electronics

MDPI

Article

State Machine Model of the Operation Control of a Differential- Drive Mobile Robot

Lluís Ribas-Xirgo

School of Engineering, Campus UAB, Universitat Autònoma de Barcelona, 08193 Barcelona, Spain; lluis.ribas@uab.cat

Abstract

Existing robotic control frameworks often rely on complex hierarchical state machines or middleware infrastructures, which may be unsuitable for resource-constrained embedded systems and difficult to map directly to low-level code. This work presents a complete state-machine model for the reactive control layer of a differential-drive mobile robot. Although mobile-robot controllers often exhibit considerable complexity—particularly at lower levels, where numerous hardware-dependent operations occur—this work shows that a network of concurrent state machines provides a clear and lightweight method for specifying and implementing control behavior. The proposed approach decomposes the reactive controller into two concurrent extended finite-state machines (CEFSMs) responsible for locomotion and lidar operation, connected through simple and predictable protocols. This structure enables a direct mapping from model diagrams to procedural code in languages such as C++ and Lua. The method has been used extensively in an undergraduate Embedded Systems course since 2011, supporting both physical robots (Arduino-based) and their digital twins in CoppeliaSim. We also introduce a discrete control strategy that approximates continuous behavior and incorporates a simplified proportional–integral–derivative (PID) controller for pose correction. The approach reduces development effort, increases model clarity, and yields reusable code across hardware and simulation platforms. Quantitative evaluation shows that the proposed PID-based control strategy can improve positioning accuracy by up to an order of magnitude compared to a baseline on–off controller, without increasing the computational cost. In addition, the control cycle time has been significantly reduced (from 31 ms to 12 ms) in order to ensure a stable execution period; this, in turn, results in a more predictable controller output and improved trajectory consistency.

Keywords: differential-drive mobile robots; embedded control systems; extended finite-state machines (EFSM); concurrent state machines; state-machine-based control; reactive robot control; model-to-code mapping; digital twins; robotics education

Check for updates

Academic Editor: Giuseppe Prencipe

Received: 11 May 2026
Revised: 3 July 2026
Accepted: 6 July 2026
Published: 8 July 2026

Copyright: © 2026 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Designing software controllers for mobile robots is inherently challenging due to the wide range of functionalities they must coordinate and the tight real-time constraints imposed by hardware operation. While software libraries can abstract sensor handling, actuator control, and communication mechanisms, the reactive behavior of the robot—how it decides and executes low-level actions—remains difficult to describe unambiguously and implement reliably.

Finite-state machines (FSMs), and hierarchical extensions thereof, are widely used to model robotic behavior. However, many existing frameworks (Section 2) provide rich semantics designed for complex applications, making them less suitable for resource-limited embedded platforms or introductory engineering environments. In educational contexts, the sophistication of these tools can obscure fundamental concepts such as event handling, time-dependent behavior, concurrency, and data flow between modules.

This paper proposes a lightweight yet expressive methodology for modeling robot controllers based on networks of concurrent extended finite-state machines (CEFSMs) with minimal and well-defined semantics. The approach aims to reduce non-recurring engineering effort, remove ambiguity from behavioral descriptions, and enable a direct and systematic translation from diagrammatic models to executable code in procedural languages such as C++ or Lua.

Within this framework, a complete modular control architecture (Section 3) for a differential-drive robot is presented, encompassing a fully reactive layer specified entirely through CEFSMs that interact by means of simple synchronization signals (Section 4). The methodology further incorporates a hybrid, discretized movement-control strategy (Section 5) that includes a lightweight PID-based correction mechanism suitable for microcontrollers with limited computational resources. To support seamless implementation, a set of programming templates and design patterns is introduced (Section 6), establishing a strict one-to-one correspondence between diagrammatic elements, such as states, transitions, and variables, and code constructs, thereby reducing implementation errors and improving maintainability.

The proposed framework has been validated in an educational context over more than a decade of classroom deployment and supports both physical Arduino-based robots and simulated platforms such as CoppeliaSim. Experimental evidence demonstrates a high degree of reusability and portability, as adapting the controller to new robotic platforms or simulation environments typically requires modifying only thin input/output layers.

