

An Environment-Aware Verification Framework for LLM-Generated Robot Control Programs

Zhanshang Nie, Wenbo Wang, Xuanming Liu, Yue Zhang, Zhendong Chen, Zirui Wang, Kai Huang, Shuai Zhao*
School of Computer Science and Engineering, Sun Yat-sen University, China

Abstract—Large language models (LLMs) are increasingly used in robotics to translate natural language instructions into executable control programs via task-specific prompts. However, existing approaches often lack correctness guarantees for LLM-generated programs, leading to compilation errors and runtime failures. While some methods consider a verification mechanism, they typically assume complete prior knowledge of the environment, making them unsuitable for complex environments where such knowledge is unavailable. This paper introduces VeBot, an environment-aware verification framework designed to ensure the correctness of robot control programs generated by LLMs. Specifically, VeBot introduces: (i) an LLM-friendly robot control language (RCL) that facilitates the program generation by abstracting away the complex Python code details, (ii) a compiler that translates LLM-generated RCL programs into a control flow graph (CFG) while verifying the lexical, syntactic, and semantic correctness, and (iii) a runtime verification mechanism that checks the CFG and compiles the verified segments into executable Python code, avoiding collisions or planning failures during execution. We illustrate the VeBot framework using a household scenario, and the evaluation shows that it consistently outperforms existing methods across a range of LLMs and tasks, achieving high success rates even with lightweight LLMs.

I. INTRODUCTION

The emergence of large language models (LLMs) has demonstrated significant potential in advancing embodied intelligence [1]–[3]. Existing methods [4]–[6] leverage LLMs to generate robot control programs from natural language instructions by constructing prompts that incorporate users’ task descriptions and relevant robot action API [7], [8]. These generated programs are subsequently deployed and executed directly on real robots. However, these works fail to guarantee the correctness of the generated programs, which are prone to various types of errors [9], including lexical, syntactic, semantic errors and runtime failures. Fig. 1 presents an example code generated by LLM where a mobile manipulator is instructed to retrieve a television remote control from the table. However, the generated code contains errors: it executes a “navigate_to” command without waiting for SLAM to fully start up, which causes failures due to missing localization data from SLAM; and “backward(-0.5)” uses a negative value, where a positive integer is expected, leading to a compilation error.

In addition, most existing verification methods focus on the LLM-generated code itself without considering robotic execution [10]–[12]. However, verifying LLM-generated code for robots is considerably more challenging, as it involves frequent interactions with the environment [13]. Some recent efforts

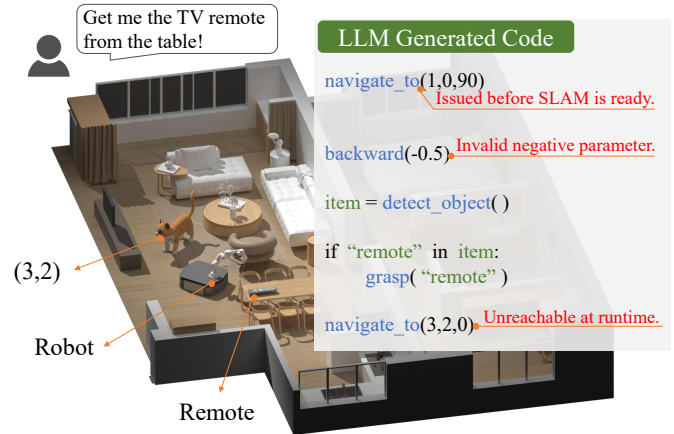


Fig. 1: Typical errors in LLM-generated robotic programs.

have incorporated a verification mechanism for robotics [14], [15]. However, they typically assume complete prior knowledge of the environment. As a result, the runtime correctness of the generated programs cannot be ensured if environment information is incomplete (e.g., limited coverage of LiDAR scans) or the positions of obstacles change over time. In such cases, collisions or planning failures may not be detected. As shown in Fig. 1, the instruction “navigate_to(3,2,0)” is generated based on the initial environment before the execution of the robot. However, the location (3,2) may become unreachable at runtime due to changes in the positions of obstacles, which leads to collisions.

