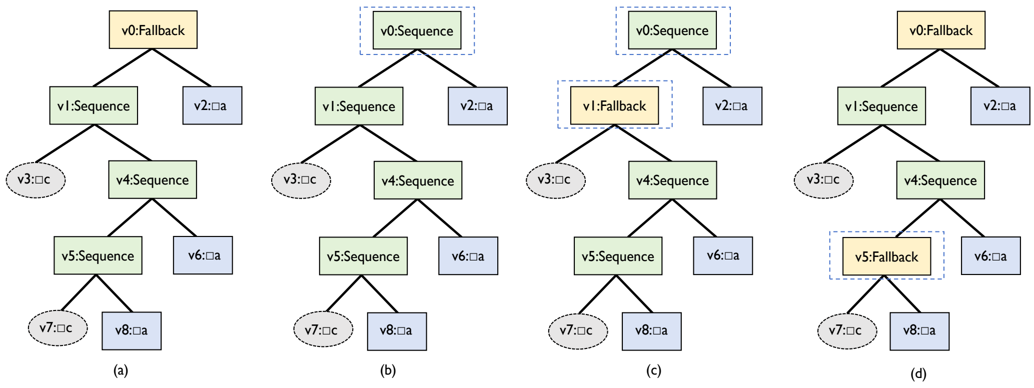


Fig. 9. (a) Represents an infeasible sketch $\mathcal{S}$ generated by the LLM: $(\square_c \fatsemi (\square_c \fatsemi \square_a) \fatsemi \square_a) \triangleright \square_a$. (b)(c)(d) are the sketches derived from (a) by modifying the labels of control nodes (as indicated by the blue dashed boxes), and (d) $(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$ satisfies the given specification.

Over-approximation. Here, we want to use over-approximation to check the possibility of generating all the positive specification examples. The over-approximation of partial programs can be seen as the set of all possible complete BTs generated by completing the partial programs. We use the denotational semantics of the partial program to check whether the program is consistent with the specifications. Figure 8 shows the over-approximation rules. Based on the characteristics of holes, condition (action) type holes can over-approximate any condition (action) (Condition-Type, Action-Type). Seq-Empty and Seq rules define the basic and induction cases for sequences, respectively. For a partial program $\mathcal{S}$, it is consistent with a sequence Seq when there exists a $\mathcal{S}$'s trace that is consistent with Seq (Consist-Seq), implying that Seq can be generated based on $\mathcal{S}$. For Consist-Tree, $\mathcal{S}$ is consistent with $\mathcal{E}^+$ if $\mathcal{S}$ is consistent with each positive example in $\mathcal{E}^+$, and by updating the labels of its n holes to generate any positive example will not affect the consistency of the updated partial program with other positive examples ($\mathcal{S}^{\Pi}[\text{Seq}] \vdash \mathcal{E}^+ \setminus \{\text{Seq}\}$). When Seq can be generated by $\mathcal{S}$ after filling $\mathcal{S}$'s n holes, *i.e.*, $[v_1 \triangleleft \text{Seq}_i, \ldots, v_n \triangleleft \text{Seq}_j]$, where Seq_i and Seq_j are the elements of Seq, $\mathcal{S}^{\Pi}[\text{Seq}]$ denotes the partial program after updating the n holes with the corresponding labels, *i.e.*, $\mathcal{S}^{\Pi}[v_1 \triangleleft \text{Seq}_i, \ldots, v_n \triangleleft \text{Seq}_j]$.

Under-approximation. Under-approximation ensures that no negative examples in the specification, including the intermediate behaviors of BTs, can be generated. Hence, the under-approximation of a partial BT $\mathcal{S}$ contains the *non-terminated* traces with holes in $\mathcal{S}$'s denotational semantics. So we can use $[\![\mathcal{S}]\!] \cap \mathcal{E}^- = \emptyset$ to ensure that no negative examples can be generated from $\mathcal{S}$.

Then, based on the approximation technique, we determine whether the target BT can be synthesized from a partial program (Line 9 of Algorithm 2). If it is deemed infeasible, the partial program is pruned, which significantly accelerates the search process.

$$\mathcal{S}_c \vdash \mathcal{E}^+ \wedge [\![\mathcal{S}_c]\!] \cap \mathcal{E}^- = \emptyset \tag{6}$$

Example 2. For the charging task in Figure 3, the LLM returns the sketch $(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$. After two iterations of the cycle, several partial programs are obtained. Suppose we select $\mathcal{S}_1$, *i.e.*, $(\text{batteryLow} \fatsemi (\text{batteryLow} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$, as the partial program for further completion. When performing the check at line 9, the over-approximation of $\mathcal{S}_1$ is not consistent with the trace $\langle \text{batteryLow}, \text{atStation}, \text{Charge} \rangle$ in $\mathcal{E}^+$, so $\mathcal{S}_1$ is pruned and cannot be further completed. We then select another partial program $\mathcal{S}_2$ $(\text{batteryLow} \fatsemi (\text{atStation} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$. Since $\mathcal{S}_2$ is consistent with the specification, $\mathcal{S}_2$ will be further filled. Finally, we obtain the target BT $(\text{batteryLow} \fatsemi (\text{atStation} \triangleright \text{goStation}) \fatsemi \text{Charge}) \triangleright \text{doOtherTask}$.

Algorithm 3 RepairSketch($\mathcal{S}, \mathcal{M}, \Psi$)

```
1:  input: $\mathcal{S} = (V, E, \Pi, r)$ is an infeasible sketch, $\mathcal{M} : \mathsf{Leaf}(\mathcal{S}) \to \Sigma^*$ is generated by LLM and
    gives each leaf node a sequence of suggested actions or conditions, and $\Psi = (\mathcal{E}^+, \mathcal{E}^-)$ is the
    specification.
2:  output: A complete BT program $\mathcal{P}$ such that $\mathcal{P} \models \Psi$ or $\bot$ representing that the repair fails.
3:  $\mathcal{E}_c^+ \leftarrow \{\mathsf{Seq} \mid \mathsf{Seq} \in \mathcal{E}^+ \wedge \mathcal{S} \vdash \mathsf{Seq}\}$
4:  $\mathcal{E}_f^+ \leftarrow \mathcal{E}^+ \setminus \mathcal{E}_c^+$
5:  $\mathcal{S}' \leftarrow \mathcal{S}$
6:  for each $\mathsf{Seq}_f \in \mathcal{E}_f^+$ do
7:      if $\mathcal{S} \vdash \mathcal{E}_c^+ \cup \{\mathsf{Seq}_f\} \wedge [\![\mathcal{S}]\!] \cap \mathcal{E}^- = \emptyset$ then
8:          continue
9:      $\mathsf{Seq}_c \leftarrow \arg\max_{\mathsf{Seq} \in \mathcal{E}_c^+} \mathsf{Similarity}(\mathsf{Seq}, \mathsf{Seq}_f)$
10:     $\langle e_1, \cdots, e_n \rangle \leftarrow \mathsf{Seq}_f \setminus \mathsf{Seq}_c$    ▷ Remove the common prefix and suffix of $\mathsf{Seq}_f$ and $\mathsf{Seq}_c$ from $\mathsf{Seq}_f$
11:     $\mathcal{S}_f, r_f \leftarrow \mathsf{ConstructSeq}(\langle e_1, \cdots, e_n \rangle)$
12:     for each $v \in \mathcal{S}.V$ do
13:         $\mathcal{S}_n \leftarrow (V \cup \{v_n\}, (E \cup \bigcup_{(v_p,v,m)\in E} \{(v_p, v_n, m)\} \cup \{(v_n, r_f, 1), (v_n, v, 2)\}) \setminus \bigcup_{(v_p,v,m)\in E} \{(v_p, v, m)\}, \Pi \cup \{v_n \mapsto \triangleright\}, r)$
14:         $\mathcal{S}_n \leftarrow (\mathcal{S}_n.V \cup \mathcal{S}_f.V, \mathcal{S}_n.E \cup \mathcal{S}_f.E, \mathcal{S}_n.\Pi \cup \mathcal{S}_f.\Pi, r)$
15:         $\mathcal{P}' \leftarrow \mathsf{SketchBasedSearch}(\mathcal{S}_n, \mathcal{M}, (\mathcal{E}_c^+ \cup \{\mathsf{Seq}_f\}, \mathcal{E}^-))$
16:         if $\mathcal{P}' \neq \bot$ then
17:             $\mathcal{E}_c^+ \leftarrow \mathcal{E}_c^+ \cup \{\mathsf{Seq}_f\}$
18:             $\mathcal{S}' \leftarrow \mathcal{S}_n$
19:             break
20:     if $\mathcal{S}' = \mathcal{S}$ then
21:         return $\bot$
22:     $\mathcal{S} \leftarrow \mathcal{S}'$
23: return $\mathcal{P}'$
```

4.5 Sketch Repair

Due to the inherent ambiguity of natural language, LLMs may misinterpret tasks or extract incomplete information, leading to incorrect node types or loss of critical information in the sketch. To address these issues, a multi-step repair procedure is employed for infeasible sketches: (1) modifying the control nodes (Lines 11 to 15 in Algorithm 1) and (2) expanding the sketch's structure (Lines 16 to 21 in Algorithm 1).

