Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

IFAC Papers Online
CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 59-26 (2025) 43–48

LLM-Enhanced Symbolic Control for Safety-Critical Applications

Amir Bayat * **Alessandro Abate** ** **Necmiye Ozay** *** **Raphaël M. Jungers.** *

* *ICTEAM Institute, UCLouvain, Louvain-la-Neuve, Belgium (e-mail: FirstName.LastName@ uclouvain.be)*
** *University of Oxford, Oxford, UK, (e-mail: Alessandro.Abate@cs.ox.ac.uk)*
*** *University of Michigan, Ann Arbor, Michigan, USA (e-mail: necmiye@umich.edu)*

Abstract: Motivated by Smart Manufacturing and Industry 4.0, we introduce a framework for synthesizing Abstraction-Based Controller Design (ABCD) for reach-avoid problems from Natural Language (NL) specifications using Large Language Models (LLMs). A Code Agent interprets an NL description of the control problem and translates it into a formal language interpretable by state-of-the-art symbolic control software, while a Checker Agent verifies the correctness of the generated code and enhances safety by identifying specification mismatches. Evaluations show that the proposed approach increases the success rate of solving reach-avoid problems from 20%, when using LLMs directly, to 80%, while also enhancing robustness to linguistic variability across test cases. The proposed approach lowers the barrier to formal control synthesis by enabling intuitive, NL-based task definition while maintaining safety guarantees through automated validation.

Copyright © 2025 The Authors. This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)

Keywords: Symbolic Control. Abstraction-Based Control, Safety-Critical applications, Large Language Models, Agentic AI, Intelligent manufacturing systems

1. INTRODUCTION

Many real-world control applications, such as autonomous vehicles, medical devices, and industrial process control systems, are safety-critical and demand stringent guarantees. In such systems, violations of operational constraints can lead to catastrophic or irreversible outcomes. To ensure both safety and performance, control strategies must be designed with formal guarantees. However, the complexity of cyber-physical systems (CPSs)—which often include nonlinear, hybrid, or high-dimensional dynamics—makes control synthesis a challenging task.

A promising solution is Abstraction-Based Controller Design (ABCD), also known as symbolic control (Tabuada, 2009; Zamani et al., 2014). In this approach, the original dynamical system is approximated by a finite, discrete abstraction through discretization of the state and input spaces, which makes the problem more tractable for algorithmic controller design. A controller is synthesized at this abstract level and then concretized to operate on the original system while preserving correctness. ABCD provides correct-by-design guarantees and has shown success in handling complex specifications, such as those expressed in temporal logic. The ability to provide formal guarantees makes ABCD especially suitable for safety-critical applications, which is the primary reason for selecting this approach in our work. However, implementing these methods is non-trivial: it typically requires deep domain expertise, significant effort, and careful tuning.

In classical symbolic control, it is the engineer's task to interpret diverse constraints, such as physical limitations, budget requirements, and sustainability goals, and convert them into a set of formal specifications understandable by the machine. This translation process can be extremely time-consuming, or even infeasible, especially for non-expert users or in systems with evolving requirements. Among various safety-critical applications, smart manufacturing and Industry 4.0 highlight the need for intelligent, flexible interfaces that accommodate decentralized operation and operators with varying expertise and terminology.

Recent advances in optimization, learning, and data-driven methods have renewed interest in ABCD approaches and have led to significant advancements in the field (Zamani et al., 2014; Nilsson et al., 2017; Haesaert et al., 2017; Banse et al., 2023; Badings et al., 2023; Calbert et al., 2024b). Notably, `Dionysos` [1] (Calbert et al., 2024a), a modular and open-source software tool, has been developed to facilitate control synthesis using symbolic methods. `Dionysos` integrates modern optimization techniques with smarter abstraction strategies, making it feasible to address more complex control problems that were previously out of reach.