This paper presents an environment-aware verification framework (named VeBot) that ensures the correctness of robotic programs generated by LLMs. To facilitate the program generation, we develop an LLM-friendly robot control language (RCL) that abstracts away Python implementation details. For the generated programs, VeBot introduces two verification mechanisms to ensure both the program-level and runtime correctness. First, we construct a compiler-based verification method that performs lexical, syntactic and semantic analysis while translating the RCL program to a control flow graph (CFG). Then, a runtime verification mechanism is proposed that checks the CFG segments to prevent collisions or planning failures. We evaluate VeBot in a household scenario using both simulation and a real mobile manipulator. Results show that VeBot outperforms existing methods across a range of LLMs and tasks, achieving high success rates even with the lightweight LLMs that can be locally deployed.

*Corresponding author: Shuai Zhao, zhaosh56@mail.sysu.edu.cn.

Example 1:	Example 2:
1 # navigate to (0,0)	1 # conditional structure
2 goto 0, 0, 90;	2 list = get_objects;
3 # grasp the bottle	3 if "bottle" in list :
4 grasp "bottle";	4 grasp "bottle";
5	5 ...
6 approach "table";	6 # loop structure
7 turnleft 90;	7 while x < 2 :
8 forward 3;	8 forward x;
9 set_grip 0.8;	9 ...

Fig. 2: Examples of the RCL program.

II. PRELIMINARIES

Our problem setting focuses on the process in which a user provides a natural language task with the goal of instructing the robot to perform desired actions in the physical world. The LLM interprets the user’s intention and translates it into a program that can be executed on the robot platform. Within this setting, we first present the target robot and scenario in this work. Then, we introduce the Robot Control Language, which facilitates the LLM-based program generation by abstracting low-level implementation details. Finally, we explain the verification objectives: ensuring program-level and runtime correctness of LLM-generated programs.

A. Robot Model and Target Scenario

The robot employed in this work is a mobile manipulator consisting of a four-wheeled base and a robotic arm. The robot is equipped with a 360° rotating 2D LiDAR and an RGB-D camera to perceive the surrounding environment. It also incorporates a library of algorithms supporting essential capabilities, including movement, SLAM, navigation, object identification, and grasping. These functions are implemented on top of the ROS-1 framework [16] and are executable through Python code, serving as the foundation for the methods introduced in the following sections.

We illustrate the proposed VeBot framework using a household scenario, which provides a typical setting with diverse spatial layouts and a wide range of robotic tasks. The household environment often consists of multiple rooms connected by narrow passages and doorways. It also contains a variety of obstacles, such as furniture or household appliances, whose positions may change over time. In such environments, the robot may be asked to perform daily tasks, including basic movements, searching for items, and object manipulation (e.g., grasping objects or operating devices). These features make the household a representative domain for our approach [17].

B. Robot Control Language

Unlike previous work that uses LLMs to generate Python code directly from natural language, in this work, we present a high-level Robot Control Language to reduce the complexity of program generation for LLMs. RCL abstracts away low-level robot control details, including actuation control, raw sensor data processing for perception and localization, and ROS communication mechanisms. This effectively facilitates the LLM-based code generation, especially for complex tasks.

TABLE I: Example grammar constructs in RCL.

Robot Capability	Keyword	Syntax
Movement	forward/backward	forward/backward <number>;
Rotation	turnleft/turnright	turnleft/turnright <degrees>;
Navigate to a location	goto	goto <x>,<y>,<yaw>;
Navigate near an object	approach	approach "<object>;"
Pick up a specified object	grasp	grasp "<object>;"
Set the gripper to a target pose	set_end	set_end <x>,<y>,<z>;
Set gripper opening width	set_grip	set_grip <0.0-0.8>;
Say the specified message to user	say	say "<message>;"
Get object list from detection	get_objects	list = get_objects;
Get position of an object	get_position	var = get_position("<object>;");
Get voice input from user	query_user	input = query_user;

The RCL programs are primarily composed of command-based instructions, each corresponding to a fully encapsulated robot capability. A command-based instruction resembles a function call, consisting of an action name followed by a list of parameters. Table I summarizes representative robot actions supported by RCL, along with their corresponding instruction keywords and syntax patterns. Example 1 in Fig. 2 presents an example program written in RCL instructions.

In addition, the RCL encapsulates perception functions that return detected object lists and the positions of target objects, which can then be used by subsequent instructions. The corresponding keywords and syntax patterns for these perception instructions are given in Table I. Moreover, RCL supports conditional and loop structures that follow the Python grammar, thereby allowing the generated programs to make decisions based on environmental conditions. An example RCL program with conditional and loop structures is shown in Example 2 of Fig. 2. Other constructs, such as identifiers, assignment statements and arithmetic (or boolean) expressions are also consistent with Python grammar.

