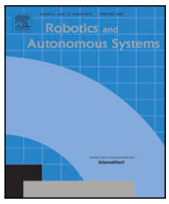
Robotics and Autonomous Systems 194 (2025) 105165

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot



Behavior tree generation and adaptation for a social robot control with LLMs

Sergio Merino-Fidalgo [a,*], Celia Sánchez-Girón [a], Eduardo Zalama [a,b], Jaime Gómez-García-Bermejo [a,b], Jaime Duque-Domingo [a]

[a] ITAP-DISA, Universidad de Valladolid, Doctor Mergelina St., 3-5, Valladolid, 47011, Spain
[b] CARTIF, 4 Francisco Vallés Av., Boecillo, 47151, Spain

ARTICLE INFO

Keywords:
Planning and execution
Networks of robots and intelligent sensors
Mobile robots
Cognitive aspects of automation systems and humans
Large language models

ABSTRACT

Large Language Models have recently emerged as a powerful tool for generating flexible and context-aware robotic behavior. However, adapting to unforeseen events and ensuring robust task completion remain significant challenges. This paper presents a novel system that leverages LLMs and Behavior Trees to enable robots to generate, execute, and adapt task plans based on natural language commands. The system employs ChatGPT to process user instructions, generating initial Behavior Trees that encapsulate the required task steps. A modular architecture, combining the BT planner and a Failure Interpreter module, allows the system to dynamically adjust Behavior Trees when execution challenges or environmental changes arise.

Unlike conventional methods that rely on static Behavior Trees or predefined state machines, our approach ensures adaptability by integrating a Failure Interpreter capable of identifying execution issues and proposing alternative plans or user clarifications in real time. This adaptability makes the system robust to disturbances and allows for seamless human–robot interaction. We validate the proposed methodology using experiments on a social robot across various scenarios in our workplace, demonstrating its effectiveness in generating executable Behavior Trees and responding to execution failures. The approach achieves an 89.6% success rate in a realistic home environment, highlighting the effectiveness of LLM-powered Behavior Trees in enabling robust and flexible robot behavior from natural language input.

1. Introduction

Task control and sequencing has always been a major challenge in any type of autonomous system, especially in robots. As these systems become increasingly complex, the need for robust, flexible, and interpretable control mechanisms has grown, and early approaches such as Hierarchical Finite State Machines (HFSMs) [1], which provided a structured way to define robot behaviors, have fallen short due to their difficulties of modification and scalability. The primary distinction between them lies in their degree of reactivity and the manner in which complex behaviors are represented. Behavior Trees provide a more structured and semantically transparent graphical formalism, which enhances human interpretability and facilitates the analysis of decision-making processes. This increased clarity is especially beneficial when modeling intricate control strategies, as it supports improved modularity, reusability, and maintainability [2].

Behavior Trees (BTs) have emerged as a powerful tool for representing robot behaviors, offering modularity, reusability, and transparency [3]. Originally developed in the gaming industry, BTs have been increasingly adopted in robotics due to their ability to manage complex tasks through a hierarchical structure of conditions and actions. Despite the advantages over hierarchical finite state machines in terms of modularity and reusability, their programming can be complex and tedious. Meanwhile, Large Language Models (LLMs) [4], such as OpenAI's ChatGPT [5,6], have demonstrated remarkable capabilities in understanding natural language and reasoning through complex problems. The convergence of these technologies opens new avenues for human–robot interaction, where natural language commands can be transformed into executable task plans avoiding the task of manually defining and programming BTs.

Despite the potential of BTs and LLMs, their combination presents significant challenges. While BTs are robust and flexible, adapting them to all kinds of unexpected situations requires careful and detailed programming, which requires a great deal of knowledge and effort. On the other hand, although LLMs can generate tasks to a high degree of detail with the appropriate prompt, they still generate text based on previous data patterns and do not understand the content as a human does, so they are not inherently equipped to manage real-time feedback or adapt dynamically during task execution. This limitation restricts the

* Corresponding author.
E-mail address: sergio.merino.fidalgo@uva.es (S. Merino-Fidalgo).

https://doi.org/10.1016/j.robot.2025.105165

Available online 19 August 2025

applicability of these methods in real-world dynamic scenarios where robots must be both reactive and resilient.

In this paper, we propose a novel framework that combines the flexibility of BTs with the reasoning capabilities of LLMs to create an adaptive system for robotic task execution. Our method uses ChatGPT to interpret user commands and generate a BT, which is executed by the robot. A key component of our approach is its adaptability to unforeseen situations, thanks to the Clarifier module, telling the user whenever the LLM does not understand the instruction or the requested actions cannot be executed, and the Failure Interpreter module, which with the occurrence of any issue while on the BT execution, helps the LLM modify the BT in real time to ensure task completion. This integration enables robots to handle external disturbances, adapt to environmental changes, and interact effectively with users.

Although some research have addressed similar systems, the primary contributions of this work are as follows:

- The combination of LLMs and Behavior Trees emerged from the need to balance flexibility and structure in robot behavior generation: LLMs offer the ability to interpret diverse natural language inputs, while BTs provide a robust, transparent, and reactive control framework. Existing methods often rely on either static templates, model training or fine-tuning or opaque neural policies, limiting adaptability and interpretability.

- We introduced the Clarifier and Failure Interpreter modules to address two critical gaps: (i) handling ambiguous or incorrect user input, which is frequently ignored in prior work, and (ii) enabling BT modification in response to execution failures, ensuring runtime adaptability—something missing in systems where BTs are generated once and remain static.

- We emphasized experimental validation in real-world conditions, using a social robot in domestic-like scenarios, unlike prior studies which rely heavily on simulators or constrained robotic arms. This not only demonstrates feasibility but highlights the relevance of our system for assistive applications in non-controlled environments, such as eldercare.

The remainder of the paper is organized as follows: Section 2 reviews related work on BTs and LLMs in robotics. Section 3 details the proposed methodology, including the BT generation process. Section 4 presents the experimental results and an example of a complex action such as searching for a fallen person, Section 5 discusses the contributions and advantages of the system as well as areas for improvement and Section 6 completes with conclusions and future work.

2. Background and related works

2.1. Behavior trees

A Behavior Tree (BT), is a hierarchical inverted tree structure used to represent tasks at an abstract level and the switch between them [7], offering an alternative to Hierarchical Finite State Machines. To enable high-level control of a robot's behavior, SMACH [8] is a task-level architecture for robot control implemented within the Robot Operating System (ROS) ecosystem [9]. It is based on the Finite State Machine (FSM) [10] paradigm and provides a Python-based framework for defining and executing hierarchical and concurrent state machines. In contrast, Behavior Trees offer a different formalism that emphasizes reactivity, modularity, and readability. While SMACH relies on predefined state transitions and often leads to tightly coupled logic, BTs use a tree-based execution model that naturally supports fallbacks, parallel execution, and dynamic behavior switching. This makes BTs generally more scalable and transparent when dealing with large and evolving behavior sets. Furthermore, the graphical representation of BTs tends to be more intuitive and easier to interpret, especially in collaborative or interdisciplinary contexts where clarity of control logic is essential.

Initially developed for computer games [11], BTs have spread to other fields such as robotics due to their reactivity in changing environments, readability and modular nature, which made them easy to modify or expand. All these features make them a significant improvement over finite state machines, which, due to their nature, are less adaptable to situations not contemplated during programming and their modification is difficult and tedious.

In the bottom-right corner of Fig. 1, the LLM's response illustrates both the identified components and a structured representation of a Behavior Tree. The execution begins at the root node (the one on top), propagating a "tick" signal to its child nodes, the ones on the lower branches. Each ticked node evaluates its logic or executes its task and then returns one of three statuses to its parent: *Success*, *Failure*, or *Running*. The simple structure from this example contains the basic nodes to build almost every BT:

1. Control Flow nodes: These nodes manage their child branches execution based on a predefined logic.

- Sequence Node (→): Ticks child nodes from left to right, executing (*Running*) next if previous returned *Success*. If one of its child nodes returns *Failure*, the sequence node stops ticking and returns *Failure*. It only returns *Success* if all child nodes return *Success*.

- Fallback or Selector Node (?): Ticks child nodes from left to right, executing (*Running*) next if previous returned *Failure*. If one of its child nodes returns *Success*, the fallback node stops ticking and returns *Success*. It only returns *Failure* if all child nodes fail.