Large Language Models (LLMs)—a class of machine learning models developed for natural language processing—have surged in popularity due to their intuitive inter-

[1] `https://github.com/dionysos-dev/Dionysos.jl`

2405-8963 Copyright © 2025 The Authors. This is an open access article under the CC BY-NC-ND license.
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2025.12.008

faces and impressive capabilities in intent detection (Arora et al., 2024; Njah et al., 2025), code generation (Jiang et al., 2024), and general knowledge synthesis. Despite these strengths, LLMs face significant challenges when it comes to reasoning about physical and dynamical systems, largely because they lack grounding in real-world dynamics (Ahn et al., 2022). As a result, their direct application to engineering problem-solving remains limited. However, performance improves markedly when complex tasks are decomposed into simpler subtasks (Khot et al., 2022). This insight has inspired frameworks like ControlAgent (Guo et al., 2024), which coordinates multiple few-shot-trained LLM agents to synthesize PID controllers from natural language (NL) specifications. While promising, such methods are currently restricted to linear systems and classical control.

Other lines of work have used LLMs to translate NL into formal specifications such as temporal logic (Chen et al., 2024; Pan et al., 2023), but to our knowledge, none have explored direct integration with ABCD pipelines or aimed to support full controller synthesis for nonlinear or hybrid systems. Given the strong performance of LLMs in translating NL to logic, integrating them into ABCD via formal specification translation remains a promising direction for future work. In this paper, however, we focus on direct code generation as an initial step toward connecting LLMs with symbolic control.

In this work, we take a first step toward automated, end-to-end ABCD guided by NL. We leverage LLMs as few-shot learners to extract the specifications of reach-avoid problems from NL descriptions and automatically generate code based on these specifications. The generated code fills a modular component within a larger framework that solves reach-avoid problems symbolically. To evaluate the effectiveness and robustness of our method, we introduce a benchmark dataset composed of diverse environments with varying obstacle configurations and target regions. This work can also be seen as an early step toward meeting emerging needs in smart manufacturing by exploring how LLMs can help interpret NL specifications and support the synthesis of safe planners/controllers.

2. PRELIMINARY

This section provides the necessary background by briefly introducing the `Dionysos` toolbox and the classical abstraction method it uses for controller synthesis.

2.1 Dionysos

As previously mentioned, `Dionysos` is a state-of-the-art modular package designed to solve optimal control problems for complex dynamical systems using ABCD methods. Given the system dynamics and problem specifications, it can automatically construct the abstraction, synthesize an abstract controller, and concretize it to operate on the concrete system. In a reach-avoid problem, these specifications involve the state space bounds, initial state, target states and constraints.

We selected `Dionysos` because it is open-source and supports a wide range of abstraction techniques, from classical methods to more advanced strategies such as ellipsoidal, hierarchical, lazy, and lazy ellipsoidal abstractions, while integrating smoothly with Julia-based workflows. Its modular architecture also facilitates integration with external tools. In this work, we focus on the *Uniform Grid Abstraction* method, which is a classical approach, though the framework can be readily extended to other built-in abstractions provided by `Dionysos`. A concise overview of this approach is provided in the following subsection.

2.2 Classical Abstraction-Based Controller Design

In this section, we provide a concise overview of classical ABCD, which serves as the foundation for the control synthesis techniques implemented in `Dionysos`. The ABCD methodology proceeds through the following main stages:

Defining the System Model We consider a continuous-time control system described by:

$$\dot{x}(t) = f(x(t), u(t)), \tag{1}$$

where $x(t) \in \mathcal{X} \subseteq \mathbb{R}^n$ is the system state and $u(t) \in \mathcal{U} \subseteq \mathbb{R}^m$ is the control input. The function $f : \mathcal{X} \times \mathcal{U} \to \mathbb{R}^n$ defines the system dynamics.

A *specification* $\Sigma \subseteq \mathcal{X}^{\mathbb{R}_{\geq 0}}$ defines a set of admissible trajectories. The control problem is to design a feedback controller $u : \mathcal{X} \to \mathcal{U}$ such that the closed-loop trajectories satisfy Σ.