4.5.1 Control Node Repair. Since the LLM may not accurately comprehend the logical relationships in the natural language descriptions, the control nodes of the sketch $\mathcal{S}$ may be incorrect. For example, the LLM may incorrectly infer a sequential execution relationship as a selective execution relationship when understanding linguistic descriptions. That is, it may generate a sketch with a $\triangleright$ relationship, whereas the task description specifies a $\fatsemi$ relationship. Similar issues can also arise when BTs are designed by experts. To address this issue, we design a specialized repair strategy to correct errors in control nodes by swapping $\fatsemi$ with $\triangleright$ and $\triangleright$ with $\fatsemi$ in all possible combinations.

By modifying the n (from 0 to $\#\mathcal{S}.V \setminus \mathsf{Leaf}(\mathcal{S})$) control nodes in $\mathcal{S}$, we can get a set of new sketches, denoted as SketchSpace($\mathcal{S}$), defined as follows.

$$\mathsf{SketchSpace}(\mathcal{S}) := \{\mathcal{S}^{\Pi}[v_1 \triangleleft l_1, ..., v_n \triangleleft l_n] \mid \exists V_1 \subseteq (V \setminus \mathsf{Leaf}(\mathcal{S})) \bullet V_1 = \{v_1, ..., v_n\} \wedge l_i \in \{\triangleright, \fatsemi\}\} \quad (7)$$

These are all the possible combinations of assignments for the internal nodes in the sketch, using $\fatsemi$ and $\triangleright$ as labels. The SketchSpace contains 2^n different sketches when the infeasible sketch has n

internal nodes. BtBot searches for a feasible sketch within the SketchSpace, and then employs the SketchBasedSearch algorithm to synthesize a BT program consistent with Ψ.

Example 3. As the charging task shown in Figure 3, we assume that the LLM generates the following sketch $\mathcal{S}$: $(\square_c \fatsemi (\square_c \fatsemi \square_a) \fatsemi \square_a) \triangleright \square_a$, as shown in Figure 9(a). By modifying the labels of the internal nodes in the sketch, a set of candidate sketches can be obtained, some as shown in Figure 9(b)(c)(d). The feasibility of these sketches of SketchSpace is determined through approximation techniques, which ultimately identify $(\square_c \fatsemi (\square_c \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$ (as shown in Figure 9(d)) as the feasible sketch satisfying the specification Ψ. Then, based on the new sketch, BtBot can find a program consistent with Ψ, as shown in Figure 2.

4.5.2 *Structure Repair.* Due to the inherent ambiguity of natural language and the limited capability of LLM in understanding natural language, the sketch generated by the LLM may lack key information. For example, in the charging task shown in Figure 3, if the LLM can only extract limited information and generate the sketch: $(\square_c \fatsemi \square_a \fatsemi \square_a) \triangleright \square_a$, some details are missing. In this case, solely using the control node repair technique described in Section 4.5.1 is insufficient to obtain a feasible sketch. To address this issue, we propose a sketch structure repair technique to compensate for the limitations of LLM in understanding tasks and extracting sketches. For each infeasible sketch in SketchSpace, we repair it by expanding the sketch structure based on the failed positive examples. To improve the efficiency, we prioritize the infeasible sketches with respect to the positive examples (Line 17 in Algorithm 1), *i.e.*, the one that covers the most positive examples will be given priority.

Algorithm 3 details the algorithm for repairing the sketch's structure. The inputs are an infeasible sketch, the hole-filling suggestions produced by LLM, and the specification. If the repairing succeeds, the algorithm will output a target BT; otherwise, it returns $\perp$, *i.e.*, the repairing fails. First, the algorithm divides the positive example set into two parts: $\mathcal{E}_c^+$ that the current sketch $\mathcal{S}$ is consistent with and $\mathcal{E}_f^+$ that $\mathcal{S}$ is not. For each trace Seq_f in $\mathcal{E}_f^+$, the algorithm needs to repair the current sketch $\mathcal{S}'$ to be consist with Seq_f. The algorithm finds the closest consistent positive example Seq_c (Line 9) and gets the inconsistent part of Seq_f according to Seq_c, where $\mathsf{Similarity}(\mathsf{Seq}_1, \mathsf{Seq}_2)$ calculates the total length of the common parts in the longest prefix and suffix between two traces as follows.

$$\mathsf{Similarity}(\mathsf{Seq}_1, \mathsf{Seq}_2) := \mathsf{SimPre}(\mathsf{Seq}_1, \mathsf{Seq}_2) + \mathsf{SimSuf}(\mathsf{Seq}_1, \mathsf{Seq}_2) \tag{8}$$

SimPre and SimSuf compute the longest common prefix and suffix lengths of the two traces and are defined as follows, respectively.

$$\mathsf{SimPre}(\mathsf{Seq}_1, \mathsf{Seq}_2) := \begin{cases} 0, & \mathsf{Seq}_1 = \langle\rangle \vee \mathsf{Seq}_2 = \langle\rangle \\ 0, & \mathsf{Seq}_1 = \langle e_1\rangle \cdot \mathsf{Seq}_1' \wedge \mathsf{Seq}_2 = \langle e_2\rangle \cdot \mathsf{Seq}_2' \wedge e_1 \neq e_2 \\ 1 + \mathsf{SimPre}(\mathsf{Seq}_1', \mathsf{Seq}_2'), & \mathsf{Seq}_1 = \langle e\rangle \cdot \mathsf{Seq}_1' \wedge \mathsf{Seq}_2 = \langle e\rangle \cdot \mathsf{Seq}_2' \end{cases}$$

$$\mathsf{SimSuf}(\mathsf{Seq}_1, \mathsf{Seq}_2) := \begin{cases} 0, & \mathsf{Seq}_1 = \langle\rangle \vee \mathsf{Seq}_2 = \langle\rangle \\ 0, & \mathsf{Seq}_1 = \mathsf{Seq}_1' \cdot \langle e_1\rangle \wedge \mathsf{Seq}_2 = \mathsf{Seq}_2' \cdot \langle e_2\rangle \wedge e_1 \neq e_2 \\ 1 + \mathsf{SimSuf}(\mathsf{Seq}_1', \mathsf{Seq}_2'), & \mathsf{Seq}_1 = \mathsf{Seq}_1' \cdot \langle e\rangle \wedge \mathsf{Seq}_2 = \mathsf{Seq}_2' \cdot \langle e\rangle \end{cases}$$

Based on Seq_c, the algorithm gets the differing part between Seq_c and Seq_f (Line 10), *i.e.*, the core reason of why Seq_f cannot be covered by the sketch, where $\mathsf{Seq}_f \setminus \mathsf{Seq}_c$ removes the common prefix and suffix of Seq_f and Seq_c from Seq_f. For example, $\langle$a, b, c$\rangle \setminus \langle$a, e, c$\rangle$ is $\langle$b$\rangle$, $\langle$a, b, c$\rangle \setminus \langle$d, e, c$\rangle$ is $\langle$a, b$\rangle$, and $\langle$a, b, c$\rangle \setminus \langle$a, e, d$\rangle$ is $\langle$b, c$\rangle$.