C. Aspects of Program Correctness Verification

For clarity of the following discussion, we represent the LLM-generated RCL program as a sequence of instructions, where each instruction is further represented as a sequence of characters. The key challenge addressed in this paper is to ensure two aspects of correctness in the RCL program:

Program-Level Correctness: As the example discussed in Section I, programs generated by LLMs may contain errors that cannot pass the compiler checks [18]. We ensure program-level correctness by focusing on the following aspects: lexical, syntactic and semantic correctness. First, the lexical specification of the language defines how character sequences in an instruction are grouped into valid lexical token types (e.g., keywords such as "if" or "while", identifiers, numeric constants, and punctuation symbols). Based on this specification, a lexical analyzer scans each instruction and segments it into a sequence of tokens. A program is considered lexically correct if every subsequence of characters in each instruction can be matched to a valid token. Second, the syntax specification of the language defines how multiple tokens form valid syntactic constructs. Based on this specification, a syntax

TABLE II: Examples of the program-level errors.

Correctness	Typical Error	Unexpected Form	Explanation
Lexical	Invalid identifier	<code>lvar = 10;</code>	Identifier cannot start with digit
	Number format error	<code>x = 123.23.45;</code>	Multiple dots in number
	Illegal character	<code>\$name = "Alice";</code>	Illegal character "\$"
Syntactic	Invalid argument	<code>forward 10 , 5;</code>	Wrong number of parameters
	Missing colon	<code>if (x > 0) </code>	Colon required after condition
	Missing semicolon	<code>x = 10 </code>	Semicolon missing
Semantic	Undeclared variable	<code>x = y + 1;</code>	Variable not declared
	Incompatible argument	<code>x = (5,2,1); grasp x;</code>	Type mismatch
	Missing prerequisite	<code>goto A;</code>	"slam" required before "goto"

analyzer analyzes the token sequence of each instruction and organizes it into an abstract syntax tree (AST) that captures its hierarchical structure. A program is considered syntactically correct if all tokens can be derived into valid constructs. Finally, each instruction must meet its corresponding semantic constraints: prerequisite instructions must be executed in order; data dependencies must be resolved (e.g., variables defined before use); and instruction arguments must match the expected types while staying within valid ranges. Table II illustrates representative lexical, syntactic and semantic errors.

Runtime Correctness: In the household scenario, LLMs must consider the complex layout of multiple rooms and the presence of obstacles when generating movement and navigation commands to avoid collisions during execution. Reasoning in such a complex environment is challenging for LLMs, which is particularly difficult for lightweight models. This increases the risk of collisions. Therefore, in addition to program-level correctness, we also consider runtime correctness. Specifically, each instruction must satisfy its corresponding environment precondition (e.g., necessary constraints on surrounding objects and SLAM information). For example, the grasp instruction requires the target object to be within the arm’s workspace (see Fig. 3(a)). The forward instruction requires the commanded distance not to exceed the available free space to avoid a collision (Fig. 3(b)). In addition, the navigation command requires the target position (0,0) to be reachable (Fig. 3(c)). Section IV-B provides a detailed definition of the environment preconditions.

III. OVERVIEW OF VE BOT AND CODE GENERATION

This section describes the overview of the VeBot framework. As illustrated in Fig. 4, the VeBot consists of three stages: code generation, program-level verification and runtime verification. In the first stage, the LLM generates an RCL program from the natural language instruction. We construct a detailed prompt to guide the generation of RCL programs (see Fig. 4(a)), which includes (i) the task in natural language, (ii) the grammar specification of RCL and a role definition for the LLM and (iii) a textual description of the surrounding environment (e.g., robot’s position, objects detected at the beginning of execution, and an approximate size of the map).

For the grammar specification in the prompt, we present the lexical rules of RCL, which include the keywords, symbols,

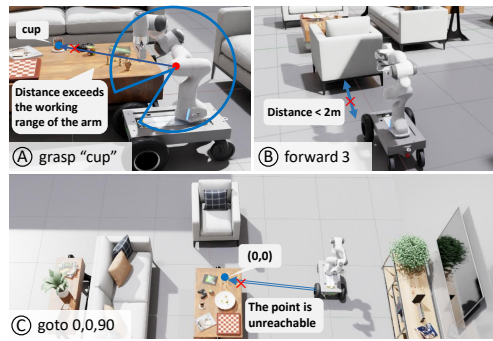


Fig. 3: Examples of runtime correctness.

integers, decimals, and the identifiers. To describe syntactic structures, we present each construct of RCL using natural language descriptions with illustrative examples. For instance, we specify the exact number of parameters required for each function, illustrate the syntax of conditional and loop statements, and demonstrate the form of assignment instructions. These descriptions help the LLM understand and generate RCL programs that follow the specified grammar.