This work focuses on *reach-avoid specifications* of the form:

$$\Sigma^{\text{reach}} = \big\{x(\cdot) \in \mathcal{X}^{\mathbb{R}_{\geq 0}} \,\big|\, x(0) \in \mathcal{X}_{\text{in}} \Rightarrow \big(\exists t_f \in \mathbb{R}_{\geq 0} : x(t_f) \in \mathcal{T}\big) \wedge \big(\forall t \in [0, t_f), x(t) \notin \mathcal{O}\big)\big\}. \tag{2}$$

where $\mathcal{X}_{\text{in}}$ is the initial state, $\mathcal{T}$ is the target set, and $\mathcal{O}$ is the obstacle set. This specification enforces that, starting from an initial state in the set, the system must eventually reach $\mathcal{T}$ at some finite time t_f, while strictly avoiding $\mathcal{O}$. The goal is to design a controller $\mathcal{C} : \mathcal{X} \to 2^{\mathcal{U}}$ such that the trajectories of the closed-loop system satisfy Σ^{reach}.

Symbolic Abstraction In ABCD, in order to enable formal reasoning and automatic controller synthesis, a *finite-state abstraction* of the continuous system is constructed. This abstraction is achieved through two main steps: i) **Discretization**: Partitioning the continuous state and input spaces into finitely many regions. ii) **Transition relation establishment**: Establishing the relation $\mathcal{R}$ between the states of the original and abstract systems. This process results in a *symbolic model* of the original system.

To discretize the continuous state space $\mathcal{X} \subseteq \mathbb{R}^n$, a uniform grid is constructed using a quantization parameter $\eta \in \mathbb{R}{>}\, 0^n$. The grid points form a lattice given by

$$[\eta_x \mathbb{Z}^n] = \{c \in \mathbb{R}^n \mid \exists k \in \mathbb{Z}^n, \forall i \in \{1, \dots, n\}, c_i = k_i \eta_{x,i}\}. \tag{3}$$

The *abstract state space* $\mathcal{X}_d$ is then defined as the set of lattice points that lie within the original continuous space:

$$\mathcal{X}_d = [\eta_x \mathbb{Z}^n] \cap \mathcal{X}. \tag{4}$$

A **relation** $R_x \subseteq \mathcal{X} \times \mathcal{X}_d$ is introduced to associate each continuous state with its corresponding abstract state. In this work, we adopt a common instantiation where the abstract state space consists of the center points of uniform, axis-aligned grid cells, and define:

$$R_x = \{(x, x_d) \mid \|x - x_d\|_\infty \leq \eta_x/2, x_d \in \mathcal{X}_d\}. \tag{5}$$

The same quantization procedure is applied to the input space $\mathcal{U} \subseteq \mathbb{R}^m$, resulting in a discrete input set $\mathcal{U}_d = [\eta_u \mathbb{Z}^m] \cap \mathcal{U}$ using input grid resolution $\eta_u \in \mathbb{R}^m_{>0}$ with the relation R_u.

Control Synthesis Once the finite abstraction is constructed, the system is represented as a finite transition system (FTS) $\mathcal{S}_d = (\mathcal{X}_d, \mathcal{U}_d, \Delta)$, where $\mathcal{X}_d$ is the discrete (abstract) state space, $\mathcal{U}_d$ is the discrete input set, $\Delta \subseteq \mathcal{X}_d \times \mathcal{U}_d \times \mathcal{X}_d$ is the transition relation.

The transition relation Δ encodes the dynamics by providing an over-approximation of the image set of every point in the abstract state-space. Specifically, for $x_d, x'_d \in \mathcal{X}_d$ and $u_d \in \mathcal{U}_d$, we say:

$$(x_d, u_d, x'_d) \in \Delta \Leftrightarrow \exists x \in R_x^{-1}(x_d), u \in R_u^{-1}(u_d), t > 0, \\ : \xi_{x,u}(t) \in R_x^{-1}(x'_d), \quad (6)$$

where $\xi_{x,u}(t)$ denotes the solution to (1) starting from state x under input u.

Given this discrete abstraction, controller synthesis can be formulated as a reachability analysis over the transition system. The goal is to design a symbolic controller $\mathcal{C}_d : \mathcal{X}_d \to 2^{\mathcal{U}_d}$ such that all trajectories starting from the initial set reach the target set while avoiding the obstacle set.