2. Execution Nodes: These nodes represent the actionable components of a BT:

- Action Nodes: Perform specific tasks or robot actions designed by the user. They return *Success* when the task is completed, *Failure* if it cannot be performed, or *Running* if still in progress.

- Condition Nodes: Evaluate Boolean conditions, such as checking sensory data or environmental states. These nodes return *Success* if the condition is true or *Failure* otherwise.

Behavior Trees are increasingly employed to design flexible and reactive control systems for robots, offering greater modularity, transparency, and scalability compared to traditional finite state machines [12]. Their structured hierarchy facilitates the integration of new behaviors without disrupting existing logic, making them particularly effective in dynamic contexts such as mobile robotics, manipulation, and Human-Robot Interaction (HRI) [13,14]. Building on these foundations, recent studies have demonstrated the versatility of BTs in increasingly complex domains. For example, Sprague et al. (2022) [15] proposed a method to integrate neural network controllers into Behavior Trees while preserving formal performance guarantees, bridging data-driven control with hybrid system stability. Wu et al. (2025) [16] proposed EffBT, a synthesis framework for correct and efficient BTs from formal GR(1) specifications, while our system bypasses formal models and allows real-time generation from user input. Albi (2023) [17] designed a BT-based mission management system for UAVs with strong execution reliability, though without interactive or language-based design. Scherf et al. (2023) [18] enabled BT learning from human demonstrations using visual input and web interaction, whereas our method relies solely on natural language, simplifying the process.

2.2. LLMs for robotic tasks

The integration of Large Language Models (LLMs) into robotics has opened new avenues for enhancing task planning, execution, and adaptability. LLMs, such as GPT models [19], are pretrained on vast datasets

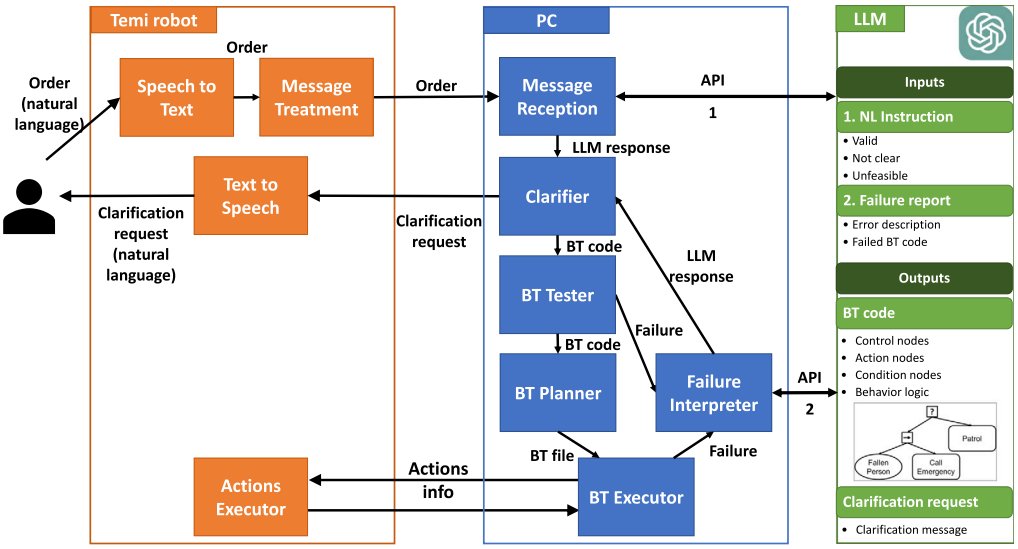


Fig. 1. Graphic representation of our method. The orange frame shows the part of the system executed by the social robot, while the blue one represents the modules in the computer. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

encompassing diverse human knowledge, allowing them to interpret natural language commands, reason about tasks, and generate structured outputs. This capability makes them particularly promising for generating Behavior Trees to control robots in dynamic environments.

Early approaches laid the groundwork for combining vision, language, and action, paving the way for advanced control systems in robotics. For instance, Palm-E, an embodied multimodal language model, was among the first to extend LLMs for embodied tasks by incorporating vision and sensor data to ground natural language reasoning in physical environments [20]. Similarly, RT-2 introduced Vision-Language-Action (VLA) models that successfully transferred web-scale knowledge into robotic control systems, enabling robots to perform tasks informed by extensive online information [21].

In order to complete complex tasks in dynamic environments, the generation of Behavior Trees took over and became the most used method, especially focused on robotic arms. LLM-BRAIn was one of the very first, a transformer-based Large Language Model fine-tuned from the Stanford Alpaca 7B model to generate static robot Behavior Trees from textual descriptions [22]. BTGenBot presented a novel approach to generating Behavior Trees for robots using lightweight Large Language Models with a maximum of 7 billion parameters, trained with a fine-tuning dataset based on existing Behavior Trees [23]. LLM-BT achieved robotic adaptive tasks based on LLMs and Behavior Trees using ChatGPT and semantic maps are constructed by an object recognition algorithm to understand the environment, having been tested in simulation [24]. LLM as BT-Planner leveraged LLMs for BT generation in robotic assembly task planning and execution where four in-context learning methods are introduced via natural language [25].

2.3. Social robots

The control of social robots focuses on enabling systems to interact effectively with humans in a variety of social contexts. These robots require advanced control architectures to interpret social cues, engage in natural communication, and adapt to human dynamics. Behavior Trees have emerged as a perfect method to deal with human–robot interactions, due to their adaptability to dynamic environments and unexpected situations, as it is human behavior. Cooper et al. introduced a Behavior Tree based design of long-term social robot behavior, where a human approaches and begins the interaction or the robot actively navigates and searches for a specific user to deliver a reminder [26].

On the other hand, LLMs offer significant human–robot interaction via natural language, with advanced conversational skills and versatility in managing diverse, open-ended user requests in various tasks and domains [27]. Hanschmann et al. developed Saleshat based on a commercial social robot and ChatPGT, which emphasizes refined natural language processing and dynamic control of robot physical appearance through the LLM [28].

However, Behavior Trees cannot comprehend the full range of human behavior and are therefore often limited in the control of social robots. LLMs, on the other hand, have generally been used to interact verbally to follow human behavior and interact with them, but without performing physical actions. This is where our proposal comes from, where we put together the advantages of BTs and LLMs and combine them to create a system where the person can interact with the robot through natural language and the robot is able to carry out actions in a structured way, beyond just having a conversation.

3. Methodology

The proposed method integrates natural language processing to handle user commands. These commands are used to generate Behavior Trees based on a Large Language Model. The resulting BTs are then executed by a robot, which performs the requested actions. The system also allows the behavior to be adapted or modified, either by the user or to ensure task completion.

This approach is intended to be broadly applicable across diverse contexts. A key application area under consideration is its deployment in the homes of elderly individuals, where a social robot offers companionship and executes tasks as directed by the residents. The full source code and prompt details are available on our GitHub repository.[1]

The integration of Large Language Models with Behavior Trees offers a powerful framework for robot behavior generation by combining the linguistic expressiveness and generalization capabilities of LLMs with the modularity, reactivity, and interpretability of BTs. BTs provide a structured and human-readable representation of robotic decision-making, enabling intuitive debugging, runtime adaptation, and explicit control flow. When paired with LLMs, this representation becomes dynamically generable from natural language, empowering non-expert users to author complex behaviors through simple verbal commands.

[1] https://github.com/sergifiUVa/Behavior-Tree-Generation-and-Adaptation-for-a-Social-Robot-Control-with-LLMs.git

Table 1
Qualitative comparison of LLM-based BT generation systems.

System	Input	HR interaction	Pre-testing	Adaptation	Real-world validation
LLM-BRAIn [22]	NL tuned	Limited (one-shot)	No	No	No (sim)
BTGenBot [23]	NL + prompts	No	Yes (static + sim)	No	Yes (TurtleBot3)
LLM-BT [24]	NL + sem. map	Partial (parser only)	No	Yes (BT update)	No (sim)
LLM-as-BT-Planner [25]	NL + RDF	No	Partial (sim)	Partial	Yes (Franka Panda)
Ours	NL	Yes (NL + Clarifier)	Yes (BT Tester)	Yes (Clarif. + Fail. Int.)	Yes (Temi robot)

Although a direct quantitative comparison is not feasible – due to differences in experimental setups, domains, and evaluation criteria – we provide here a qualitative assessment based on key system capabilities. LLM-as-BT-Planner emphasizes hierarchical planning for structured assembly tasks, integrating in-context learning and simulation feedback, but lacks real-time error handling or user interaction during execution. LLM-BRAIn enables fast generation of Behavior Trees using a fine-tuned lightweight model, yet produces static plans and offers no runtime adaptation or clarification mechanisms. BTGenBot achieves robust syntactic and semantic correctness using compact LLMs, but is limited to offline generation and cannot respond to task failure or user queries. LLM-BT introduces adaptability through BT expansion based on semantic maps and a BERT-based parser, but it depends on predefined templates and does not support user feedback or execution in real-world social contexts. In contrast, our approach uniquely combines natural language understanding, user clarification, structural validation, execution-time repair, and task prioritization, all within a unified and fully deployed system on a real social robot. This makes it particularly well-suited for dynamic, open-ended environments – such as eldercare – where transparency, reactivity, and ease of interaction for non-expert users are critical. Table 1 summarizes the key features of the LLM-based BT generation methods.