The RCL programs generated from Fig. 4(a) are then verified through the lexical, syntactic, and semantic analysis in the second stage (i.e., the program-level verification in Fig. 4(b)). As shown in Fig. 4, the program fails the syntactic analysis because the command “goto” requires three parameters in the form of $\langle x, y, yaw \rangle$, yet only two parameters are provided. The syntax analyzer then generates the diagnostic information and sends it back to the LLM for correction. While the figure shows a simplified example, VeBot provides detailed, LLM-friendly feedback specifying the exact line number, error cause, and erroneous code segment. The LLM then produces a revised program for verification again. After the verification, the corrected RCL program is translated into a control flow graph (CFG), which is passed to the next stage in Fig. 4(c).

In the third stage, VeBot verifies the program correctness in the environment by checking for potential collisions. Since obstacles and complex room structures can cause incomplete SLAM maps while the obstacle positions may change over time (e.g., furniture rearrangement), we assume only partial environment information is collected at the start of execution. This makes it difficult to perform the complete runtime verification in a single pass. To address this challenge, we traverse the segments of the CFG to identify potential collisions. The runtime verifier starts from the entry node of the CFG and checks each instruction against its runtime preconditions (see Section IV-B). Verified segments are compiled into executable code and dispatched to the robot. After execution, the runtime data is updated with newly perceived environment information, and verification repeats for the remaining instructions.

For example, at time t_1 , the verifier confirms the segment from the entry node to the instruction “slam”. It suspends at “goto 1,4,90” and executes the verified code, and then navigates to a position near (1, 4) (see Section IV-B for further explanation). Once the execution ends, the verifier receives updated runtime data and resumes traversal. This process

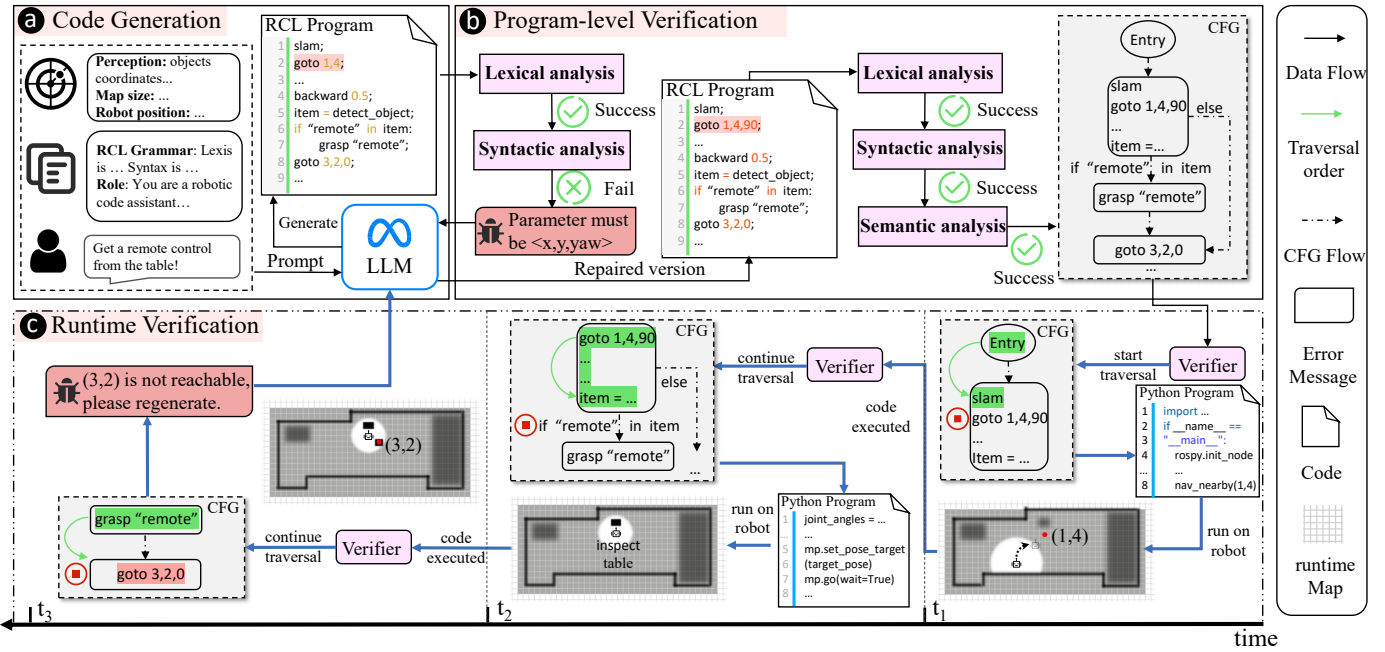


Fig. 4: The overall structure and workflow of the VeBot framework.