Concretization and Implementation Now, the symbolic controller $\mathcal{C}_d$ is concretized for implementation on the original system using *feedback concretization*. For any continuous state $x \in \mathcal{X}$, the controller selects an input:

$$\mathcal{C}(x) \in \mathcal{C}_d(R_x(x)), \quad (7)$$

where $R_x(x)$ is the abstract state corresponding to x under relation R_x. This ensures that any input selected from $\mathcal{C}_d$ is valid for the system's true dynamics. Thus, the closed-loop system with controller $\mathcal{C}$ satisfies the original specification Σ^{reach}, assuming a sound abstraction.

The ABCD pipeline enables formal controller synthesis with guaranteed safety and performance. This framework forms the foundation of the symbolic synthesis procedures implemented in `Dionysos`.

3. PROBLEM STATEMENT

As a case study, we focus on reach-avoid problems, using the bicycle model (Reissig et al., 2016) with dynamics:

$$f(x, (u_1, u_2)) = \begin{pmatrix} u_1 \cos(\alpha + x_3) \cos(\alpha)^{-1} \\ u_1 \sin(\alpha + x_3) \cos(\alpha)^{-1} \\ u_1 \tan(u_2) \end{pmatrix}, \quad (8)$$

where $x \in \mathcal{X} \subseteq \mathbb{R}^3$ is the state vector: the first two components represent the position of the bicycle, and the third is its orientation. The control inputs $(u_1, u_2) \in \mathcal{U} \subseteq \mathbb{R}^2$ correspond to the rear wheel velocity and the steering angle, respectively. The parameter α is the angle between the bicycle's orientation and the velocity direction of its center of mass, computed as: $\alpha = \arctan(\tan(u_2)/2)$. The inputs are constrained to the compact set $\mathcal{U} = [-1, 1] \times [-1, 1]$, while the state space $\mathcal{X}$ depends on the environment in which the bicycle operates.

The goal of the reach-avoid problem is to synthesize a controller that drives the system from an initial set $\mathcal{X}_{\text{in}}$ to a target set $\mathcal{T}$ within finite time, while avoiding a set of obstacles $\mathcal{O} \subset \mathcal{X}$ throughout the trajectory.

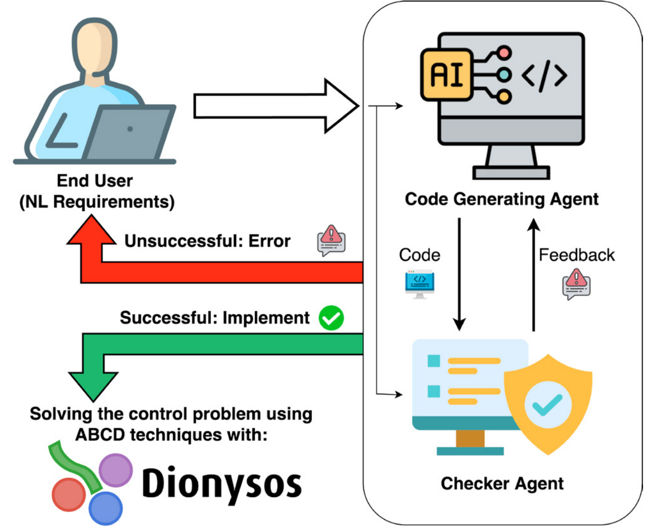

Fig. 1. Overview of the procedure with two LLM-based agents.

In this setting, the system dynamics are fixed across tasks, while the problem specifications (e.g., target and initial states) vary from task to task. Therefore, in this work, LLMs are leveraged only at the level of problem specification extraction. Once the specifications are correctly extracted, the control problem can be solved using symbolic control techniques.

4. LLM-BASED AGENTS

This section introduces two LLM-based agents, both implemented using OpenAI's ChatGPT-4o via few-shot prompting. These agents are integrated into the workflow as illustrated in Fig. 1. The first agent serves as a Code Generator, and the second functions as a Checker.