Then, the algorithm fixes the sketch based on the core reason sequence $\langle e_1, \cdots, e_n\rangle$. First, ConstructSeq function constructs a BT S_f with root node r_f where the labels of the leaf nodes correspond

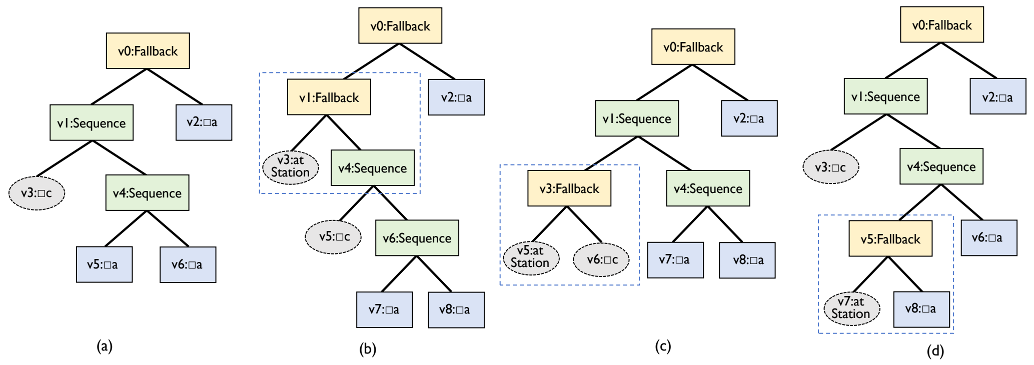


Fig. 10. (a) Initial BT $(\square_c \fatsemi \square_a \fatsemi \square_a) \triangleright \square_a$. (b)(c)(d) Perform structure repair process on (a). The structure of the Fallback node connecting the S_f root node r_f and the currently searched node is in the blue dotted box. (d) $(\square_c \fatsemi (\text{atStation} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$ satisfies the specifications Ψ.

to $\langle e_1, \cdots, e_n \rangle$ and the labels of all control nodes are ' $\fatsemi$ ' (Line 11). For example, ConstructSeq constructs a BT '$e_1 \fatsemi e_2$' when the core reason sequence is $\langle e_1, e_2 \rangle$. Then, the position of each node in the sketch is searched to determine whether the position is a valid position for S_f. For each node v in $\mathcal{S}$, the algorithm replaces the original subtree (the subtree with the current node as the root) with a new subtree formed by combining the original subtree with $\mathcal{S}_f$ (Lines 13&14), generating a new partial program $\mathcal{S}_n$. The algorithm then employs Algorithm 2 to search the target BT based on $\mathcal{S}_n$ that is consistent with the positive examples, including $\mathcal{E}_c^+$ and Seq_f (Line 15). The algorithm continues this process for each node of sketch until finding a new partial program consistent with $\mathcal{E}_c^+$ and Seq_f. Besides, the algorithm repeats the same procedure by gradually fixing all inconsistent positive examples, aiming to find a BT program consistent with the specification Ψ.

Example 4. For the automatic charging task in Figure 3, the sketch $(\square_c \fatsemi \square_a \fatsemi \square_a) \triangleright \square_a$ is shown in Figure 10(a). This sketch is infeasible, and the control node repairing cannot find the target BT. $\mathcal{S}$ is selected as the initial sketch for repair. The partial positive example set $\mathcal{E}_c^+$ that is consistent with this sketch and the only inconsistent positive example Seq_f are as follows.

$$\mathcal{E}_c^+ = \{\langle \text{batteryLow}, \text{goStation}, \text{Charge} \rangle, \langle \text{batteryLow}, \text{doOtherTask} \rangle, \langle \text{doOtherTask} \rangle\}$$

$$\text{Seq}_f = \langle \text{batteryLow}, \text{atStation}, \text{Charge} \rangle$$

So, the trace $\langle \text{batteryLow}, \text{goStation}, \text{Charge} \rangle$ is the positive example in $\mathcal{E}_c^+$ that is most similar with Seq_f. $\langle \text{atStation} \rangle$ is the core reason sequence. Since the core reason sequence contains only one element, S_f is a BT consisting of a single node, atStation. The position of each node in $\mathcal{S}$ is searched, and an attempt is made to combine S_f with subtrees of $\mathcal{S}$ using $\triangleright$, generating new sketches as shown in Figure 10. For example, locate the v1:Sequence node in Figure 10(a) and integrate S_f with the subtree rooted at this node, forming a new subtree that replaces the original one in the sketch, resulting in a new partial BT as shown in Figure 10(b) which does not meet the specifications $\mathcal{E}_c^+ \bigcup \{\text{Seq}_f\}$ and is discarded. The algorithm continues to search the other nodes (the v3:$\square_c$ node in Figure 10(a)), producing the partial BT shown in Figure 10(c). This process is repeated until the partial BT, such as the one shown in Figure 10(d), satisfies $\mathcal{E}_c^+ \bigcup \{\text{Seq}_f\}$.

$$(\square_c \fatsemi (\text{atStation} \triangleright \square_a) \fatsemi \square_a) \triangleright \square_a$$

If there are more inconsistent positive examples, the repairing process for the partial program is repeated based on these examples. In this example, there is only one inconsistent positive example, so the repair process terminates, and the partial BT can be used to find the target BT in Figure 2.

4.6 Soundness and Completeness

Theorem 4.1. **(Soundness)** *Given a specification Ψ and a natural language description of a task $\mathcal{D}$, if $\mathcal{P}$ is the complete BT program returned by Algorithm 1, then $\mathcal{P} \models \Psi$.*

Proof. According to the design of Algorithm 1, Algorithm 2, and Algorithm 3, these algorithms only return a complete BT program when the program is consistent with Ψ; otherwise, $\perp$ is returned. Hence, the soundness of BtBot is proven.

Theorem 4.2. **(Partial Completeness)** *Given a specification Ψ and a natural language description of a task $\mathcal{D}$, if there exists a BT program $\mathcal{P}$ such that $\mathcal{P} \models \Psi$, Algorithm 1 can generate a program $\mathcal{P}'$ such that $\mathcal{P}' \models \Psi$ given that the LLM can generate a sketch from which the target BT can be obtained through search and repair, and the synthesis time is adequate.*

Proof. Algorithm 2 employs a pruning technique to accelerate the search process (Line 9). The pruning technique is sound and a partial BT is pruned if and only if it is provably infeasible, *i.e.*, regardless of how its remaining holes are filled, (i) it can not generate all positive examples (over-approximation failure), or (ii) it has already produced a negative example that will persist (under-approximation failure). To prove case (i), we define $\nvDash$ to denote over-approximation's inconsistency. For a partial BT $\mathcal{S}$ such that $\mathcal{S} \nvDash \mathcal{E}^+$, *i.e.*, $\exists \text{Seq} \in \mathcal{E}^+ \bullet \mathcal{S} \nvDash \text{Seq} \vee \mathcal{S}^\Pi[\text{Seq}] \nvDash \mathcal{E}^+ \setminus \{\text{Seq}\}$, which means $(\forall s \in [\![\mathcal{S}]\!] \bullet s \nvDash \text{Seq}) \vee \mathcal{S}^\Pi[\text{Seq}] \nvDash \mathcal{E}^+ \setminus \{\text{Seq}\}$, if $\forall s \in [\![\mathcal{S}]\!] \bullet s \nvDash \text{Seq}$ is the case, according to the Basic, Action-Type, and Condition-Type rules in Figure 8, $\forall s \in [\![\mathcal{S}]\!] \bullet s \nvDash \text{Seq}$ will continue to hold regardless of how the remaining holes in $\mathcal{S}$ are filled. For the later case $\mathcal{S}^\Pi[\text{Seq}] \nvDash \mathcal{E}^+ \setminus \{\text{Seq}\}$, the infeasibility can be proven inductively. Therefore, no completion of $\mathcal{S}$ can lead to the target BT. To prove case (ii), according to Lemma 4.1, if a partial BT generates a negative example Seq, then all completions of $\mathcal{S}$ generate Seq. Therefore, $\mathcal{S}$ is infeasible and can be safely pruned. Moreover, partial BTs that do not exhibit either of the above two failures will not be pruned. Therefore, the pruning procedure is sound, ensuring that every pruned partial BT is infeasible, and no feasible partial BT is mistakenly pruned.