repeats until time t_3 , where an error is found. The verifier then reports the diagnostic information to the LLM, which corrects the program segment. This process continues until the program passes all checks or the maximum attempt number is reached.

IV. PROGRAM-LEVEL AND RUNTIME VERIFICATION

A. Program-level Verification

Lexical and Syntactic Analysis. The VeBot first performs lexical analysis using the lexical specification introduced in Section II-C, which is implemented by a collection of regular expressions [19]. During this step, the lexical analyzer scans the generated program and applies the regular expressions to each instruction. The resulting token sequence then serves as the input for syntactic analysis. In this phase, the syntax analyzer loads a predefined grammar specification (written in an ANTLR-like format) and translates it into a set of LL(1) grammar productions. Based on these productions, a predictive parsing table is constructed and used to parse the instruction sequence into an abstract syntax tree (AST) [20], [21].

Semantic Analysis. Once the AST is generated, its root is a virtual node whose child subtrees correspond to individual syntactic constructs. By traversing these subtrees from left to right, we can recover the program order of instructions and verify semantic correctness, which includes: (i) prerequisite instructions are satisfied (e.g., a “goto” instruction is only valid after a “slam” instruction); (ii) variables are defined before use; and (iii) instruction parameters must match the expected types within valid ranges. Once the semantic check is completed, the AST is transformed into a CFG for runtime verification.

Error Feedback. When errors are detected during lexical, syntactic, or semantic analysis, the corresponding analyzer generates feedback that records the error type, code fragment, line number, and the cause. In lexical analysis, the fragment

is a character subsequence that cannot be mapped to a valid token. In syntax analysis, it is a sequence of tokens that fails to match any grammar production. For semantic analysis, the fragment is the instruction that violates the preconditions. This feedback provides the LLM with precise diagnostics to guide correction. To reduce iterations, instead of stopping at the first error, the analyzer skips the fragment and continues checking, allowing multiple errors to be reported in a single pass.

B. Runtime Verification

We now explain the working mechanism of the runtime verification, as shown in Alg. 1. The algorithm takes the control flow graph G and the initial environment information as the input. The environment information includes the robot’s current state (position, orientation, and gripper status) and the perceived environment (the occupancy grid map from SLAM and detected objects). The algorithm produces two outputs: an ordered list L_v containing the successfully verified instructions and a message E of the error instruction with the cause.

The algorithm first stores the environment information as the runtime data D and sets N (the currently examined node) to the entry node N_{entry} in the CFG. It also initializes both the verified instruction list L_v and the runtime error messages E . Within each node N , the verifier repeatedly calls `getNextInstruction(N)` (line 3) to fetch the next instruction I . For each instruction I , the verifier invokes `checks(D, I)` (line 5) to evaluate whether the current runtime data D satisfies the preconditions of I , which enforces that (i) no collision will occur for motion commands (such as “forward” or “goto”) and gripper pose setting commands (such as “set_end” or “set_grip”) and (ii) the grasp instructions have a valid target object located within the robot’s workspace and the required gripper pose is kinematically feasible.

Algorithm 1: Process of the runtime verification.

```

1  $N \leftarrow N_{entry}, L_v \leftarrow [], E \leftarrow [];$ 
2 while  $N \neq N_{exit}$  do
3    $I \leftarrow getNextInstruction(N);$ 
4   if  $I \neq \text{None}$  then
5      $\Phi_{ver}, \Phi_{new}, E \leftarrow check(D, I);$ 
6     if  $\Phi_{ver} = \text{true}$  then
7        $L_v \leftarrow append(L_v, I);$ 
8        $D \leftarrow update(D, I);$ 
9     else
10      if  $\Phi_{new} = \text{true}$  then
11         $L_v \leftarrow append(L_v, I^*);$ 
12      end
13      break;
14    end
15  else
16     $N \leftarrow getNextNode(G, N);$ 
17    if  $N = \text{None}$  then
18      break;
19    end
20  end
21 end
22 return  $L_v, E;$ 