Unlike prior works, our approach explicitly considers robust user-in-the-loop adaptation, error handling, and interactivity, all within a unified framework. Furthermore, those previous studies either do not report BT generation times, or present significantly longer processing durations, limiting their applicability in real-time or user-facing scenarios. Finally, while most existing approaches have been validated exclusively in simulated or narrowly defined domains, our method has been tested in a real-world robotic platform. Additionally, the envisioned application scenario is domestic environments, particularly in assistive contexts such as eldercare.

Our method is designed to be integrated into a research line to improve the lives of elderly people who live alone in their homes. The home setup includes a home automation system with distributed sensors that monitor the environment and resident activity. A local computer manages devices, a voice assistant, and a social robot, while also handling notifications and communication with caregivers. All data is sent to a remote server, where caregivers and coordinators can access real-time information through a web application.

3.1. System architecture

The proposed system architecture to generate and execute Behavior Trees is illustrated in Fig. 1. The system integrates a robot, a PC, and a Large Language Model accessed via API. Each component interacts through a structured pipeline that facilitates natural language understanding, BT generation, and task execution. The main modules are described as follows:

1. Natural Language Interaction

The process begins with the user issuing a task command in natural language. This input is processed in the robot through a Speech-to-Text module, which converts the spoken command into a textual format. This text is then passed to the Message Treatment module for pre-processing and structuring, creating a json structured message that with all the information needed to create and execute the task.

2. Task Interpretation and Behavior Tree Planning

The pre-processed message is sent to the PC via MQTT protocol, as it is the communication method between the robot and the computer. The message is received by the Message Reception module. This block extracts the message's information and sends it to the LLM via API. We use ChatGPT with a well designed prompt, which consists of three main parts:

- A basic part that defines the behavior of the LLM. The model is expected to: (i) generate the structure of Behavior Trees using the *py_trees* library, leveraging its built-in control nodes such as *Sequence*, *Fallback*, and *Parallel* to ensure structural validity and (ii) request clarification from the user when the instruction is ambiguous, unrecognizable, or infeasible.

- A description of the environment which consists of the rooms the robot can go to, the actions the robot can perform, and the interaction between the system and the robot. This information gives context to the LLM to obtain better outputs and to know when something is wrong.

- Must follow rules, where we define key aspects of the Behavior Trees, such as fixed structures for a specific node, details of action nodes or when the BT ends. Additionally, an example output is provided to improve the LLM behavior.

Thanks to this prompt, the LLM can generate BTs from all kinds of instructions, from simple ones that specify the actions to be performed by the robot (*"Go to my bedroom and call Mary"*) to more complex inputs such as *"Tell me if there is someone in the kitchen, I want to do the dishes"* in which ChatGPT extracts the actions to be performed by the robot (go to the kitchen, look for someone there to later come back and tell if there is a person or not) and generates the BT or, if it did not understand, asks the user for clarification. Execution logic is not needed to be included in the instructions, because the LLM is able to understand it from natural language and translate it into the BT structure.

In the design of the system, we deliberately kept the prompt as simple and focused as possible, avoiding unnecessary complexity that could distract the model from its primary objective: generating the logic of the BT as effectively as possible. Reactivity, which is an intrinsic feature of BTs, is intentionally delegated to the predefined actions, which have been carefully designed to autonomously handle unusual situations that may arise during execution. This ensures the robustness and reliability of the overall behavior, preventing the BT from getting stuck in any action under any circumstance.

Next, we explain every module of the system involved after the LLM generates an output.

Clarifier: This module receives and analyzes the generated reply from the LLM. If the output begins with ChatGPT's code block identifier (*"'python*) or directly with a function definition (*def*), it is assumed to be BT code, and this module sends it to the BT Planner. However, if the LLM returns that it did not understand what the user wants, it is ambiguous, or the requested task is impossible to perform, the Clarifier sends a json message to request clarification or suggest possible interpretations from the user in natural language thanks to robot's Text-to-Speech module. This ensures that the generated BT corresponds to the user's intent. In case the LLM returns a BT that syntactically appears valid but contains semantic or execution errors, the system will detect the failure at runtime. In such cases, the execution is interrupted and passed to the Failure Interpreter module.

BT Tester: To evaluate the structural soundness of the generated Behavior Trees, we developed a module called BT Tester. This module analyzes the BT code to extract all actions and the connections between nodes. It then generates a simplified copy of the original BT, in which each action node is replaced with a lightweight version that only returns either *Success* or *Failure*. Using the *pytest* testing framework, BT Tester exhaustively executes all possible combinations of return values across the tree. After completing every run, it records which nodes were ticked and compares this to the full set of nodes. This exhaustive approach ensures that the structure and logic of the BT are valid—that all nodes are reachable and can be executed under at least one condition. If any node is found to have never been ticked in any combination, the system logs the error, identifies the unreachable node, and forwards this information to the Failure Interpreter module for further analysis and correction.

BT Planner: The BT Planner receives the code of the Behavior Tree from the BT Tester. It is embedded into a generalized structure common to all BTs and saved in a file. This structure includes the necessary calls to predefined actions and libraries, an error logging mechanism in case failures occur, and a main function to execute the Behavior Tree. In addition, it identifies who requested the BT to extract the execution priority. For this purpose, each agent able to request the execution of a BT has a value associated with it. This value completes the file's name, so the system knows when the plan has to be executed. Finally, the module notifies the BT Executor about the new plan.

BT Executor: Whenever this module receives a message with the name of a new plan from the BT Planner, the plan is added to an execution queue based on its priority. If no action is running, the BT Executor starts the execution of the Behavior Tree from the execution queue as a subprocess. If a high-priority emergency task is received, the system immediately interrupts the currently running action and switches to executing the emergency behavior. The actions to be performed by the robot are sent via MQTT messages, just as the robot sends back information about whether it has finished an action, the position where it is, or the user's answer to a question. When the Behavior Tree execution is finished, the BT Executor terminates and deletes the subprocess, and remains ready to execute the next task. In case it receives multiple plans to be executed, the module queues them according to the priority of the actions and the order of arrival. If the execution of the Behavior Tree fails, either because of an error in the code or because during runtime it has returned *Failure*, the BT Executor stops the subprocess and sends the error to the Failure Interpreter module.

Failure Interpreter: This block ensures the robustness and adaptability of the system during task execution. Addresses one of the fundamental challenges in robotics: the ability to recover and adapt when a robot encounters situations where it cannot complete a task as planned. Through its integration with a Large Language Model, the system combines reasoning and creativity to propose solutions. When a node cannot be ticked or a Behavior Tree execution returns *Failure*, the Failure Interpreter module automatically retrieves the error from the log file and queries the LLM with a structured behavior prompt, the failure description (e.g., "GoToKitchen node cannot be ticked", "Robot cannot reach target location", "Target location does not exist", or "Missing argument on action Go") along with the original BT code. The LLM then outputs a modified version of the BT, which is passed to the Clarifier module for validation and, if necessary, further refinement before execution.

The input to the LLM consists of a prompt that is essentially identical to the one used for BT generation, with the exception that the initial section is modified to specify that its purpose is to correct errors. It also includes the reported error and the code of the BT that failed. This ability to dynamically revise plans using natural language to help the LLM solve the problem and high-level task understanding gives the robot human-like problem solving capabilities.

Finally, robot actions are performed by the Actions Executor module, which executes the primitive robotic actions defined in the BT, such as navigation or videoconference. These actions are the result of the BT execution process, sent in a json format where the topic specifies the type of action to perform and the payload the necessary information to carry it out, such as the destination location and the speed to send the robot somewhere. This module returns information from the robot to the BT Executor module in order to execute the Behavior Tree successfully.