Given a NL task description specifying the environment and safety constraints, the Code Agent extracts relevant information and produces code compatible with the `Dionysos` input format for solving reach-avoid problems. The second agent evaluates the code generated by the first agent, identifying mismatches between the code and the original specification. It then provides targeted feedback to refine the code and improve its correctness and safety. This iterative process can be repeated up to $k_{\max}$ times, allowing the system to revise the code until a correct implementation is found. If the code remains incorrect after $k_{\max}$ attempts, the process terminates and the final feedback is forwarded to the user instead of executing invalid code. This design reflects our central assumption that narrowing down the responsibilities of each LLM leads to more reliable outputs. By assigning the generation task to one agent and the evaluation task to another, the system leverages the strengths of LLMs in both structured code generation and high-level semantic reasoning.

4.1 Code Generator

This agent processes NL descriptions of the operating environment, such as its boundaries, the positions of

Code Agent

Task Assignment

I want to use the Julia package Dionysos to solve optimal control problems using abstraction-based methods. To this end, I will provide problem specifications in natural language. Your task is to extract the relevant specifications—including the environment, initial state, target state, and obstacles—and generate the corresponding Julia code for that part of the problem, ensuring compatibility with the rest of the codebase.
The generated code must strictly follow the format used in the provided examples. In some cases, you may receive a sequence of coordinates representing maze-shaped obstacles. These obstacles can typically be interpreted as a set of adjacent rectangular regions. In such cases, you should first decompose them into individual rectangles and then treat each one as a standard rectangular obstacle.
I will now provide an example so you know exactly how to proceed.

Example 1

Input: In the [0,8]×[0,5] environment, there are four rectangular obstacles with the following vertices: I) (0,4), (3.5,4), (3.5,4.2), (0,4.2) II) (4.5,4), (7,4), (7,4.2), (4.5,4.2) III) (2,2.4), (6,2.4), (6,3), (2,3) IV) (3,0.5), (5,0.5), (5,1.6), (3,1.6). The task is to first go from the initial position (0.6, 4.6) to (7, 2.5), and then from there to (1, 0.5), avoiding all obstacles.
Your expected response:

```
x_low = [0.0, 0.0, -pi - 0.4]
x_upp = [8.0, 5.0, pi + 0.4]
x_start= [0.6, 4.6, 0.0]
x_target =[ [7.0, 2.5, 0.0], [1.0, 0.5, 0.0] ]
x1_lb = [0.0, 4.5, 2.0, 3.0]
x1_ub = [3.5, 7.0, 6.0, 5.0]
x2_lb = [4.0, 4.0, 2.4, 0.5]
x2_ub = [4.2, 4.2, 3.0, 1.6]
```

Example #

Fig. 2. Coder Agent prompt including task assignment and examples.

obstacles, and the requirements, such as the target(s) to be reached. The agent must extract the essential information and generate corresponding code in a specific format suitable for integration into the reach-avoid pipeline. The prompt provided to the LLM is illustrated in Fig. 2.

In constructing the examples for the LLM, care has been taken to include a diverse range of specifications, such as: (i) simple goals with a single target, e.g., "Go to Position$_1$ avoiding all obstacles"; (ii) sequential goals with multiple targets, e.g., "Reach Position$_2$ after visiting Position$_1$, while avoiding all obstacles"; and (iii) safety constraints such as clearance, e.g., "You must not get closer than 1 unit to any obstacle."

Moreover, multiple NL variants are considered for defining obstacles. A rectangular obstacle may be described using: (i) the coordinates of its four vertices, (ii) the coordinates of two diagonal vertices, or (iii) its center along with the side lengths in each dimension.

The examples reflect a representative range of phrasing and descriptive styles common in NL specifications. While such variability is inherent to NL, the goal is to provide the LLM agent with structured prompts to demonstrate its ability to interpret and process these variations within the task scope.

Checker Agent

Task Assignment

You are assigned the role of a checker and feedback provider. We have an LLM-based agent that, given the natural language requirements for reach-avoid problems, extracts the corresponding specifications and generates code in a specific format. However, we are not certain whether the generated code is correct.
Your task is to compare the natural language requirements with the generated code to ensure they match. If there are discrepancies or errors, you should identify the issue and inform the Code Generating agent with a clear explanation. If the code is accurate and fully reflects the requirements, simply return "True", allowing the process to proceed using the validated information.
Now, we will provide some examples so you can learn how to perform this task.