```

The function `check()` returns two booleans (Φ_{ver} and Φ_{new}) and an error message E . Φ_{ver} indicates whether the instruction passes runtime verification, and Φ_{new} shows whether its execution would move the robot into an unknown map region. Unknown regions are cells in the SLAM occupancy grid labeled with -1, meaning their state has not been observed. The verifier handles the result as follows. If $\Phi_{ver} = \text{true}$ (line 6), the instruction I is appended to the verified instruction list L_v , and the runtime data D is updated to reflect the instruction’s effects on the robot state or the environment (lines 7-8). Otherwise (*i.e.*, $\Phi_{ver} = \text{false}$ at line 9), the traversal terminates and two cases are further distinguished according to the value of Φ_{new} . First, if $\Phi_{new} = \text{true}$ (line 10), it is impossible to determine the correctness of the instruction given the current runtime data. For example, the target position of “goto 1,4,90” in Fig. 4 lies in an unknown area, making it impossible to determine whether the position is reachable. In this case, the instruction is not appended to L_v ; instead, an exploration instruction I^* is added (line 11), which uses the A* algorithm to find the nearest reachable point within the known map. After reaching this point, the map is updated and the verifier resumes verification. This enables runtime checks without complete prior knowledge of the environment. Second, if $\Phi_{new} = \text{false}$, the instruction can cause a collision. In this case, the algorithm terminates (line 13) with E returned to guide the LLM in correcting the subsequent instructions.

After all instructions in N have been checked, the verifier determines the next node by invoking `getNextNode(G, N)` (line 16). If the result is `None` (line 17), it means that the next node cannot be determined at the time point. For example, if the variable `list` in the instruction `list = get_operable_objs` is unknown before execution, the subsequent condition `if “bottle” in list` becomes undetermined. In this case, the verifier suspends at the current

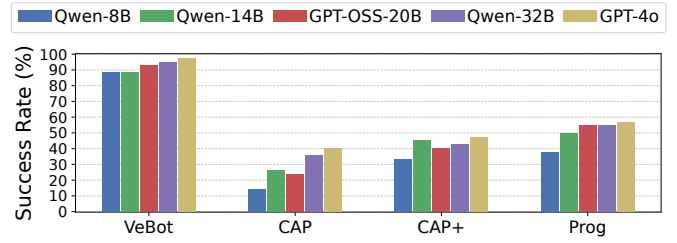


Fig. 5: Success rate comparison under different LLMs.

node and waits for the robot to execute the verified code, thereby updating the required values. This design allows verification to proceed without the need for complete knowledge of the environment and its objects. Finally, the algorithm terminates when the returned node is the exit node N_{exit} .

V. EVALUATION

This section evaluates the constructed VeBot framework in terms of (i) the overall success rate in generating programs across different LLMs, (ii) the number of iterations required (if applicable) for generating correct programs, and (iii) performance in a real-world case study.

A. Experimental Setup

Simulation Platform. All experiments are conducted in Isaac Sim. We deploy a differential-drive Segway RMP Lite 220 mobile base equipped with a 7-DoF Franka Emika Panda arm. The perception module relies on Isaac Sim’s built-in sensor interfaces. The system is implemented on ROS-1, and all RCL functions are encapsulations of existing algorithms. We evaluate VeBot using both locally deployable and API-based LLMs. The locally deployed models include Qwen-8B, Qwen-14B, and GPT-OSS-20B, while the API-based models include Qwen-32B and GPT-4o. This allows us to assess performance across LLMs of different scales.

Tasks. The evaluation is performed under five categories of tasks: (i) object manipulation (e.g., grasp and place a cube on the tabletop), (ii) exploration and navigation (e.g., search for a trash bin in the house), (iii) mobile manipulation (e.g., go to the bedroom and get a bottle), (iv) human–robot interaction, and (v) ambiguous instruction tasks. The last category corresponds to cases where the user provides vague instructions (e.g., “I am thirsty”), requiring the LLM to interpret and complete the task. Each category consists of seven tasks, yielding a total of 35 tasks for evaluation.