The process of generation of a BT and clarification whenever the LLM does not understand the instruction can be described as follows in pseudo-code from Algorithm 1, where ϕ stores generation prompt, δ is LLM input, μ is LLM output, M is our LLM model, Q is execution queue, ϵ is the agent who requested the order and τ is priority. The user introduces an instruction as an input for the LLM, that along with the prompt, generates an output who reaches the Clarifier. If LLM output is not a BT (because ChatGPT did not understand the instruction or the robot cannot perform the task), the system asks the user for clarification. Otherwise, the system assigns the priority to the BT and introduces it in the execution queue, sorting it by priority.

Algorithm 1: BT generation and clarification

```
Input: inputSTT()
δ ← inputSTT();
while True do
    μ ← M(φ ∪ δ);
    if notBT(μ) then
        askClarification();
        break;
    else
        τ(μ) ← τ(ϵ);
        add(Q, μ);
        sort(Q, τ);
```

The execution and failure handling process is shown in pseudo-code of Algorithm 2, where δ is LLM input, μ_i is LLM output stored in execution queue, Q is execution queue and λ is the BT Executor. If the BT Executor module is idle and the execution queue contains any plans, λ executes the first BT of the queue. When the BT ends, if it returns *Failure* the Failure Interpreter module gets the error, which is saved as a new input for the LLM to modify the BT. If the execution return *Success*, the BT Executor module removes finished BT, ready to execute next.

Algorithm 2: BT execution and failure handling

```
while True do
    if idle(λ) and notEmpty(Q) then
        λ(μ1);
        if status(μ1) = Failure then
            error ← failureInterpreter();
            δ ← error;
        else if status(μ1) = Success then
            empty(λ);
            break;
```

To provide a comprehensive understanding of how our method operates, we present an example of a complete interaction with the system. The process begins with a user issuing a natural language instruction to the robot: *"Temi, go to the kitchen and say 'Hello, my name is Temi'."* The spoken sentence is transcribed on the robot and sent via MQTT to the PC; the Message Reception module packages it with a system prompt and forwards it to the LLM. ChatGPT parses the order, infers the two actions GoToKitchen and SayHello, and replies with a function which contains the logic and nodes of a Behavior Tree, as shown in the code below:

```
def create_behavior_tree(mqtt):
    try:
        # Behavior Tree Nodes
        root = py_trees.composites.Sequence(name="Root",
              memory=True)
        sequence1 = py_trees.composites.Sequence(name="
              sequence1", memory=True)
        move_destination = MoveToDestination(name="
              GoToKitchen", destination="kitchen", mqtt=
              mqtt)
        speak_message = SpeakMessage(name="SayHello",
              message="Hello, my name is Temi", mqtt=mqtt)
        reminder = Reminder(name="Reminder", mqtt=mqtt)

        # Nodes children's
        sequence1.add_children([move_destination,
              speak_message])
        failure_is_success = py_trees.decorators.
              FailureIsSuccess(name = "failure_is_success",
               child = sequence1)

        # Add branches to root
        root.add_children([failure_is_success, reminder])

        return root
    except Exception as e:
        logging.error(f"Error in create_behavior_tree: {e
              }")
```

The Clarifier inspects that reply, requesting clarification if the LLM is unable to generate correct BT code due to an ambiguous or invalid instruction: if it is a BT code, the BT Tester analyzes its structure and logic looking for a non-tickable node. Because it is a valid BT structure, the module passes it to the BT Planner, which builds the full executable BT code, stamps the file with the user's priority and notifies the BT Executor about this new task. This module receives the message and queues the new plan based on its priority. When the BT Executor becomes free, it dequeues the plan and executes it as a subprocess. This ticks the tree node-by-node, and for every action node publishes MQTT messages that steer the robot's navigation stack or speech engine. If all nodes return *Success*, execution ends with a friendly reminder node and the subprocess is finalized and the plan file erased; if any node returns *Failure* – for instance the robot cannot reach the kitchen – the BT Executor ends the execution and calls the Failure Interpreter. That module retrieves the relevant log fragment which contains the failure, sends both the error context and the original BT back to the LLM, receives a patched or clarified tree, and routes it again through the Clarifier to make sure the LLM response is valid, so the updated plan is replanned and re-executed—closing a fully autonomous sense–plan–act–repair loop.

4. Experiments and results

Experiments have been carried out in a laboratory to show that it can be implemented in elderly homes. Temi robot [29] was deployed in our work environment, which was divided into rooms to recreate a house distribution, and we tried multiple natural language instructions.

The system is planned to be integrated into a home automation setup (Fig. 2) consisting of a network of small sensors distributed throughout the house to monitor environmental conditions and the resident's activities. Data from these sensors is processed by a local computer, which also manages actuators, a voice assistant (e.g., Alexa), and a social robot that interacts with the resident. This computer handles notifications and calls to the caregiver's mobile phone and communicates with a server-side web application. The server stores all relevant data, which is accessible to caregivers and coordinators through a web interface that provides real-time updates on the home and the resident's status [30].

To verify our proposal, we ran several experiments divided into two parts: the generation of Behavior Trees after introducing a natural language order and the final test including task execution and modification on a real social robot, as shown in Fig. 3.

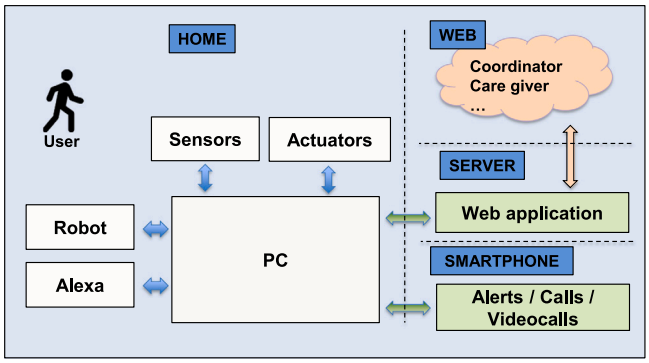


Fig. 2. General overview of the environment.

For the experiments, we selected six different actions that the Temi robot is capable of performing. Speak allows the robot to deliver a spoken message; Go commands it to navigate to a specified location; Ask prompts the user with a question and stores the response; Videoconference initiates a video call with one of the pre-configured contacts; and Alert is used to notify the caregiver by sending a message to their mobile phone whenever the system detects an issue or requires attention; The FallCheck action is designed to detect whether a person has fallen by capturing an image of the surrounding environment, analyzing it using a trained computer vision model, and returning the result in terms of the number of people identified as fallen versus those who are standing or in a normal posture. This action is integrated into the BT as a perception module and can be triggered periodically or in response to specific events. The output includes structured information ({fallen: 1, not_fallen: 0}), which can be used by subsequent decision-making nodes to trigger appropriate actions, such as alerting a caregiver, approaching the person, or initiating a safety verification sequence.

In order to detect fallen people in an image, a methodology based on deep learning and image processing has been implemented. The image captured by the Temi robot's camera is sent to the fall detection model via MQTT. Once the image has been received, the next step is to process it using a model specialized in object and person detection. For the proposed methodology, an approach using YOLOv8s. YOLO is a model for detecting objects in the image; for this purpose, it returns the bounding box of these objects, which are 4 coordinates that define the location of the element in the image. The idea is to use YOLO to obtain an image of the person cropped from the coordinates of the bounding box of the person, as well as to obtain the coordinates of other objects of interest. Chairs, beds and sofas have been defined as elements of interest, since it has been considered that they could influence in detecting whether the person is down or in another position. When performing this detection, a confidence threshold was established for the identification of the different classes of objects.

To help the network distinguish whether a person is fallen or lying on a bed or sofa, a function was integrated to detect the level of overlap between the person and the object in the image by applying the IoU metric [31]. For this purpose, it is checked whether the area of the rectangle that forms the bounding box of the segmented person intersects with the area of the bounding box of each object of interest detected. If there is a significant overlap, above a certain threshold, the coordinates of the persons in the image and the objects that overlap with them are preserved. With these we assume a possible interaction of the person with the object. If the object of interest is not detected or is not overlapping with the person, then we indicate that the bounding box is null. Thus, after detection we will obtain the image of the person cropped with respect to the original image and four sets of four coordinates, by combining the bounding box of the person and the three bounding boxes of the objects present or not.

(a) Temi asking the user a question (b) Temi checking if someone is fallen

Fig. 3. Temi robot performing different tasks.