Example 1

NL Requirements: *Same as Fig. 2.*

Code: *Same as Fig. 2, except for the following differences:*

```
x_upp     = [8.0, 8.0, pi + 0.4]
x1_ub     = [3.5, 7.0, 6.5, 5.0]
```

Your Response: There are two mistakes in the code. First, the upper bound in the x_2 direction is 5.0, but in the generated code, it is incorrectly set to 8.0. Second, the upper bound of the third obstacle is 6.0, but you incorrectly wrote 6.5 in the code. Please regenerate the code with these corrections.

Example #

Fig. 3. Checker Agent prompt including task assignment and examples.

4.2 Checker Agent

The Checker Agent receives both the original NL specifications and the generated code. Using a few-shot prompting strategy with targeted examples, it is queried to identify whether the implementation aligns with the intended specifications, as shown in Fig. 3. If the output satisfies the specification, the Checker confirms correctness by simply returning the word `True`, and the process continues. Otherwise, it returns a structured explanation of the mismatches, prompting the Code Agent to revise its output. An example of the expected back-prompting of the Checker Agent is also available in Fig. 3. The structure of the feedback generated by the Checker Agent is identical for both internal communication (with the Code Agent) and external communication (with the end user).

In this work, $k_{\max}$ is set to 2. This relatively low value was chosen based on empirical observations that, in longer loops, the Code Agent sometimes learns to bypass the checker by producing incorrect but deceptively valid-looking code. Setting a strict limit on the number of iterations improves robustness by preventing infinite feedback loops and reducing the risk of unsafe behavior due to a misled Checker Agent.

5. PERFORMANCE EVALUATION

We evaluate the methodology on a dataset of varied environments, measuring agent performance by success rate across strategies.

5.1 Test Set

We constructed a dataset of 20 environments with varying obstacle configurations. For each environment, we first manually crafted a NL specification describing the task and constraints. These specifications span a range of application scenarios, such as autonomous vehicles navigating without colliding with buildings, or mobile robots operating in warehouses while avoiding shelves.

To introduce linguistic diversity and test robustness, each original specification, as described in Sec. 4.1, was paraphrased three times, resulting in a total of 60 statements. The paraphrases vary not only in phrasing but also in how obstacles and environments are described, while preserving the original semantics. This enriches the evaluation dataset and allows us to assess the robustness of the LLM (Siska et al., 2024): if the generated output remains consistent across semantically equivalent inputs, it indicates that the LLM handles variations in expression reliably.

5.2 Results

This section presents an ablation study of three strategies: (1) solving directly using an LLM, (2) code generation using only the Code Agent, and (3) code generation and validation using both the Code Agent and the Checker Agent (see Fig. 4). This comparison highlights the effectiveness of LLM-based agents in combining the flexibility of NL interfaces with the reliability of formal methods.

In the top chart of Fig. 4, all 60 paraphrased instructions from the test set are categorized into four groups: (i) Incorrect execution: incorrect results that were executed anyway, (ii) Correct Code, but not Checked: correct results without any mechanism to verify their correctness (thus unreliable), (iii) Incorrect Code Blocked by Checker: incorrect results that were detected and blocked by the Checker Agent, and (iv) Correct and Checked Code: correct results that were verified by the Checker Agent, improving reliability. The number of correct implementations rises significantly from 7 in the first strategy to 34 in the second and 39 in the third. While using the Code Agent alone improves performance, many incorrect codes are still executed, and none are formally verified. Incorporating the Checker Agent not only increases the number of correct solutions but also greatly improves reliability by preventing incorrect implementations, except for a few cases.

Throughout the entire test set, the LLMs did not produce any syntactic errors. Since the target code has a constrained format and consists of only a few lines, the outputs were always syntactically valid. All incorrect results stemmed from semantic errors, where the LLMs failed to correctly interpret and reason about the NL specifications.