B. Overall Performance Comparison under Varied LLMs

We compare VeBot against three methods that construct prompts for guiding LLMs to generate Python-based robot control programs. The Code as Policies (CAP) [4] directly executes the LLM-generated Python code without any program-level or runtime verification. A variant of CAP (denoted as CAP+) augments CAP with feedback-based corrections by returning raw Python exceptions to guide the LLM. The third method, ProgPrompt (Prog) [6], utilizes raw Python exceptions for feedback, but further incorporates assertion

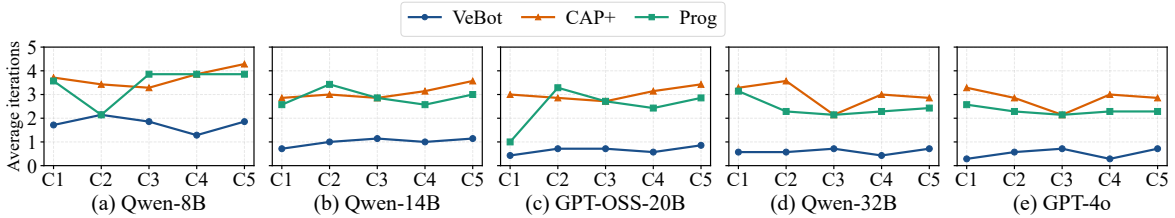


Fig. 6: Comparison of average iterations for VeBot, CAP+, and Prog on five LLMs (*C1 to C5* denote the five task categories).

checks generated by the LLM. These assertions query the robot’s perception interfaces and raise runtime errors, which are returned as feedback to the LLM.

We evaluate the competing methods using the success rate of the 35 tasks, where a task is considered successful if the LLM-generated code executes without program-level errors or runtime failures. For methods without verification (CAP), success requires correct execution on the first attempt. For feedback-based methods (CAP+, Prog, and VeBot), a task succeeds if it can be repaired within at most five iterations. An iteration is defined as one pass in which the system detects an error in the program, provides feedback to the LLM, and updates the program accordingly. The success rate is computed as the proportion of successful tasks. All methods are evaluated with the same task description prompts.

Fig. 5 presents the success rates of the competing methods on five LLMs. From the results, the VeBot consistently outperforms other methods, which achieves over 88% success across all LLMs while others remain below 60% (57.14% for Prog under GPT-4o). These results show that our method effectively reduces errors during program generation. Compared with CAP, both CAP+ and Prog do improve the success rate, but the gains remain limited compared to VeBot. This is because the CAP+ relies on raw Python error messages that LLMs struggle to interpret while lacking runtime verification, whereas the Prog depends on LLM-generated assertions that are often not comprehensive and fail to insert checks for critical instructions. In contrast, our method maintains a high success rate even with lightweight models (such as Qwen-8B) due to the LLM-friendly feedback and the runtime checks.

C. Feedback Iteration Efficiency

In addition to overall success rate comparison, we further evaluate the efficiency of error correction on different LLMs, as shown in Fig. 6. We compare methods that support feedback-based corrections, *i.e.*, CAP+, Prog and VeBot. For each task category (see Section V-A), we report the average number of iterations required until the generated program executes successfully. The tasks that fail to generate correct programs are recorded as five iterations.

As shown in the figure, VeBot consumes much fewer iterations than CAP+ and Prog across all task categories and LLMs. With small models such as Qwen-8B, our method converges within 1–3 iterations, whereas other methods often exceed 3 iterations. For the CAP+ and Prog with larger models, we observe that increasing the model size does not effectively reduce the iteration count due to the lack of an

TABLE III: Success rate and average number of iterations of 15 real-world tasks.

Model	CAP		CAP+		Prog		VeBot	
	Succ. (%)	Iter.	Succ. (%)	Iter.	Succ. (%)	Iter.	Succ. (%)	Iter.
Qwen-8B	20.00	–	33.33	4.00	33.33	3.40	80.00	1.87
Qwen-14B	26.67	–	46.67	2.87	46.67	2.93	86.67	1.60
GPT-OSS	33.33	–	40.00	3.20	46.67	3.00	86.67	1.40
Qwen-32B	33.33	–	40.00	3.20	53.33	2.67	86.67	1.33
GPT-4o	40.00	–	46.67	2.87	53.33	2.73	93.33	1.20

effective feedback-based correction process. In contrast, our framework sustains a low iteration count, with an average number of iterations of 1.7 with Qwen-8B and remains below 1 with Qwen-32B and larger models. This is expected as the LLM-friendly feedback mechanism can pinpoint the wrong instruction with an informative error message.