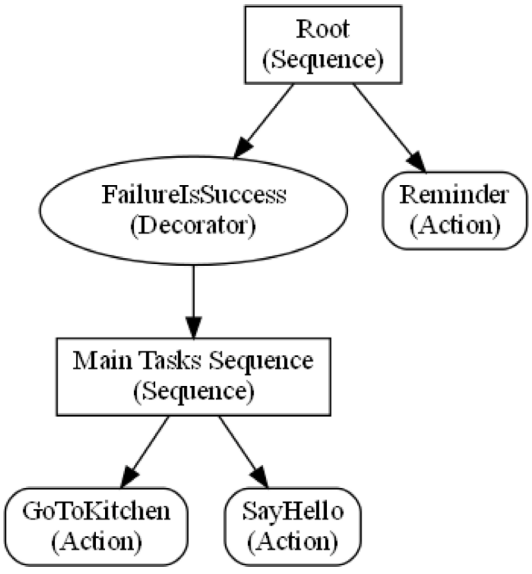


Fig. 4. Graphic representation of BT generated from first command.

Once the model has detected the people in the image and the objects that are overlapping their area, the cropped images and corresponding coordinate data are fed into a neural network to perform the fall detection process [32]. This architecture is composed of multiple submodels trained using the Cross Validation Voting (CVV) method [33], which enhances generalization and robustness. During training, the dataset is divided into multiple slots, and a separate model is trained for each one. All models share the same structure, consisting of a ConvNeXt network and a Fully Connected (FC) layer. The FC layer processes a 16-dimensional vector derived from the bounding boxes, one for the person and up to three for nearby objects of interest (sofa, bed, chair), while the ConvNeXt network processes the cropped image of the person. The outputs of both branches are concatenated and passed to a final dense layer, which produces a single output value representing the binary classification: fallen or not fallen. This system was trained using an extended version of the Fallen People Detection Dataset (FPDS) [34], which consists of 6982 images, including 5023 images of falls and 2275 images of non-falls, depicting individuals in various everyday scenarios such as standing, sitting, lying on a sofa or bed, or walking. The dataset encompasses a wide range of indoor environments, poses, occlusions, and lighting conditions, contributing to the model's strong generalization capability in real-world conditions. The trained system achieved an accuracy of 92.95%, a recall of 89.67%, and an F1-score of 90.52%, as shown in the confusion matrix provided in the original article. These metrics demonstrate a well-balanced performance between sensitivity and specificity, which is an essential requirement for emergency applications. Moreover, when tested in 118 real-world scenarios using the Temi social robot, the system reached up to 96% accuracy, outperforming other architectures such as ResNet and earlier segmentation-based models.

For the priorities associated with the BTs for their execution order, we defined three agents: the user who is the elder person who directly interacts with the robot, the caregiver who sends the written instruction and is received by the Message Reception module and a local service with predefined BTs for daily tasks or emergency situations, such as sending the robot to the bedroom and saying good morning when the user wakes up or calling the caregiver if the person leaves home at 3 a.m., respectively. These predefined actions are directly sent to the BT Executor to be added to the execution queue, securing a quick and robust execution. The highest priority belongs to predefined emergency tasks, followed by user's orders, caregiver's instructions and autonomous routine plans.

We also defined in the prompt the locations where the robot can navigate in the house and the contacts saved for videoconference, so if a requested task includes the action Go to a non-available location or the user wants to call a non-saved contact, the system will inform the user about this issue.

4.1. Natural language recognition and BT generation

In this phase, we evaluated the ability of the system to generate accurate and interpretable BTs from natural language commands. The process began with the user providing a task description, for this test, a written order, which was processed by the LLM to generate the code of a Behavior Tree. Some of the introduced commands were:

Temi, go to the kitchen and say "Hello, my name is Temi"

Can you please go to the garage and, if there's someone, call David?

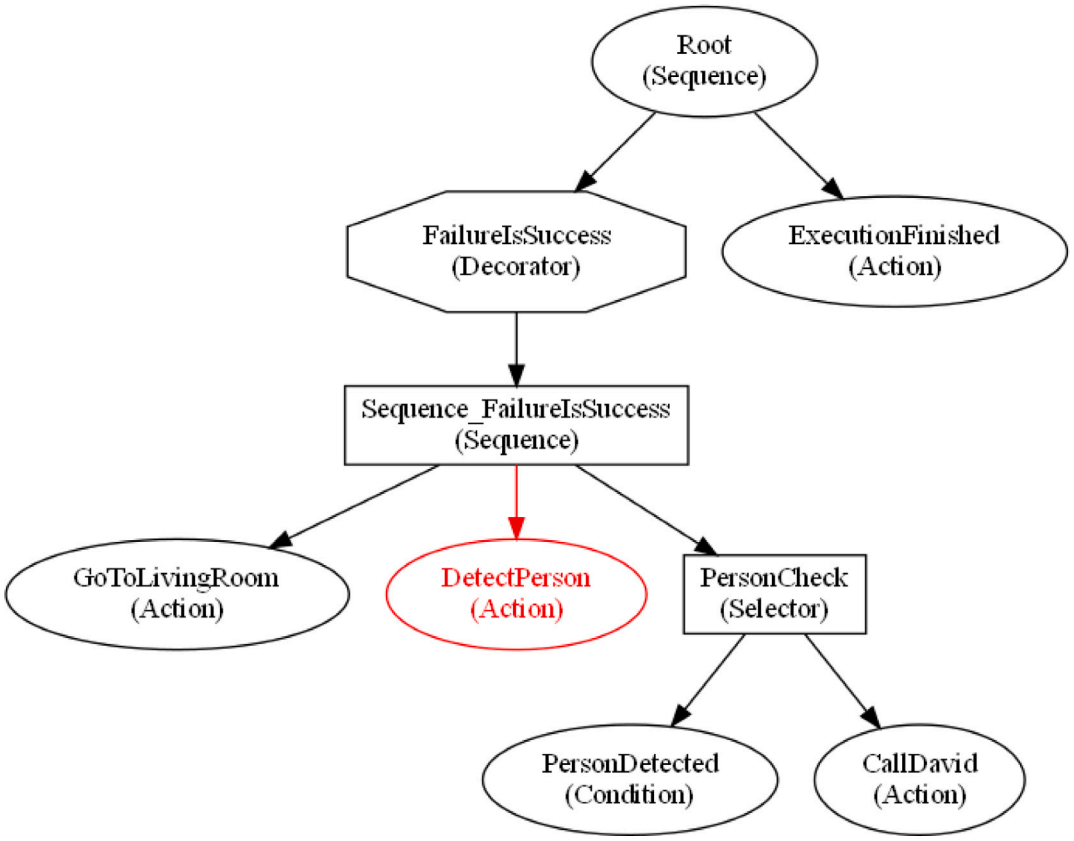


Fig. 5. Initial and modified BT after added nodes (in red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Bring me my mobile phone, it's in my bedroom

The results demonstrated the system's capability to generate BTs efficiently, with all clarifications resolved within two iterations on average. For example, first command was quickly identified and generated in BT in a fast and correct way, whose code was perfectly executable. Fig. 4 shows the representation of the generated BT.

The sequence of actions in the bottom represents the task from the command, first goes to the kitchen, to later reproduces the message. FailureIsSuccess node returns *Success* if childs return *Failure*, which guarantees the execution of the Reminder node, both being a standard in every BT. This structure is designed to maintain control over the outcome, regardless of the specific structure generated by the LLM. Specifically, we define a blackboard that serves as a shared memory space. In each action node, if an error occurs or the node returns *Failure*, this event is recorded on the blackboard. The FailureIsSuccess decorator is used to ensure that the final node – the Reminder node – is always executed, regardless of the outcome of the preceding nodes. This Reminder node checks the blackboard, and if any failure has been recorded during execution, it forces the entire BT to return *Failure*.

This last node reproduces a useful message for the user, like a tip on how to use the robot, when the actions part was executed successfully. If not, this node sends the error (saved by the node where it occurred) to the Failure Interpreter module.

For the second command, we introduced a location that was not registered in the house information, so the robot cannot navigate to it. Therefore, LLM reported this problem to the Clarifier module and then communicated to the user the message "*The requested location does not exist*". After selecting the "living room" as the new target location, ChatGPT generated the black nodes shown in Fig. 5, which was queued and executed by the BT Executor. However, the PersonDetected condition returned an error because no FallCheck action had been executed beforehand. The Failure Interpreter module identified this issue and reported it to the LLM, which then corrected the behavior. As a result, the modified code included the DetectPerson node (highlighted in red), allowing the BT to execute successfully.