For partial BTs that are not pruned, Lines 10 and 11 of Algorithm 2 further explore the corresponding search space by dividing it into multiple subspaces. Each subspace is independently pruned and searched. Although BtBot is based on heuristic search (Line 5 of Algorithm 2), the unexplored partial BTs are not pruned. Algorithm 2 guarantees that all unpruned BTs will eventually be explored (Line 11) until the target BT is found. However, as indicated by the design of Algorithm 1, Algorithm 2, and Algorithm 3, BtBot relies on the ability of LLM to generate sketches (Line 5 of Algorithm 1) and requires sufficient search time to explore the search space. Therefore, the completeness of BtBot is partial.

5 Evaluation

This section presents the evaluation of BtBot, demonstrating its effectiveness in generating BTs. We compare BtBot with four BT synthesis techniques and conduct ablation studies to evaluate the effectiveness of the key components proposed in BtBot. Our experiments aim to answer the following research questions:

- **(RQ1)** How does BtBot perform in generating BTs that solve the tasks?
- **(RQ2)** How does BtBot perform compared with existing BT synthesis methods?
- **(RQ3)** How important are the different components of BtBot in solving the tasks?
- **(RQ4)** How helpful is BtBot for users?

***Benchmark*.** To evaluate the effectiveness of BtBot, we collected 70 different tasks from the existing literature [Hong et al. 2023; Izzo et al. 2024]. The benchmark includes four scenarios: robot movement, object transmission, manipulation, and priority decision-making. Typically, as

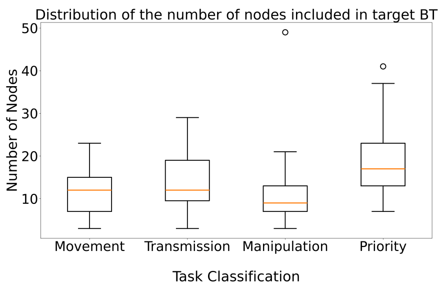


Fig. 11. Complexity distribution of the benchmark. Measure complexity based on the number of nodes in the BT.

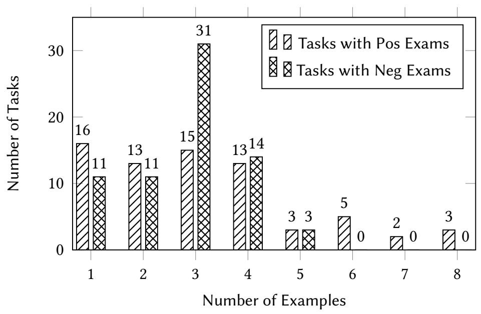


Fig. 12. The distribution chart of the number of positive and negative examples contained in the tasks and the corresponding number of tasks.

the number of BT's nodes increases, the complexity of the task also increases. Figure 11 shows the distribution of the total number of nodes in the *BTs constructed by human experts* for each task. As indicated by the figure, the complexities of these tasks are well distributed, and the benchmark can provide a comprehensive evaluation of the effectiveness and applicability of BtBot.

We designed task descriptions for all tasks that adhere to the language patterns introduced in Section 4.3.2, along with adequate positive and negative examples based on the natural language descriptions to verify the consistency between the generated BTs and the descriptions as shown in Figure 12. The time required to provide input to BtBot is closely associated with the complexity of the task. We measured the time spent on two parts: *understanding the task and designing the natural language description along with positive and negative examples*. For simpler tasks involving 3 to 15 nodes, this process typically took 1 to 5 minutes. For more complex tasks involving 29 to 49 nodes, it could take 5 to 15 minutes. Additionally, these task descriptions and sketch extraction tasks are entirely new information and instructions for a general LLM. Therefore, BtBot can be effectively extended to tasks in other scenarios.

Satisfying the specifications of positive and negative examples in the benchmark is the *necessary condition* for meeting task requirements. Therefore, the specifications are used to determine the correctness of BTs. This is a weaker way to verify the correctness of the BT, but it can achieve the desired goal when controlling the robot to perform specified behaviors.

Experimental Setup. All experiments have been conducted on a server equipped with 16 cores at 2.5 GHz and 64 GB of physical memory, running the Ubuntu 22.04 operating system. We employ ChatGPT [Islam and Moushi 2024] via the OpenAI API [Paredes et al. 2023] as the sketch generator. For each task, we query the LLM for a maximum of 10 iterations. To ensure fairness, the timeout for all baselines is set to 3600 seconds for each task. However, BtBot and its variants complete most tasks (either successfully solving them or reaching the maximum number of LLM calls) within 30 seconds.

5.1 Results of RQ1

To demonstrate the effectiveness of BtBot, we compare BtBot with BTLLMGen (which directly generates BT programs using LLMs) regarding their performance on the benchmark tasks. Besides, to evaluate the impact of LLMs on BT generation, we use three different LLMs to evaluate the influence on both BtBot and BTLLMGen. Table 1 shows the results. BTLLMGen yields a relatively low success rate, whereas BtBot achieves a success rate of up to 96.3% . Furthermore, the impact of different LLMs on BtBot is minimal, with the success rate differing by only 16.0%. In contrast, the performance of BTLLMGen differs by 26.3% between ChatGPT-3.5 and ChatGPT-4o. Therefore, compared with BTLLMGen, BtBot is more effective and robust in solving tasks.

In the experiments, for most tasks, a single call to an LLM is sufficient to solve the task. For more complex tasks (containing 29 to 39 nodes), 2 to 5 calls are required to generate different sketches to solve the tasks. A small number of very complex tasks (containing more than 39 nodes) may require 4 to 10 calls to solve, or may even result in task failure.

Table 1. Comparison between BtBot and BTLLMGen which refers to the direct generation of BT programs using only LLMs.

LLMs	**BtBot**		**BTLLMGen**	
	Success Number	Success Rate	Success Number	Success Rate
GPT 3.5	56.2	80.3%	15	21.4%
GPT 4	65	92.9%	31.6	45.1%
GPT 4o	67.4	96.3%	33.4	47.7%

Besides, an analysis of the results in Table 1 reveals that, although the LLMs exhibit varying levels of task comprehension and sketch generation capabilities, BtBot effectively addresses the problems in the generated sketches, compensating for the limitations of the LLMs. Furthermore, as the LLMs' ability to understand natural language descriptions improves, BtBot can leverage the enhanced capabilities of the latest LLMs, thereby improving its performance.

Answer to RQ1. BtBot can synthesize a BT consistent with the specifications for 96.3% of the benchmarks. Furthermore, BtBot can leverage the ability of the latest LLMs to understand tasks and generate sketches, enhancing its task-solving capability.

5.2 Results of RQ2

We compare BtBot with the following four BT synthesis techniques to answer the second research question, where ChatGPT-4o serves as the sketch generator for BtBot.

- **BTGenBot**: Recent works have used fine-tuned LLMs to generate BTs [Izzo et al. 2024; Li et al. 2024]. We use an open-source project called BTGenBot [Izzo et al. 2024] as our baseline, which employs a fine-tuned large language model for generating the BTs according to the natural language task descriptions.
- **MCTS-Syn**: We use the existing work [Hong et al. 2023] that combines Monte Carlo Tree Search (MCTS) [Browne et al. 2012; Chaslot 2010] with Communicating Sequential Processes (CSP) [Roscoe 1994; Schneider 1999] based formal verification for BT synthesis as our baseline. This approach generates BTs that satisfy the given Linear Temporal Logic (LTL) [Emerson 1990; Pnueli 1977] specifications. The timeout of MCTS-Syn is set to 3600 seconds.
- **BTLLMGen-GPT4o**: We use ChatGPT-4o [Islam and Moushi 2024] as the BT generator to directly synthesize BTs from task descriptions. We determine the correctness of the generated BT by checking if it satisfies the given positive and negative example specifications. If the BT does not meet the specifications, the LLM is invoked again to generate a different BT. This process is repeated to 10 times, with a timeout of 3600 seconds.
- **BTLLMExam-GPT4o**: We use ChatGPT-4o [Islam and Moushi 2024] as the BT generator to directly synthesize a BT based on task descriptions, positive and negative examples, and failure feedback. The correctness of the generated BT is evaluated by checking whether it satisfies the given positive and negative example specifications. If the BT does not meet the specifications, the failed examples and the previous solution are fed back to the LLM as guidance for generating a new BT. This process is repeated up to 10 times, with a timeout of 3600 seconds.