The bottom chart of Fig. 4 compares the number of test problems (out of 20) that were solved *robustly* under each strategy. By *robust*, we mean that the strategy produced a

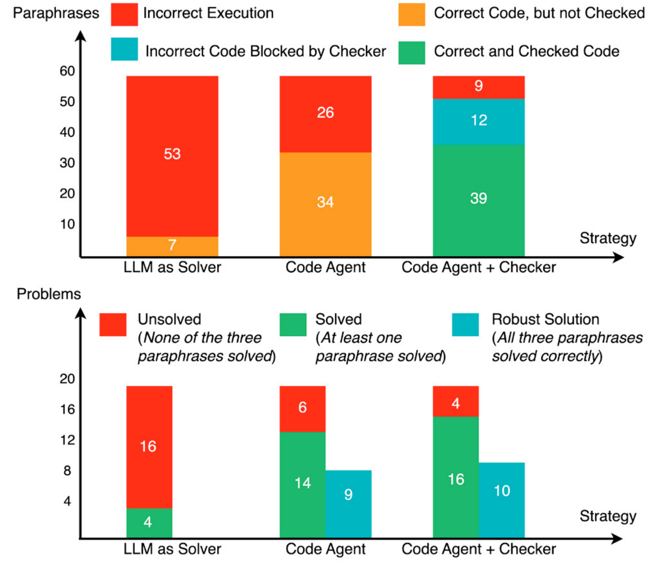

Fig. 4. Ablation study on the performance of the LLM-based agents. Top: performance across all paraphrases ((20 problems) × (3 paraphrases) for each). Bottom: performance over all 20 problems in the test set.

valid solution for all three paraphrased variants of a given problem. A problem is considered *solved* if at least one paraphrase led to a correct code, and *incorrect* if none did. Among the four problems that were solved using only the LLM (Strategy 1), none were solved robustly across all three paraphrases. In contrast, using the Code Agent alone led to 9 out of 14 problems being solved robustly (64.3%), and the full pipeline with both agents solved 10 out of 16 problems robustly (62.5%). It is worth noting that, while the results may vary slightly due to the non-deterministic nature of LLM outputs, the overall robustness, exceeding 60% in both agent-assisted strategies, along with our observations, indicates that performance is relatively stable and is not expected to change significantly across repeated runs.

Figure 5 compares solutions for three cases: (1) a trajectory generated by directly using an LLM to solve the control problem; (2) an incorrect trajectory due to the Code Agent misrepresenting obstacle specifications; and (3) a correct trajectory obtained via symbolic control. In case (2), the mismatch between actual and represented obstacles is visible. Although the Checker Agent correctly flagged this issue, we proceeded with the flawed input to illustrate such errors. Case (1) shows that directly using an LLM results in trajectories violating obstacle constraints, highlighting their limitations as solvers.

6. CONCLUSION

This paper takes a first step toward a seamless and safe framework for solving safety-critical control problems using NL as the user interface to ABCD. We demonstrated that LLMs can effectively bridge the gap between high-level task descriptions and symbolic control implementation. A Code Agent uses few-shot prompting to extract reach-avoid specifications from NL and generate code compatible with `Dionysos`, while a second agent, the Checker Agent, validates the correctness of this code and provides

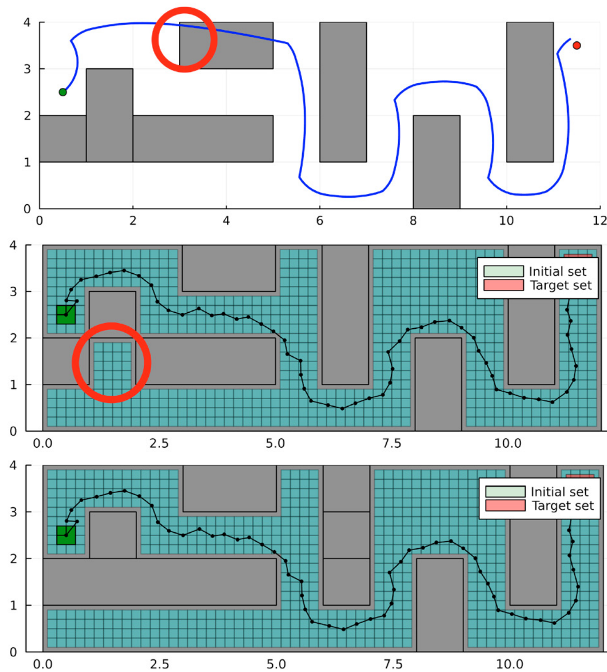

Fig. 5. Example trajectories from three strategies. Top: LLM used as a solver fails to avoid obstacles; Middle: Code Agent extracts incorrect specifications; Bottom: correct trajectory using symbolic control

structured feedback. This iterative process improves both safety and reliability. Although the approach lacks formal guarantees, it significantly increases the success rate in solving reach-avoid problems and illustrates the potential of LLMs to reduce the domain expertise typically required for correct-by-design control synthesis.