The last example includes an action the robot cannot perform (pick up and bring objects), so LLM replied through the Clarifier module pointing out this issue. And as long as it is something the robot cannot carry out, no BT was generated.

The BTs generated to perform simple tasks (one or two actions) were successful 97.3% times, using 30 different instructions 5 times each, which resulted in 146 success out of 150 attempts. For complex tasks (more than two actions or more control nodes needed), the percentage reached 92.7%, following the same evaluating method and obtaining 139 correct outputs.

Additionally, we conducted a test in which an experienced user manually built the BT code for the first two instructions. This manually created BT was then compared with the versions generated by the system (LLM-based). The comparison revealed several key differences. In terms of time, the LLM-generated BTs were produced within seconds, whereas the manual implementation required several minutes per task, depending on complexity. Manually created BTs often used shorter or less descriptive variable names to save programming time, while the LLM-generated BTs consistently produced more explicit and self-explanatory names. Regarding validation, manual BTs were incrementally reviewed and tested by the user during the development process, adding to the overall programming time. In contrast, the LLM-based generation does not include an inherent validation step. Instead, validation is performed post-generation by the BT Tester module. If execution failures are detected, they are subsequently handled and corrected by the Failure Interpreter module. Overall, although expert users are more likely to successfully develop very complex tasks, this

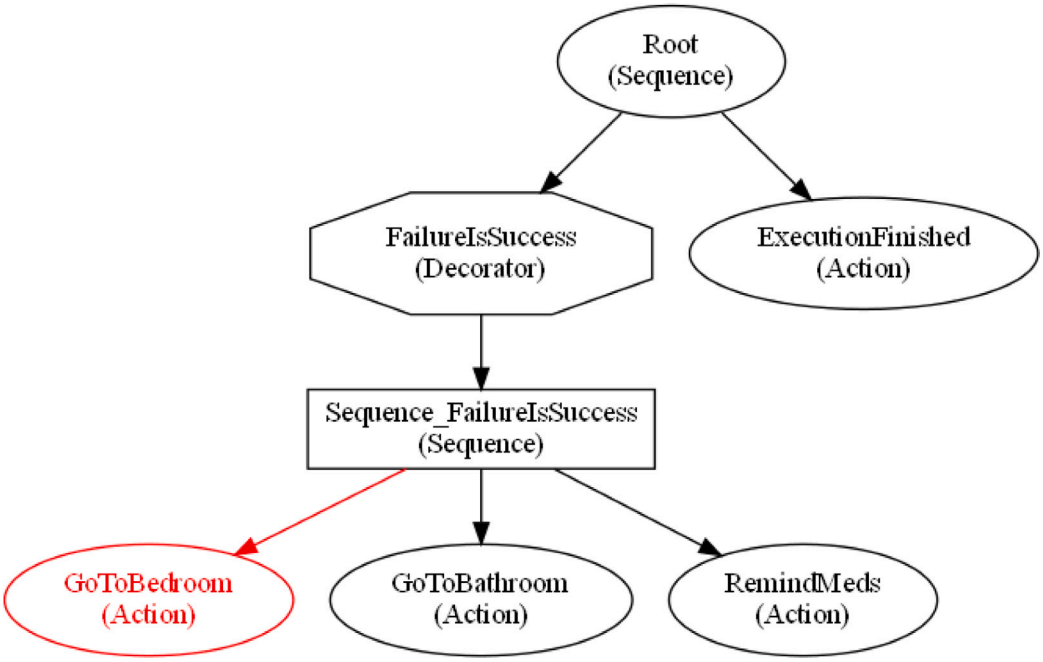


Fig. 6. Initial and modified BT after added nodes (in red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

LLM-based system reduces development time by generating executable BTs directly from natural language input. This enables non-expert users to define complex robot behaviors without programming knowledge, making human–robot interaction more accessible and intuitive.

4.2. Real robot experiments

To evaluate the performance and success rate of the system, 25 users with no prior experience were asked to give the robot five types of instructions: a simple task (e.g., two sequential actions), a complex conditional task (involving multiple actions and decision-making), an ambiguous request (not explicitly stating the required actions), an incorrect instruction (with invalid parameters, such as unknown locations or contacts), and an impossible task (requesting actions beyond the robot's capabilities, like grabbing objects).

To test robustness and adaptability, 5 unexpected situations were introduced in each set of 25 trials, including environmental changes (e.g., blocked paths), missing user responses, and third-party interference. These scenarios aimed to simulate realistic conditions and assess the system's response to unforeseen events.

We now provide an example of the system behavior in one of these situations. We executed a plan (represented by the black nodes in) Fig. 6 where the robot, initially located in the hallway, was instructed to go to the bathroom and remind the user to take their medication. However, during execution, an unforeseen situation was introduced: the hallway door – part of the robot's shortest route – was found closed. As the robot cannot open doors and always selects the most efficient path, the plan returned *Failure* because the robot could not reach the location. At this point, the Failure Interpreter module queried the LLM and it inferred that since the bathroom is also accessible via the bedroom, the plan could be modified to take this alternate route. The added node in red finally allowed the robot to complete the task.

In another trial, the robot was unable to reach the kitchen due to a closed door and lack of alternative paths. The system detected the issue and informed the user: "I'm sorry, I cannot reach the kitchen".

The results from the experiments are presented in Table 2. Clarified column shows the successful applications of the clarification feature per instruction out of the total uses, which means that if a user input needed to be clarified three times, if the LLM finally generated a BT, the table displayed 1/1. The Adapted column indicates the same two parameters but when the BT needed to be modified due to a wrong coding or because a node returned *Failure*. Finally, the Overall column illustrates the successful tests of each kind of instruction out of the 25 total.

Table 2
Results by type of task.

Task	Clarified	Adapted	Overall
Simple Task	2/2	3/3	25/25
Complex Task	9/11	13/15	21/25
Ambiguous Task	18/22	8/9	20/25
Wrong Task	23/25	6/8	22/25
Impossible Task	24/25	0/0	24/25
Total	76/83	30/34	112/125

The success rate of the general system was 89.6% after 125 tests. In total, there were 13 failures, 6 of them caused by repeated wrong and impossible instructions, which led the system to stop asking for clarification and not generating a BT. Three more failures occurred when the task logic expressed by the user was unclear or difficult to understand, causing the system to execute BTs that did not meet the user request. Another two failures were caused by the robot not being able to reach the target location, and the last one was caused by an unexpected videoconference malfunction.

To further assess the subjective user experience, we administered the QUESI questionnaire [35,36] to 10 participants, comprising both men and women aged between 25 and 65 years. All participants had a university-level education but no prior experience with the system. This evaluation was conducted as a separate experiment from the performance assessment previously described, with a different group of users. While the earlier study focused on measuring execution success and system reliability, this experiment aimed to qualitatively assess how intuitive and user-friendly the system is from the user's perspective.

Before starting the experiment, participants were informed about the robot's capabilities, including the actions it can perform, the available locations, and the list of known contacts. They were also given a few example instructions. During the session, users were encouraged to

explore different ways of expressing instructions and testing the robot's understanding.

Each participant was then asked to provide between 3 and 5 instructions for the robot to perform, including at least one involving unavailable actions or locations. For instance, some participants used complex and natural language instructions such as: *"There is a cup at the living room and a bottle at the kitchen. Go to the location where the cup is and say that you have located it."* The success rate of the requested task executions was similar to the prior experiment, reaching 88%. The analysis of their responses revealed a generally high perception of intuitive use. The average scores across the five subscales were notably high: Subjective Mental Workload (M = 4.47, SD = 0.55), Perceived Achievement of Goals (M = 4.17, SD = 0.39), Perceived Effort of Learning (M = 4.47, SD = 0.67), Familiarity (M = 4.37, SD = 0.29), and Perceived Error Rate (M = 4.05, SD = 0.64). These results suggest that users found the system easy to understand and operate, requiring little effort to learn and interact with, while encountering few errors during use. Overall, the high QUESI scores align with the system's performance and indicate strong support for its intuitive usability.