Since 73% of the tasks in BtBot's benchmark are derived from the fine-tuning dataset of BTGenBot, evaluating BTGenBot on these tasks may lead to unfair evaluation. Furthermore, due to the setup for fine-tuning the LLM, we are unable to reproduce BTGenBot locally. For these two reasons, we compare the performance of BtBot and BTGenBot by solving the tasks in BTGenBot's test set. In this evaluation, BtBot is provided with appropriate task descriptions and positive and negative examples. The task is considered successfully solved by BtBot if the generated BT is consistent with the task description of BTGenBot test tasks.

Table 2. Comparison between BtBot and BTGenBot.

test task	BtBot	LlamaChat			CodeLlama		
		ZS	OS	OS+SA	ZS	OS	OS+SA
Task 1	✓		✓	✓		✓	✓
Task 2	✓		✓	✓			
Task 3	✓			✓		✓	✓
Task 4	✓		✓	✓		✓	✓
Task 5	✓		✓	✓			
Task 6	✓			✓			
Task 7	✓						
Task 8	✓			✓			
Task 9	✓						

We directly use the experimental results from BTGenBot [Izzo et al. 2024] (shown in Table V) for comparison with BtBot. Analyzing whether the BTs generated by both methods meet the task requirements, Table 2 shows the results, where ZS, OS, and OS+SA respectively represent the modes of zero-shot (ZS), one-shot prompts (OS), and the combination of one-shot prompting and the correction method in BTGenBot. As the table indicates, BtBot successfully solves the nine tasks. However, BTGenBot can only solve a maximum of seven tasks, indicating that BtBot can solve more tasks than the fine-tuned LLM technique. Furthermore, since BTGenBot relies on fine-tuned LLM techniques, its generalizability to other scenarios remains to be explored.

Since no prior work guides BT synthesis using positive and negative examples, we compare BtBot with MCTS-Syn [Hong et al. 2023], a search-based BT synthesis approach, as well as BTLLMGen-GPT4o and BTLLMExam-GPT4o, two LLM-based BT generation methods. The LTL formulas are provided for MCTS-Syn based on the task's natural language descriptions and appropriate prompts are designed for BTLLMGen-GPT4o and BTLLMExam-GPT4o. MCTS-Syn generates BTs that satisfy LTL specifications, where LTL is a formalism used to describe the task. All negative examples of the tasks in BtBot violate the corresponding LTL formulas, while all positive examples do not. Although the two techniques enforce different specifications for the generated BTs, the resulting BTs are equivalent and meet the task requirements.

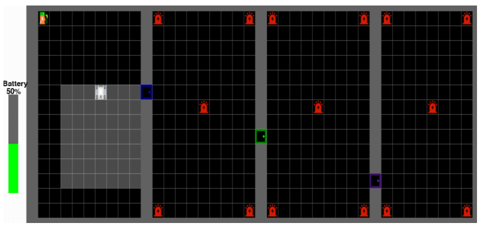


Fig. 13. An environment example where a BT is used to control autonomous charging and patrolling tasks, with the experimental environment built using Mini-Grid 3.0.

As shown in the environment depicted in Figure 13, one of the tasks in the benchmark is: *When the robot is in a low battery state, it should check whether it is at the charging station or move to the charging station and finally start charging; otherwise, it should sequentially navigate to the room A, B and C and patrol in each.* The specifications of the task for BtBot are shown in Figure 3, where 'doOtherTask' represents the task ⟨goA, Patrol, goB, Patrol, goC, Patrol⟩. BtBot can generate the desired BT within 5s with only a single LLM call as follows. By comparison, while BTLLMExam-GPT4o requires 2–5 calls, and both MCTS-Syn and BTLLMGen-GPT4o fail due to timeout or exceeding LLM call limits.

$$(\mathsf{batteryLow} \fatsemi (\mathsf{atStation} \triangleright \mathsf{goStation}) \fatsemi \mathsf{Charge}) \triangleright ((\mathsf{goA} \fatsemi \mathsf{Patrol}) \fatsemi ((\mathsf{goB} \fatsemi \mathsf{Patrol}) \fatsemi (\mathsf{goC} \fatsemi \mathsf{Patrol})))$$

This BT consists of 19 nodes and has a 5-layer structure, making it a relatively complex task [Iovino and Smith 2023; Li et al. 2024]. BtBot can generate the desired BT program for tasks with 49 nodes, demonstrating good scalability on complex tasks.

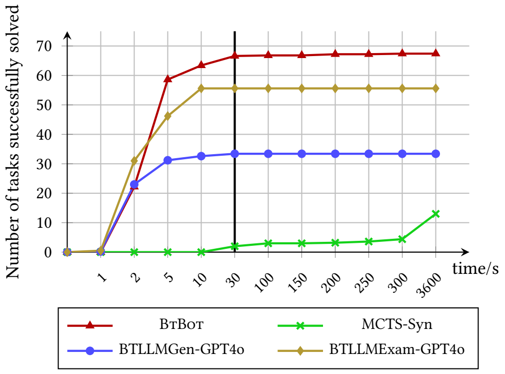


Fig. 14. The number of successfully solved benchmark tasks is shown when the time limit is varied.

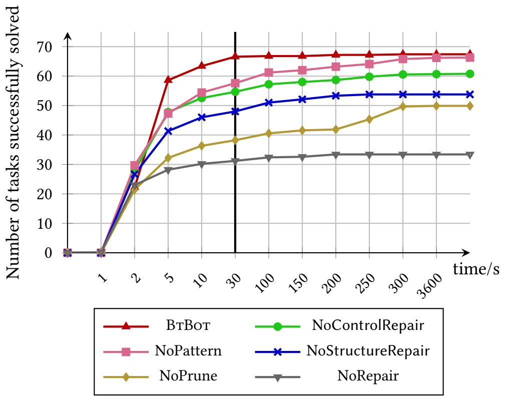


Fig. 15. Ablation Study of BtBot.

As shown in Figure 14, the number of benchmark tasks solved when we vary the time limit is displayed. The results are averaged over five repeated experiments. The results indicate that BtBot can solve most tasks within a short time (1 to 5 seconds), more complex tasks can be solved within 30 seconds (as indicated by the bold line at 30 seconds) or fail due to reaching the LLM's maximum iteration limit. Compared to BtBot, all three methods demonstrate lower success rates: MCTS-Syn incurs higher time costs with limited success, BTLLMGen-GPT4o solves only 47.7% of benchmark tasks, and BTLLMExam-GPT4o achieves at most 79.4%. Besides, MCTS-Syn requires LTL specification for each task, which presents a challenge for non-expert users. Comparing the BTs generated by the four approaches, BtBot, BTLLMGen, and BTLLMExam generally produce more interpretable BTs, as they leverage LLMs to understand the task descriptions. In contrast, MCTS-Syn relies on LTL-guided search and it encodes the preconditions of actions within the actions themselves, which reduces the number of nodes in the generated BTs, but at the cost of lower readability. For example, for an *alarm task* (introduction omitted), MCTS-Syn generates a BT with 9 nodes and 5 layers, while BtBot generates one with 13 nodes and the same depth.

Answer to RQ2. We evaluated the performance of BtBot by comparing it against four different kinds of baseline BT synthesis approaches. The experimental results demonstrate that BtBot outperforms the state-of-the-art methods in terms of both success rate and efficiency.

5.3 Results of RQ3

To evaluate BtBot further, we conducted an ablation study of BtBot's key components. We consider the following variants of BtBot.