Within this context, the paper presents a development process that starts with the design of the controller architecture (Sections 3 and 4), defining the number of CEFSMs, their functional responsibilities, and their interaction and synchronization mechanisms. Each state machine is then specified in detail (Section 4), including hybrid variants that integrate discretized models of continuous-time controllers (Section 5). The resulting models are finally translated into software using consistent design patterns and programming templates (Section 6), enabling both functional and behavioral verification in a digital twin as well as deployment on the real robotic platform. This approach maintains conceptual simplicity while supporting expressive modeling of reactive, time-dependent, and hybrid robotic behaviors. It is therefore well-suited for engineering education, rapid prototyping, and embedded-systems development.

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

- A lightweight architecture based on CEFSMs with minimal semantics, enabling direct mapping from models to embedded code.
- A complete reactive control layer for differential-drive robots, including locomotion and lidar subsystems.
- A discretized motion-control strategy incorporating a proportional–derivative (PD)-like controller derived from incremental pose measurements.
- A systematic programming methodology based on templates ensuring correctness and portability.
- Experimental validation demonstrating real-time performance on both simulated and physical platforms.

https://doi.org/10.3390/electronics15142993

2. Literature Overview

FSM models, and their extensions such as extended finite-state machines (EFSMs), possess formal semantics that simplify behavior specification, verification, and software synthesis. In industrial robotics, particularly, robot operations can be translated into FSMs for repeatable, autonomous execution. For example, Sauer, Hender, and Martens (2019) [1] propose kinesthetic programming where users teach robots procedures by generating state machines that record both operations and sensor data, enabling decision-making when alternative operation sequences exist. Halim et al. (2022) [2] propose a similar system to teach tasks to robots. However, their operation sequences are limited to guiding robot arm tip trajectories, and state machines represent the main controller's behavior during learning, teleoperation, or playback modes. The former system uses an interpreter for generated state machines, while the latter employs hard-coded ones. Both systems demonstrate state machines' suitability for representing system behaviors without requiring user expertise in state machines or programming. These cases involve human-machine interfaces using state machines to model high-level robot behavior [1] and user-robot interaction [2], with relatively simple state machine structures. General behaviors require more flexible state machine structures and frameworks supporting development through implementation.

To describe complex behaviors, state machines can be combined in various ways (parallel or hierarchical organizations) [3–5] or integrated with other computational models including programs. Explicit hierarchical FSM composition introduces compound states into parent state machines and additional features defining subordinate FSM activation (with history or initial state initiation) and termination (upon reaching end states or exiting their parent compound state). Numerous frameworks exist for describing state machines. For instance, Samek and Montgomery (2000) [6] presented C/C++ patterns for hierarchical state machines (HSMs) represented by UML state machines, enabling direct mapping between the two. Other similar frameworks are RAFCON [7] and RoboChart [8,9]. RAFCON [7] is a graphical tool for engineering robotic tasks through UML-like HSMs, capable of exporting data for software synthesis tools. RoboChart [8,9] is a tool based on simplified UML state machines that supports model checking and verification of temporal properties. Although all these frameworks support code generation, HSMs can also be programmed directly via specialized languages like XRobots [10], where relations between compound states and contained FSMs are established through function-like calls, or using programming patterns [6].

This work employs state-oriented programming based on patterns and templates [11] across different programming languages, without relying on additional libraries. For simplicity, hierarchy is established via handshaking protocols between master and slave modules. This approach greatly simplifies the graphical language semantics and reduces the framework's learning curve compared to alternatives.

Although state machine diagrams can be programmed manually, automatic translation from diagrams.net (formerly draw.io) drawings to Lua or other programming languages can be achieved using our custom tool [12] or, more recently, using artificial intelligence (AI) tools. While no direct link to model checking tools exists, properties can be tested dynamically by including specific methods in corresponding state machine classes, and specific loggers can capture execution traces.

Robot controller behaviors are typically described using multiple state machines organized as HSMs [13] or layered architectures, as in our proposal where no explicit hierarchical dependencies exist. Combinations are also possible. For example, YASMIN [14] enables designing state machines that communicate through a blackboard and synthesizing code for some modules of a predefined architecture on top of ROS2. More generally, recent robotic architectures rely on middleware such as ROS2 to support distributed execution,

real-time communication, and modular system integration [15]. In addition, frameworks such as Navigation2 build upon ROS2 to provide navigation capabilities and commonly employ behavior trees (BTs) for task execution [16]. These approaches provide improved modularity and scalability for complex task execution and decision-making in robotics [17].