Future work includes fine-tuning LLMs for safety-critical tasks, translating NL into formal specifications like temporal logic, and integrating LLMs for real-time interaction with dynamic systems. This is especially relevant in applications where evolving requirements demand adaptive behavior to maintain correctness over time.

ACKNOWLEDGEMENTS

Raphaël Jungers is a FNRS honorary Research Associate. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 864017 - L2C, from the Horizon Europe programme under grant agreement No101177842 - Unimaas, and from the ARC (French Community of Belgium)- project name: SIDDARTA.

REFERENCES

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., et al. (2022). Do as I can, not as I say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Arora, G., Jain, S., and Merugu, S. (2024). Intent detection in the age of LLMs. *arXiv preprint arXiv:2410.01627*.

Badings, T., Romao, L., Abate, A., Parker, D., Poonawala, H.A., Stoelinga, M., and Jansen, N. (2023). Robust control for dynamical systems with non-gaussian noise via formal abstractions. *Journal of Artificial Intelligence Research*, 76, 341–391.

Banse, A., Romao, L., Abate, A., and Jungers, R. (2023). Data-driven memory-dependent abstractions of dynamical systems. In *Learning for Dynamics and Control Conference*, 891–902. PMLR.

Calbert, J., Banse, A., Legat, B., and Jungers, R.M. (2024a). Dionysos. jl: a modular platform for smart symbolic control. *arXiv preprint arXiv:2404.14114*.

Calbert, J., Egidio, L.N., and Jungers, R.M. (2024b). Smart abstraction based on iterative cover and non-uniform cells. *IEEE Control Systems Letters*.

Chen, Y., Arkin, J., Dawson, C., Zhang, Y., Roy, N., and Fan, C. (2024). Autotamp: Autoregressive task and motion planning with LLMs as translators and checkers. In *2024 IEEE International conference on robotics and automation (ICRA)*, 6695–6702. IEEE.

Guo, X., Keivan, D., Syed, U., Qin, L., Zhang, H., Dullerud, G., Seiler, P., and Hu, B. (2024). ControlAgent: Automating control system design via novel integration of LLM agents and domain expertise. *arXiv preprint arXiv:2410.19811*.

Haesaert, S., Soudjani, S., and Abate, A. (2017). Verification of general Markov decision processes by approximate similarity relations and policy refinement. *SIAM Journal on Control and Optimization*, 55(4), 2333–2367.

Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. (2024). A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., and Sabharwal, A. (2022). Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*.

Nilsson, P., Ozay, N., and Liu, J. (2017). Augmented finite transition systems as abstractions for control synthesis. *Discrete Event Dynamic Systems*, 27(2), 301–340.

Njah, Y., Leivadeas, A., and Falkner, M. (2025). An AI-driven intent-based network architecture. *IEEE Communications Magazine*, 63(4), 146–153.

Pan, J., Chou, G., and Berenson, D. (2023). Data-efficient learning of natural language to linear temporal logic translators for robot task specification. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 11554–11561. IEEE.

Reissig, G., Weber, A., and Rungger, M. (2016). Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Transactions on Automatic Control*, 62(4), 1781–1796.

Siska, C., Marazopoulou, K., Ailem, M., and Bono, J. (2024). Examining the robustness of LLM evaluation to the distributional assumptions of benchmarks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 10406–10421.

Tabuada, P. (2009). *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

Zamani, M., Esfahani, P.M., Majumdar, R., Abate, A., and Lygeros, J. (2014). Symbolic control of stochastic systems via approximately bisimilar finite abstractions. *IEEE Transactions on Automatic Control*, 59(12), 3135–3150.