- **NoPattern**. A variant of BtBot without using the language pattern designed in Section 4.3.2 when prompting the LLM to generate a sketch.
- **NoPrune**. A variant of BtBot without the pruning technique in Section 4.4, *i.e.*, it does not use abstraction-based pruning for partial programs. Instead, it only checks whether the complete BT programs are consistent with the specification.
- **NoControlRepair**. A variant of BtBot without employing the control node repair process (*c.f.*, Section 4.5.1) when the LLM-generated sketch is infeasible. Instead, it directly applies the algorithm RepairSketch (*c.f.*, Section 4.5.2) for repairing.
- **NoStructureRepair**. A variant of BtBot without employing the structure repair process (*c.f.*, Section 4.5.2) to repair infeasible sketch and only searches within SketchSpace($\mathcal{S}$).
- **NoRepair**. A variant of BtBot without performing sketch repair operations (*c.f.*, Section 4.5.1 and Section 4.5.2). It just repeatedly invokes the LLM to generate new sketches.

Following the experimental setup in Figure 14 of Section 5.2, the results are averaged over five repeated experiments. All variants utilize ChatGPT-4o as the sketch generator. Figure 15 illustrates the number of tasks completed within the specified time. As Figure 15 demonstrates, BtBot solves more tasks than the other variants. The results of the variants NoControlRepair, NoStructureRepair, and NoRepair indicate that both repair stages are important in solving tasks successfully. The NoPrune variant demonstrates that pruning is essential for improving synthesis efficiency. The NoPattern indicates that repairing an infeasible sketch requires more time, whereas the presence of language patterns improves the accuracy of sketch generation. In summary, this ablation study demonstrates that all the components in BtBot play a key role in accelerating synthesis and improving the success rate.

Answer to RQ3. The experimental results demonstrate that the BtBot's four key components can enhance the efficiency and success rate of BT program synthesis. The pruning technique and language patterns improve the synthesis efficiency, while the multi-step sketch repairing technique improves the success rate.

5.4 Results of RQ4

To evaluate the effectiveness of BtBot in assisting users in constructing BTs, we conducted two user studies. The first study (the Single-User Study) examined the effect of using BtBot once, while the second study (the Multi-User Study) explored the impact of using BtBot iteratively with feedback from failed attempts. Each study recruited 12 participants. The 12 participants in the Single-User Study included 4 computer science graduate students (experts) who regularly design and use BTs, and 8 computer science graduate students (non-experts) unfamiliar with BTs. The Multi-User Study also involved 12 participants, consisting of 5 computer science graduate students who regularly design and use BTs (1 of whom did not participate in the Single-User Study), and 7 computer science graduate students unfamiliar with BTs (5 of whom had not participated in the Single-User Study). The two studies were conducted more than three months apart, and we consider it unlikely that the earlier Single-User Study influenced the results of the Multi-User Study.

Before the experiment, participants who were unfamiliar with BTs received a training session covering the basic concepts of BTs. All participants were introduced to the task requirements and tool usage. Each participant was asked to randomly select ten tasks from the benchmark and divide them into two groups of comparable difficulty based on their understanding of the tasks. *We assumed that the outcomes for individual tasks were independent and that performance on one group would not influence performance on the other. Participants were only provided with a basic description of each task, without access to the natural language descriptions or examples from the benchmark.* Each participant randomly selected one group of tasks to solve by manually constructing BTs. For the other group, they were asked to provide their own natural language descriptions and positive and negative examples based on their understanding of the tasks and used them as input to BtBot to synthesize the BTs. The examples in the benchmark were used to validate the BTs constructed (manually or aided by BtBot) by users. In the Single-User Study, participants were informed that they had only one chance and were instructed to do their best to complete the tasks. In the Multi-User Study, participants were informed that they could iteratively solve the tasks up to four times. They were allowed to revise either the manually constructed BTs or the descriptions and examples based on feedback from failed attempts.

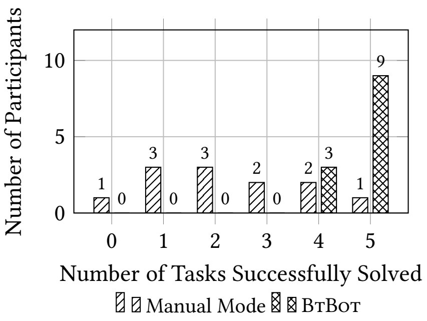
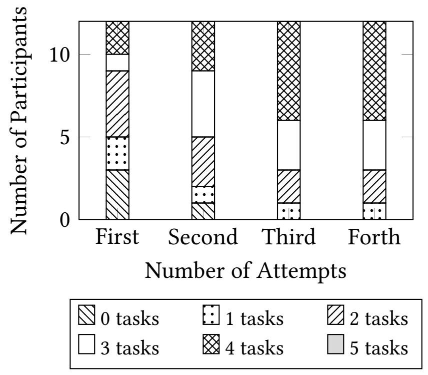
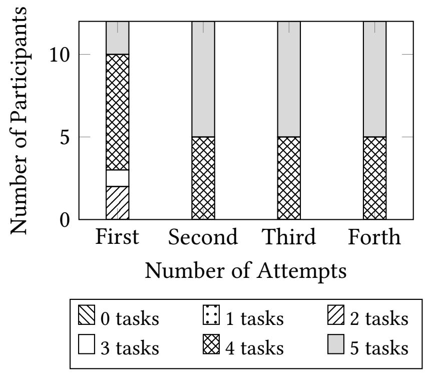


Fig. 16. Comparison of BtBot and Manual BT Construction in the Single-User Study.



(a) Results of iterative manual BT construction based on feedback.



(b) Results of iterative BT construction aided by BtBot based on feedback.

Fig. 17. Comparison of BtBot and Manual BT Construction in the Multi-User Study.

We counted the number of tasks successfully solved by the participants, and the results of the Single-User Study are shown in Figure 16. The X-axis displays the number of tasks successfully solved, and the Y-axis displays the number of participants who solved the *n* tasks successfully. The results demonstrate that BtBot performs well in a single attempt, with all participants using BtBot solving four or more tasks. In contrast, only 3 participants were able to manually construct four or more correct BTs, and one participant failed to solve any task successfully. The results of the Multi-User Study are shown in Figure 17. Figure 17(a) shows the distribution of participants who solved *n* tasks by manually constructing and iteratively modifying BTs, while Figure 17(b) shows the distribution of participants who solved *n* tasks using BtBot by iteratively modifying the task descriptions and examples. The results indicate that incorporating failure feedback improves the success rate of BT construction using both manual methods and BtBot. However, after up to four attempts, only 6 participants were able to manually solve at least four tasks (*a higher success rate* than the results in Figure 16). In contrast, all participants successfully completed four or more tasks using BtBot. The failed tasks were generally complex, with challenging task descriptions and examples, with a limited number of examples provided.

The time spent by users to provide the necessary input for BtBot is 0.8 to 1.5 times the time required to manually construct the BT, with an average of 1.25 times. In the Single-User Study, among the participants, two reported that BtBot required excessive input, which they found cumbersome, and felt that manually writing BT programs was faster and more intuitive. However, the two participants completed 2 out of 5 tasks and 0 task manually, respectively. Although providing the necessary information for BtBot takes longer, 7/12 participants found that providing this information for BtBot was less challenging and user-friendly. Ten participants indicated that providing a description of the task along with key positive and negative examples is easier and more effective in obtaining the desired BT when solving complex tasks. Since providing all the positive and negative examples is a difficult task, participants preferred to provide a subset of critical examples, which was also sufficient to generate a BT that meets the task requirements. In the Multi-User Study, all participants generally agreed that while failure feedback was helpful for revising BTs manually, the process remained challenging, particularly due to syntax errors which were often difficult to locate and correct. In contrast, all participants found BtBot effective in constructing correct BTs and required less time to revise its input information. They were not required to ensure that the BTs conformed to syntactic rules; instead, they were only expected to focus on accurately describing the tasks. With the assistance of feedback on failure, they were able to effectively revise the input information, and all participants solved at least four tasks by the second attempt. Although a few challenging tasks remained unsolved after four iterations, the participants reported satisfaction with the results. These findings suggest that BtBot can effectively assist both expert and non-expert users in efficiently constructing correct BT programs.

Answer to RQ4. The results of the user studies demonstrate that BtBot helps users design more desirable BT programs. Specifically, 83% of participants reported that using BtBot was more convenient and efficient than manually constructing a BT in a single attempt. Moreover, all participants agreed that refining the task description and examples based on feedback to generate a BT using BtBot was simpler and more effective than manual construction.