5. Discussion

The proposed system demonstrates a novel approach to generating and executing robot Behavior Trees by leveraging a Large Language Model for both initial task planning and real-time adaptation. This integration of natural language understanding, BT planning, and failure interpretation highlights several strengths and introduces new perspectives on the use of BTs in robotics and artificial intelligence. Natural language as input simplifies the use of the system, especially when it comes to a non-expert person such as elderly people. A meticulously designed prompt based on strict rules and restricted to some of the actions the robot can perform ensures a highly reliable LLM output. The inclusion of the Clarifier module whenever the system does not understand what the user wants and especially the Failure Interpreter module, as it ensures that the system can handle unforeseen situations with minimal disruption, raises the robustness of the system.

Our approach achieves a 89.6% success rate in a real scenario, where a physical robot executes BTs generated from natural language commands in a real-world environment recreating a real house. Nonetheless, it is important to recognize that direct comparisons are inherently difficult, given the significant differences in system architectures, experimental methodologies, and deployment contexts across existing works. Compared to mentioned prior work, our system offers a competitive success rate while requiring no training or fine-tuning. Nonetheless, it is important to recognize that direct comparisons are inherently difficult, given the significant differences in system architectures, experimental methodologies, and deployment contexts across existing works. BTGenBot reports 88.9% success on a real robot but depends on fine-tuned LLMs trained on BT datasets, limiting generality. LLM-BT reaches 85% success but only in a specific task in a simulated environment. LLM-BRAIn and LLM as BT-Planner focus on BT generation without reporting any success rate from execution trials.

While the success rate is slightly lower than manually built BTs, our approach offers significant practical advantages. It allows untrained users to generate BTs from natural language, without requiring programming or BT design expertise. The system supports arbitrary, flexible commands, enabling rapid behavior creation and adaptation, which greatly reduces development time. Unlike prior approaches – which include methods based on pre-trained or domain-specific models, those that do not support Behavior Tree adaptation upon failure, and others that have not been tested in real-world environments –, our method requires no model training or fine-tuning, relying solely on general-purpose LLMs via prompt engineering, which makes it lightweight and easily deployable. It includes a built-in clarification mechanism that prompts the user when input is ambiguous or unfeasible, and enables BT modification in response to execution failures. Furthermore, it is designed for deployment in real-world environments such as eldercare, where ease of use and adaptability are critical. We have clarified these benefits in the revised manuscript (see Section 3).

However, it also presents certain challenges and areas for future exploration. The performance of the system is highly dependent on the quality and precision of the LLM's responses. While the LLM excels at generating BTs for well-structured tasks thanks to the well-designed prompt, it may occasionally produce plans that are suboptimal or inconsistent. The integration of an LLM such as ChatGPT for task planning and failure interpretation introduces latency, particularly when the robot operates in environments requiring frequent plan revisions. This issue could result in short delays until the execution of the requested order. In this first approach we used some of the actions the robot can perform, but as tasks become more complex and with the addition of more robot actions, the size and depth of the BTs increase, potentially making them harder to execute and interpret, which would require a redesign of the generating prompt.

6. Conclusions and future work

In this paper, we presented a novel framework for generating and executing Behavior Trees using Large Language Models for robotic task execution. Our approach combines the interpretability and modularity of BTs with the reasoning capabilities of LLMs, enabling robots to perform complex dynamic tasks with a high degree of adaptability. Using a Failure Interpreter module, the system not only responds effectively to task failures, but also dynamically updates BTs to overcome unforeseen obstacles. Experimental validation demonstrated the robustness and flexibility of the proposed method across a variety of scenarios, highlighting its potential for real-world applications in robotics.

Future work will focus on several key areas. First, we aim to integrate multimodal input sources, such as home sensors data, to enhance robot situational awareness. Second, we plan to explore techniques for optimizing the BT generation process while adding more actions to execute more complex tasks and improve scalability. In order to improve the performance of our method, the entire action queue can be embedded within a reactive sequence in a higher-level BT, where emergency-relevant actions are placed alongside their corresponding conditions—forming a teleo-reactive structure. In this configuration, the system can dynamically update the condition, allowing the robot to immediately interrupt its current action and execute the appropriate emergency response. This approach leverages the inherent reactivity of BTs, enabling timely and context-sensitive behavior adaptation. Finally, we plan to install and evaluate the system in real environments, with supervised deployment of the system in multiple real homes of elderly people to verify its real performance.

CRediT authorship contribution statement

Sergio Merino-Fidalgo: Writing – original draft, Validation, Supervision, Software, Methodology, Investigation, Conceptualization. **Celia Sánchez-Girón:** Writing – original draft, Software, Methodology, Investigation. **Eduardo Zalama:** Writing – review & editing, Supervision, Project administration, Conceptualization. **Jaime Gómez-García-Bermejo:** Writing – review & editing, Supervision, Investigation, Conceptualization. **Jaime Duque-Domingo:** Writing – review & editing, Supervision, Methodology, Investigation, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to improve language and readability. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research presented in this paper has received funding from ROSOGAR project PID2021-123020OBI00 funded by MCIN/AEI/10.13039/501100011033 /FEDER, EU, and from the EIAROB project funded by Consejería de Familia de la Junta de Castilla *y* León - Next Generation EU IN./22/M/01.

Data availability

No data was used for the research described in the article.

References

[1] A. Girault, B. Lee, E.A. Lee, Hierarchical finite state machines with multiple concurrency models, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 18 (6) (1999) 742–760, http://dx.doi.org/10.1109/43.766725.

[2] M. Colledanchise, P. Ögren, Behavior Trees in Robotics and AI: An Introduction, CRC Press, 2018, http://dx.doi.org/10.1201/9780429489105.

[3] P. Ögren, C.I. Sprague, Behavior trees in robot control systems, Annu. Rev. Control. Robot. Auton. Syst. 5 (2022) 81–107, http://dx.doi.org/10.1146/ANNUREV-CONTROL-042920-095314/CITE/REFWORKS.

[4] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P.S. Yu, Q. Yang, X. Xie, A survey on evaluation of large language models, ACM Trans. Intell. Syst. Technol. 15 (2024) 39, http://dx.doi.org/10.1145/3641289.

[5] OpenAI, Introducing chatgpt | openai, 2024, https://openai.com/index/chatgpt/. (Accessed 26 December 2024).

[6] J. Deng, Y. Lin, The benefits and challenges of chatgpt: An overview, Front. Comput. Intell. Syst. 2 (2022) 81–83, http://dx.doi.org/10.54097/FCIS.V2I2.4465.

[7] M. Colledanchise, P. Ögren, Behavior trees in robotics and ai: An introduction, Behav. Trees Robot. AI (2017) http://dx.doi.org/10.1201/9780429489105, http://arxiv.org/abs/1709.00084.

[8] J. Bohren, S. Cousins, The SMACH high-level executive [ROS news], IEEE Robot. Autom. Mag. 17 (4) (2010) 18–20, http://dx.doi.org/10.1109/MRA.2010.938836.

[9] A. Koubaa, et al., Robot Operating System (ROS), vol. 1, Springer, 2017.

[10] M. Ben-Ari, F. Mondada, Finite state machines, Elements Robot. (2018) 55–61, http://dx.doi.org/10.1007/978-3-319-62533-1_4.

[11] D. Isla, Handling complexity in the halo2 ai, in: Game Developers Conference, 2005.

[12] M. Iovino, E. Scukins, J. Styrud, P. Ögren, C. Smith, A survey of behavior trees in robotics and ai, Robot. Auton. Syst. 154 (2022) 104096, http://dx.doi.org/10.1016/J.ROBOT.2022.104096.

[13] C. Pezzato, C.H. Corbato, S. Bonhof, M. Wisse, Active inference and behavior trees for reactive action planning and execution in robotics, IEEE Trans. Robot. 39 (2023) 1050–1069, http://dx.doi.org/10.1109/TRO.2022.3226144.

[14] N. Axelsson, G. Skantze, Modelling adaptive presentations in human–robot interaction using behaviour trees, in: SIGDIAL 2019-20th Annual Meeting of the Special Interest Group Discourse Dialogue - Proceedings of the Conference, 2019, pp. 345–352, http://dx.doi.org/10.18653/V1/W19-5940.

[15] C.I. Sprague, P. Ögren, Adding neural network controllers to behavior trees without destroying performance guarantees, in: Proc. of the 2022 IEEE 61st Conference on Decision and Control, CDC, 2022, pp. 3989–3996, http://dx.doi.org/10.1109/CDC51059.2022.9992501.