D. Real-World Case Study

To further evaluate the applicability of VeBot, we deploy it on Rosmaster X3 Plus. The robot is equipped with an omnidirectional mobile base and a 5-DoF robotic arm. The control and verification framework runs on the Jetson Orin Nano board with ROS-1, and is supported with the same robot function as with the simulation. In total, 15 tasks that cover the five categories described in Sec. V-A are evaluated, where the success rate and the average number of iterations required are reported in Table. III. All tasks are carried out in a realistic household environment with a single room.

As shown in Table. III, the VeBot achieves a higher success rate across all LLMs, which has a success rate of at least 80% on Qwen-8B, where the failed tasks are of high complexity that are hard to correct within five iterations (*e.g.*, object searching in a complex environment). In contrast, the success rates of the competing methods remain below 60%. This observation is consistent with the simulation results. In addition, VeBot achieves so by consuming fewer iterations (*i.e.*, at most 1.87 on average) than the CAP+ and Prog, which again justifies the effectiveness and efficiency of the proposed VeBot framework.

VI. CONCLUSION

This paper presents VeBot, an environment-aware verification framework for LLM-generated robot control programs. Results show that VeBot consistently outperforms existing methods, achieving higher success rates with fewer iterations, even on lightweight models. Future work will extend VeBot to dynamic environments that require adaptive re-verification when unexpected changes occur.

REFERENCES

- [1] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter, "Inner monologue: Embodied reasoning through planning with language models," 2022. [Online]. Available: <https://arxiv.org/abs/2207.05608>
- [2] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei, "Voxposer: Composable 3d value maps for robotic manipulation with language models," in *Proceedings of The 7th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, J. Tan, M. Toussaint, and K. Darvish, Eds., vol. 229. PMLR, 06–09 Nov 2023, pp. 540–562.
- [3] B. Ichter, A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," in *Proceedings of The 6th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, K. Liu, D. Kulic, and J. Ichnowski, Eds., vol. 205. PMLR, 14–18 Dec 2023, pp. 287–318.
- [4] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, "Code as policies: Language model programs for embodied control," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 9493–9500.
- [5] M. G. Arenas, T. Xiao, S. Singh, V. Jain, A. Ren, Q. Vuong, J. Varley, A. Herzog, I. Leal, S. Kirmani *et al.*, "How to prompt your robot: A promptbook for manipulation skills with code as policies," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 4340–4348.
- [6] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "Progprompt: Generating situated robot task plans using large language models," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 11 523–11 530.
- [7] K. Rana, J. Haviland, S. Garg, J. Abou-Chakra, I. Reid, and N. Suenderhauf, "Sayplan: Grounding large language models using 3d scene graphs for scalable task planning," *arXiv preprint arXiv:2307.06135*, 2023.
- [8] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 9118–9147.
- [9] S. H. Vemprala, R. Bonatti, A. Bucker, and A. Kapoor, "Chatgpt for robotics: Design principles and model abilities," *IEEE Access*, vol. 12, pp. 55 682–55 696, 2024.
- [10] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "Llmdfa: analyzing dataflow in code with large language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 131 545–131 574, 2024.
- [11] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=Q3YaCghZNt>
- [12] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 172–184.
- [13] F. Joublin, A. Ceravola, P. Smirnov, F. Ocker, J. Deigmoeller, A. Belardinelli, C. Wang, S. Hasler, D. Tanneberg, and M. Gienger, "Copal: Corrective planning of robot actions with large language models," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 8664–8670.
- [14] Y. Li, L. Wang, S. Piao, B.-H. Yang, Z. Li, W. Zeng, and F. Tsung, "Mccoder: Streamlining motion control with llm-assisted code generation and rigorous verification," 2025. [Online]. Available: <https://arxiv.org/abs/2410.15154>
- [15] R. A. Izzo, G. Bardaro, and M. Matteucci, "Btgenbot: Behavior tree generation for robotic tasks with lightweight llms," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024, pp. 9684–9690.
- [16] M. Quigley, "Ros: an open-source robot operating system," in *IEEE International Conference on Robotics and Automation*, 2009. [Online]. Available: <https://api.semanticscholar.org/CorpusID:6324125>
- [17] H. Sahin and L. Guvenc, "Household robotics: autonomous devices for vacuuming and lawn mowing [applications of control]," *IEEE Control Systems Magazine*, vol. 27, no. 2, pp. 20–96, 2007.
- [18] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.
- [19] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston: Addison-Wesley, 2006.
- [20] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2006.
- [21] T. J. Parr and R. W. Quong, "Antr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.