6 Related Work

Generating BT Programs from Natural Language. Many techniques employ LLMs to generate BT programs from natural language task descriptions [Li et al. 2024; Vemprala et al. 2024]. However, since LLMs has not been specifically trained in the BT domain, the generated BTs tend to be of lower quality. Current techniques typically fine-tune the parameters of lightweight LLMs to enhance their understanding of BT tasks and improve their ability to generate BTs [Izzo et al. 2024]. However, fine-tuning LLMs requires significant relevant data and computational resources. In contrast, our approach also leverages an LLM to understand the task's natural language description but does not require fine-tuning the LLM. Instead, it provides a few simple examples to guide the LLM in understanding the task. The LLM, acting as a neural sketch generator, provides an initial sketch for the subsequent search process.

Learning Control Strategies from Examples or Demonstrations. Some existing works use imitation learning [Hussein et al. 2017] to generate control strategies consistent with given demonstrations [Hua et al. 2021; Stepputtis et al. 2020; Verma et al. 2018]. However, this process typically requires a large number of examples. Some approaches, such as techniques for learning strategies from demonstrations [Patton et al. 2023], extract regular expressions from a few demonstrations and then convert these expressions into strategy programs. Our work is the first to propose using positive and negative examples to guide sketch-based BT synthesis techniques. We design a multi-step sketch repair process based on the idea of Counterexample-Guided Inductive Synthesis (CEGIS) [Solar-Lezama et al. 2005] to search for the target BT satisfying given specifications.

Program synthesis based on sketches. Sketch-based program synthesis [Solar-Lezama 2009] is a classic program synthesis technique, but how to obtain the feasible sketch has been a longstanding research challenge. Recently, Chen *et al.* [Chen et al. 2023] has leveraged LLMs to extract sketches from examples and diagnosed the reasons for infeasibility, guiding the LLM to generate more accurate sketches. Similarly, Yuviler and Drachsler [Yuviler and Drachsler-Cohen 2023] employed LLMs to learn user demonstrations and generate sketches, thereby synthesizing the programs that address user preference problems. On the other hand, our method leverages LLM to directly generate sketches from natural language descriptions and designs repair algorithms to fix infeasible sketches, rather than providing LLM with information about the errors and asking it to generate a more suitable sketch. This approach helps reduce the overall cost to some extent.

Program Synthesis Using Programming by Examples (PBE). PBE is a classic technique that uses positive and negative examples to guide the synthesis process. However, PBE techniques often require searching through all possible candidate programs, resulting in a large search space. Smore [Chen et al. 2023] utilizes an LLM first to extract information from the examples, generate a sketch, and then decompose the example to complete each hole in the sketch. Regel [Chen et al. 2020] captures crucial hints from natural language descriptions and parses them into a hierarchical sketch. The PBE engine utilizes this information to prioritize the search and employs inference to prune the search space. BtBot leverages an LLM to understand natural language descriptions, extract crucial hints, and generate a sketch. Positive and negative examples and abstraction techniques for partial programs are used to prune the search space while filling the sketch.

Generating BT Programs with LLMs. Using an LLM to generate BTs presents a challenge, as the LLM is not specifically trained for the BT domain. Therefore, the correctness of the generated BTs cannot be guaranteed. Many existing works address this issue by fine-tuning LLMs to improve their performance in BT generation [Li et al. 2024]. However, this requires large amounts of training data and significant computational resources. Alternatively, some works leverage the LLM's understanding and reasoning capabilities to provide input for existing BT generation techniques [Chen et al. 2024; Zhou et al. 2024]. BtBot leverages the LLM's ability to understand and generalize natural language, generating sketches that are used for synthesis.

Learning BT from demonstrations. Past works on learning BTs from demonstrations (only positive examples) typically learn strategies from given demonstrations. Past work [Robertson and Watson 2015] proposed representing all examples as single-layer BTs expressed by sequences, then iteratively merging the behavior trees to obtain the final BT. French *et al.* [French et al. 2019] introduced learning decision trees from demonstrations and then converting them into BTs. Recent work [Gustavsson et al. 2022] extracted the sequence of actions from demonstrations and used planning-based learning for BT construction. Our work, however, uses positive and negative examples to guide BT learning. BtBot not only explicitly specifies the behaviors (positive examples) that must be satisfied but also clearly defines the behaviors (negative examples) that are prohibited, which effectively improves the correctness of the generated BTs.

7 Conclusion and Future Work

We propose BtBot, a novel multi-modal technique for automatically synthesizing BT programs from natural language descriptions. BtBot is based on two key insights: first, leveraging LLMs to generate task execution sketches from natural language descriptions; second, applying abstraction techniques for partial BT programs to prune the search space, thereby accelerating the synthesis process. Besides, since the sketches generated by LLMs may be infeasible, BtBot can repair the sketches based on positive examples. We conducted a comprehensive evaluation of BtBot, and the results show that it outperforms existing BT synthesis baselines, achieving a task success rate of up to 96.3%. To assess robustness, we tested BtBot with three different LLMs, observing a success rate variation of only 16.0%. We also designed ablation experiments to verify the necessity of each component of BtBot. Finally, two user studies demonstrate the effectiveness of BtBot in assisting both experts and non-experts in constructing BTs.

Requiring a lot of input information (task descriptions and positive/negative examples) is a primary limitation of BtBot. Its reliance on positive and negative examples is also a limitation, but this is an inherent limitation of PBE techniques. To address these limitations and further refine our work, future work will focus on two aspects: 1) Reducing the requirement for excessive input information by exploring more efficient BT synthesis techniques. 2) Investigating synthesis techniques for more advanced BTs, such as those incorporating parallel and decorator nodes.

Data Availability Statement

The artifact associated with this paper is available at Zenodo [Zhang et al. 2025].

Acknowledgments

We thank the anonymous reviewers for the valuable comments. This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429).

References

David Ångström. 2022. Genetic Algorithms for optimizing behavior trees in air combat.

Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Isil Dillig. 2023. ImageEye: Batch Image Processing using Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 686–711. doi:10.1145/3591248

Oliver Biggar and Mohammad Zamani. 2020. A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic. *RA-L 2020* 5, 2 (2020), 2341–2348. doi:10.1109/LRA.2020.2970634

Oliver Biggar, Mohammad Zamani, and Iman Shames. 2021. An Expressiveness Hierarchy of Behavior Trees and Related Architectures. *IEEE Robotics Autom. Lett.* 6, 3 (2021), 5397–5404. doi:10.1109/LRA.2021.3074337

Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games* 4, 1 (2012), 1–43. doi:10.1109/TCIAIG.2012.2186810

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. AAAI Press, 6058–6065. doi:10.1609/AAAI.V35I7.16755

Guillaume Maurice Jean-Bernard Chaslot Chaslot. 2010. Monte-carlo tree search. (2010).

Qiaochu Chen, Arko Banerjee, Çagatay Demiralp, Greg Durrett, and Isil Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. 7, OOPSLA2 (2023), 1848–1877. doi:10.1145/3622863

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *PLDI 2020*. ACM, 487–502. doi:10.1145/3385412.3385988

Xinglin Chen, Yishuai Cai, Yunxin Mao, Minglong Li, Wenjing Yang, Weixia Xu, and Ji Wang. 2024. Integrating Intent Understanding and Optimal Behavior Planning for Behavior Tree Generation from Human Instructions. *IJCAI 2024*.

Michele Colledanchise and Petter Ögren. 2018. *Behavior trees in robotics and AI: An introduction*. CRC Press.

Maaike de Boer, Quirine Smit, Michael van Bekkum, André Meyer-Vitali, and Thomas Schmid. 2024. Design Patterns for LLM-based Neuro-Symbolic Systems. (2024).

E. Allen Emerson. 1990. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen (Ed.). 995–1072. doi:10.1016/B978-0-444-88074-1.50021-4

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *PLDI 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 420–435. doi:10.1145/3192366.3192382

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI 2015*. ACM, 229–239. doi:10.1145/2737924.2737977

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*. 7791–7797. doi:10.1109/ICRA.2019.8794104

Yanchang Fu, Long Qin, and Quanjun Yin. 2016. A reinforcement learning behavior tree framework for game AI. In *SSEME 2016*. Atlantis Press, 573–579.

Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2022. PPT: Pre-trained Prompt Tuning for Few-shot Learning. In *ACL 2022*. 8410–8423. doi:10.18653/V1/2022.ACL-LONG.576

Sumit Gulwani and Prateek Jain. 2017. Programming by examples: PL meets ML. In *APLAS 2017*. 3–20. doi:10.1007/978-3-319-71237-6_1

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. doi:10.1561/2500000010

Oscar Gustavsson, Matteo Iovino, Jonathan Styrud, and Christian Smith. 2022. Combining context awareness and planning to learn behavior trees from demonstration. In *RO-MAN 2022*. 1153–1160. doi:10.1109/RO-MAN53752.2022.9900603

David Harel, Dexter Kozen, and Jerzy Tiuryn. 2001. Dynamic logic. *SIGACT News* (2001), 66–69. doi:10.1145/568438.568456

Weijiang Hong, Zhenbang Chen, Minglong Li, Yuhan Li, Peishan Huang, and Ji Wang. 2023. Formal Verification Based Synthesis for Behavior Trees. In *SETTA 2023*. 72–91. doi:10.1007/978-981-99-8664-4_5

Jiang Hua, Liangcai Zeng, Gongfa Li, and Zhaojie Ju. 2021. Learning for a robot: Deep reinforcement learning, imitation learning, transfer learning. *Sensors* (2021), 1278. doi:10.3390/S21041278

Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. 2017. Imitation learning: A survey of learning methods. *CSUR 2017* (2017), 1–35. doi:10.1145/3054912

Matteo Iovino and Christian Smith. 2023. Behavior Trees for Robust Task Level Control in Robotic Applications. *CoRR* (2023). doi:10.48550/ARXIV.2301.06434

Raisa Islam and Owana Marzia Moushi. 2024. Gpt-4o: The cutting-edge advancement in multimodal llm. *Authorea Preprints* (2024).

Riccardo Andrea Izzo, Gianluca Bardaro, and Matteo Matteucci. 2024. BTGenBot: Behavior Tree Generation for Robotic Tasks with Lightweight LLMs. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024*. 9684–9690. doi:10.1109/IROS58592.2024.10802304

Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. 2025. Automated Repair of Ambiguous Natural Language Requirements. *CoRR 2025* (2025). doi:10.48550/ARXIV.2505.07270

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In *The Eleventh International Conference on Learning Representations, ICLR 2023*. https://openreview.net/forum?id=_nGgzQjzaRy

Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *FSE 2017*. 593–604. doi:10.1145/3106237.3106309

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* (2024). doi:10.48550/ARXIV.2401.08089

Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *ICSE 2024*. 1–13. doi:10.1145/3597503.3639180

Feipeng Ma, Yizhou Zhou, Yueyi Zhang, Siying Wu, Zheyu Zhang, Zilong He, Fengyun Rao, and Xiaoyan Sun. 2024. Task Navigator: Decomposing Complex Tasks for Multimodal Large Language Models. In *CVPR 2024*. 2248–2257. doi:10.1109/CVPRW63382.2024.00230

Ryan Marcotte and Howard J Hamilton. 2017. Behavior trees for modelling artificial intelligence in games: A tutorial. *The Computer Games Journal* 6 (2017), 171–184. doi:10.1007/S40869-017-0040-9

Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ögren. 2014. Towards a unified behavior trees framework for robot control. In *ICRA 2014*. 5420–5427. doi:10.1109/ICRA.2014.6907656

Alberto Muñoz-Ortiz, Carlos Gómez-Rodríguez, and David Vilares. 2024. Contrasting Linguistic Patterns in Human and LLM-Generated News Text. *Artif. Intell. Rev.* (2024), 265. doi:10.1007/S10462-024-10903-2

Petter Ögren and Christopher I Sprague. 2022. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems* (2022), 81–107. doi:10.1146/ANNUREV-CONTROL-042920-095314

Magnus Olsson. 2016. Behavior trees for decision-making in autonomous driving.

Cristian Mauricio Gallardo Paredes, Cristian Machuca, and Yadira Maricela Semblantes Claudio. 2023. ChatGPT API: Brief overview and integration in Software Development. *International Journal of Engineering Insights* 1, 1 (2023), 25–29.

Noah Patton, Kia Rahmani, Meghana Missula, Joydeep Biswas, and Isil Dillig. 2023. Program Synthesis for Robot Learning from Demonstrations. *CoRR* (2023). doi:10.48550/ARXIV.2305.03129

Chris Paxton, Nathan D. Ratliff, Clemens Eppner, and Dieter Fox. 2019. Representing Robot Task Plans as Robust Logical-Dynamical Systems. In *IROS2019*. IEEE, 5588–5595. doi:10.1109/IROS40897.2019.8967861

Amir Pnueli. 1977. The temporal logic of programs. In *sfcs 1977*. 46–57. doi:10.1109/SFCS.1977.32

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis. *OOPSLA 2021* (2021), 1–29. doi:10.1145/3485535

Glen Robertson and Ian Watson. 2015. Building behavior trees from observations in real-time strategy games. In *INISTA 2015*. 1–7. doi:10.1109/INISTA.2015.7276774

Bill Roscoe. 1994. Model- checking CSP. (1994).

Emily Scheide, Graeme Best, and Geoffrey A Hollinger. 2021. Behavior tree learning for robotic task planning through Monte Carlo DAG search over a formal grammar. In *ICRA 2021*. IEEE, 4837–4843. doi:10.1109/ICRA48506.2021.9561027

Steve Schneider. 1999. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons.

Yoones A Sekhavat. 2017. Behavior trees for computer games. *IJAIT 2017* (2017), 1730001. doi:10.1142/S0218213017300010

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *APLAS 2009*. Springer, 4–13. doi:10.1007/978-3-642-10672-9_3

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 281–294. doi:10.1145/1065010.1065045

Simon Stepputtis, Joseph Campbell, Mariano Phielipp, Stefan Lee, Chitta Baral, and Heni Ben Amor. 2020. Language-conditioned imitation learning for robot manipulation tasks. *NeurIPS 2020* 33 (2020), 13139–13150. https://proceedings.neurips.cc/paper/2020/hash/9909794d52985cbc5d95c26e31125d1a-Abstract.html

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. 2024. ChatGPT for Robotics: Design Principles and Model Abilities. *IEEE Access* 12 (2024), 55682–55696. doi:10.1109/ACCESS.2024.3387941

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically interpretable reinforcement learning. In *ICML 2018*. PMLR, 5045–5054. http://proceedings.mlr.press/v80/verma18a.html

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* POPL (2017), 1–30. doi:10.1145/3158151

Roel M Willems, Stefan L Frank, Annabel D Nijhof, Peter Hagoort, and Antal Van den Bosch. 2016. Prediction during natural language comprehension. *Cerebral cortex* 26, 6 (2016), 2506–2516.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 63:1–63:26. doi:10.1145/3133887

Quan Yuan, Mehran Kazemi, Xin Xu, Isaac Noble, Vaiva Imbrasaite, and Deepak Ramachandran. 2024. Tasklama: probing the complex task understanding of language models. In *AAAI 2024*. 19468–19476. doi:10.1609/AAAI.V38I17.29918

Tom Yuviler and Dana Drachsler-Cohen. 2023. One Pixel Adversarial Attacks via Sketched Programs. *PLDI 2023* 7, PLDI (2023), 1970–1994. doi:10.1145/3591301

Wenmeng Zhang, Zhenbang Chen, and Weijiang Hong. 2025. Artifact for: Multi-modal Sketch-based Behavior Tree Synthesis. doi:10.5281/zenodo.16921187

Yi Zhang, Sujay Kumar Jauhar, Julia Kiseleva, Ryen White, and Dan Roth. 2021. Learning to decompose and organize complex tasks. In *HLT 2021*. 2726–2735. doi:10.18653/V1/2021.NAACL-MAIN.217

Haotian Zhou, Yunhan Lin, Longwu Yan, Jihong Zhu, and Huasong Min. 2024. LLM-BT: Performing Robotic Adaptive Tasks based on Large Language Models and Behavior Trees. In *IEEE International Conference on Robotics and Automation, ICRA 2024*. 16655–16661. doi:10.1109/ICRA57147.2024.10610183

Received 2025-03-25; accepted 2025-08-12