[16] Z. Wu, Y. Huang, P. Huang, S. Wen, M. Li, J. Wang, Effbt: An efficient behavior tree reactive synthesis and execution framework, in: Proc. of the 2025 IEEE/ACM 47th Int. Conf. on Software Engineering, ICSE, 2025, http://dx.doi.org/10.1109/ICSE55347.2025.00225, 761–761.

[17] M. Albi, Onboard Mission- and Contingency Management Based on Behavior Trees for Unmanned Aerial Vehicles (Master's thesis), Aalto University, 2023, Available: https://aaltodoc.aalto.fi/handle/123456789/124063.

[18] L. Scherf, A. Schmidt, S. Pal, D. Koert, Interactively learning behavior trees from imperfect human demonstrations, Front. Robot. AI 10 (2023) http://dx.doi.org/10.3389/frobt.2023.1152595, art. 1152595.

[19] K.S. Kalyan, A survey of gpt-3 family large language models including chatgpt and gpt-4, Nat. Lang. Process. J. 6 (2024) 100048, http://dx.doi.org/10.1016/j.nlp.2023.100048.

[20] D. Driess, F. Xia, M.S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, P. Florence, Palm-e: An embodied multimodal language model, Proc. Mach. Learn. Res. 202 (2023) 8469–8488.

[21] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, X. Chen, K. Choromanski, T. Ding, D. Driess, A. Dubey, C. Finn, P. Florence, C. Fu, M.G. Arenas, K. Gopalakrishnan, K. Han, K. Hausman, A. Herzog, J. Hsu, B. Ichter, A. Irpan, N. Joshi, R. Julian, D. Kalashnikov, Y. Kuang, I. Leal, L. Lee, T.W.E. Lee, S. Levine, Y. Lu, H. Michalewski, I. Mordatch, K. Pertsch, K. Rao, K. Reymann, M. Ryoo, G. Salazar, P. Sanketi, P. Sermanet, J. Singh, A. Singh, R. Soricut, H. Tran, V. Vanhoucke, Q. Vuong, A. Wahid, S. Welker, P. Wohlhart, J. Wu, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, B. Zitkovich, Rt-2: Vision-language-action models transfer web knowledge to robotic control, Proc. Mach. Learn. Res. 229 (2023).

[22] A. Lykov, D. Tsetserukou, A. Lykov, D. Tsetserukou, Llm-brain: Ai-driven fast generation of robot behaviour tree based on large language model, 2023, http://dx.doi.org/10.48550/ARXIV.2305.19352, arXiv arXiv:2305.19352.

[23] R.A. Izzo, G. Bardaro, M. Matteucci, Btgenbot: behavior tree generation for robotic tasks with lightweight llms, 2024, arXiv.

[24] H. Zhou, Y. Lin, L. Yan, J. Zhu, H. Min, Llm-bt: Performing robotic adaptive tasks based on large language models and behavior trees, in: Proceedings - IEEE International Conference on Robotics and Automation, 2024, pp. 16655–16661, http://dx.doi.org/10.1109/ICRA57147.2024.10610183.

[25] J. Ao, F. Wu, Y. Wu, A. Swikir, S. Haddadin, Llm as bt-planner: leveraging llms for behavior tree generation in robot task planning, 2024, arXiv.

[26] S. Cooper, S. Lemaignan, Towards using behaviour trees for long-term social robot behaviour, in: ACM/IEEE International Conference on Human-Robot Interaction 2022-March, 2022, pp. 737–741, http://dx.doi.org/10.1109/HRI53351.2022.9889662.

[27] C.Y. Kim, C.P. Lee, B. Mutlu, Understanding large-language model (llm)-powered human–robot interaction, in: ACM/IEEE International Conference on Human-Robot Interaction, 2024, pp. 371–380, http://dx.doi.org/10.1145/3610977.3634966.

[28] L. Hanschmann, U. Gnewuch, A. Maedche, Saleshat: A llm-based social robot for human-like sales conversations, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 14524 LNCS, 2024, pp. 61–76, http://dx.doi.org/10.1007/978-3-031-54975-5_4.

[29] Temi, Introducing temi robot v3, 2024, https://www.robotemi.com/product/temi-sales-contact/. (Accessed 03 January 2025).

[30] S. Merino-Fidalgo, E. Zalama, J. Gómez-García-Bermejo, J. Duque-Domingo, R. Gómez, P. Viñas, D. García, H. Urueña, Sistema de monitorización no intrusiva para vivienda de personas mayores, Jornadas Nac. Robótica Bioingeniería 2023: Libr. Actas (2023) 115–121, http://dx.doi.org/10.20868/UPM.BOOK.74896.

[31] H. Zhang, C. Xu, S. Zhang, Inner-iou: more effective intersection over union loss with auxiliary bounding box, 2023, arXiv, https://arxiv.org/abs/2311.02877v4.

[32] C. Sánchez-Girón, M.G. Gómez, J.D. Domingo, J.G. García-Bermejo, E.Z. Casanova, Integración convnext-yolo mediante cvv para detectar caídas en robot social, Jornadas Automática (2024) http://dx.doi.org/10.17979/ja\-cea.2024.45.10788.

[33] J.D. Domingo, R.M. Aparicio, L.M.G. Rodrigo, Cross validation voting for improving cnn classification in grocery products, IEEE Access 10 (2022) 20913–20925.

[34] S. Maldonado-Bascón, C. Iglesias-Iglesias, P. Martín-Martín, S. Lafuente-Arroyo, Fallen people detection capabilities using assistive robot, Electronics 8 (9) (2019) http://dx.doi.org/10.3390/electronics8090915, article 915.

[35] J. Hurtienne, A. Naumann, QUESI – A questionnaire for measuring the subjective consequences of intuitive use, in: Interdisciplinary College 2010. Focus Theme: Play, Act and Learn, Fraunhofer Gesellschaft, Sankt Augustin, 2010, p. 539.

[36] A. Naumann, J. Hurtienne, Benchmarks for intuitive interaction with mobile devices, in: Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction, INTERACT 2010, in: ACM International Conference Proceeding Series, 2010, pp. 401–402, http://dx.doi.org/10.1145/1851600.1851685.

Sergio Merino-Fidalgo received the degree in Industrial Electronic and Automatic Engineering from the University of Valladolid (Spain) in 2021, the M.S. degree in Industrial Engineering from the University of Valladolid in 2023. He is a Ph.D. student in Industrial Engineering in the University of Valladolid and works as a researcher at the Institute for Advanced Production Technologies (ITAP), from the University of Valladolid since 2023. His field of research focuses on task planning in robotics.

Celia Sánchez-Girón received the degree in Industrial Electronic and Automatic Engineering from the University of Valladolid (Spain) in 2021, the degree in Biomedical Engineering from the University of Valladolid in 2022, the M.S. degree in Biomedical Engineering from the University of Valladolid in 2024. She is a Ph.D. student in Industrial Engineering in the University of Valladolid and works as a researcher at the Institute for Advanced Production Technologies (ITAP), from the University of Valladolid since 2023. Her field of research includes computer vision and robotics.

Eduardo Zalama received the Ph. D. Degree in Control Engineering from the University of Valladolid (Spain) in 1994. He is full professor in the school of Industrial Engineering in the University of Valladolid. He is author and co-author of over 100 papers in the field of robotics and computer vision. Nowadays he is the director of Industrial and Digital Systems at Cartif Technological Center, leading a group of 25 research engineers. From the industrial point of view, he has been involved in a wide range of industrial projects for the development of mobile robots, control systems and manufacturing.

Jaime Gómez-García-Bermejo received the B.S. degree in electronics and automatic control engineering from the University of Valladolid (Spain), in 1990, the M.S. degree in image processing from the École Nationale Supérieure des Telecommunications of Paris (France), in 1991; and the Ph.D. degree in industrial engineering from said University, in 1995. He is currently full professor at the Dep. of Systems Engineering and Automatic Control at this University, and leads the Computer Vision Laboratory at CARTIF Tech. Center where he has lead about two hundred research projects and contracts. He is also the coordinator of a Ph.D. Program in his University. His research interests computer vision and robotics.

Jaime Duque-Domingo received the B.S. degree in Computer Engineering from the University of Valladolid (Spain), in 2011, the M.S. degree in Software and IT Systems Engineering from the UNED of Madrid (Spain), in 2014; and the Ph.D. degree in Control and Systems Engineering from said University, in 2018. He has combined his academic career with a professional career of over 20 years working on IT projects in various fields, including projects at the European Commission in Brussels. He is currently an associate professor at the University of Valladolid. His field of research includes artificial intelligence, computer vision and robotics.