Beyond middleware-based approaches, other options for higher-level behavior descriptions include dynamic stack deciders [18], a type of stack automata capable of changing action plans by modifying the state stack with decision trees. Simulating stack automata is possible in our case because we use FSMs with extended memory (EFSMs) that can be changed from cycle to cycle in the same way as the main state.

However, these middleware-based approaches typically target distributed, high-level robotic systems and introduce additional complexity and runtime overhead, which may be unnecessary or unsuitable for low-level embedded control layers. This work focuses on the lower levels of the robot controller stack, where actuation plans remain constant. With respect to this, it closely relates to earlier work from 2004 [19], where algorithmic state machines (ASMs) and FSMs modeled drivers and the L0 level of a mobile robot. However, that work constrained models to binary inputs and outputs (each corresponding to a micro-operation), like the control part of an external operational unit. Our EFSMs are unconstrained, allowing changes to both the main state and additional state variables during transitions. Although transitions in state machines modeling high-level robot operations and decision-making processes are not highly constrained, some steps may require continuous control output updates or guaranties of timely discrete-time updates. This is particularly relevant for robot movement, where locomotion controllers are modeled as discrete-time state machines that can operate in continuous mode at certain states, i.e., as hybrid state machines.

For differential-drive robots, simple modifications to control variables applied at high rates can approximate continuous operation. For example, Rustam et al. [20] presented a simple control algorithm for straight-line movement that applies small step changes to control variables based on sideways drift. In our case, steps derive from an analytical model and are proportional to the error and a derivative factor to prevent single-cycle error correction. Indeed, controllers based on PID formulations are commonly used, with parameters identified to achieve smooth robot operation [21,22].

In addition to architectural frameworks, numerous control strategies have been proposed for mobile robots, including PID-based control, fuzzy logic approaches, model predictive control, and learning-based methods. These approaches can be broadly categorized into three paradigms: model-based control, learning-based control, and structure-based behavioral modeling.

Adaptive PID methods dynamically adjust controller gains to cope with time-varying or nonlinear system dynamics, improving convergence and robustness compared to fixed-gain controllers [23,24]. Similarly, BELBIC-based PID controllers, inspired by brain emotional learning, have been extended in recent works to improve dynamic performance in complex environments [25–27]. CMAC-PID controllers integrate neural learning mechanisms to approximate nonlinear mappings and tune controller parameters, with recent variants focusing on improved generalization and adaptation capabilities [28–30]. In addition, more recent formulations include sigmoid-based PID controllers with nonlinear gain modulation and neuroendocrine-inspired PID structures, which further extend adaptive and biologically inspired control capabilities [31,32].

Many of these approaches rely on incorporating nonlinear activation functions and advanced adaptation mechanisms to enhance control performance, by incorporating nonlinear activation functions and advanced adaptation mechanisms. For example, hyperbolic tangent-based PID controllers introduce smooth nonlinear gain modulation that improves

transient response and robustness in strongly nonlinear systems [33], while modern nonlinear adaptive PID schemes employ dynamic gain shaping or modulation functions to enhance stability and tracking performance under uncertainty [34].

Overall, these developments highlight that PID-based control remains an active and evolving research area, particularly in nonlinear and adaptive formulations.

While these approaches improve performance under nonlinear and uncertain conditions, they also introduce higher implementation overhead and reduced interpretability, which highlights a fundamental gap between advanced controller design and the practical constraints of embedded control systems.

From a broader perspective, model-based methods (e.g., model predictive control) rely on accurate system models and optimization, while learning-based approaches (e.g., neural or adaptive controllers) require training data or online adaptation, often increasing system complexity and reducing transparency. In this context, the present work adopts a lightweight, structure-based approach, where the control logic is explicitly represented using interacting CEFSMs. This approach enables predictable execution and well-defined timing behavior, along with a direct mapping from models to code, while allowing the integration of a simple yet effective PD-like control strategy within a well-defined execution model. This combination facilitates straightforward verification and direct implementation in embedded systems without requiring explicit system identification or training processes.

This perspective complements, rather than replaces, existing control strategies by focusing on system-level integration and execution semantics rather than controller optimality. This design choice emphasizes simplicity, interpretability, and deterministic behavior, making it particularly suitable for resource-constrained platforms and educational environments.

Some works enable robots to reach specific target points given in polar coordinates without prescribed trajectories, while others constrain robot movement to specific paths like curved trajectories [35]. The latter cases require more complex expressions than simple PID formulations [36] to adjust control variables for accurate, smooth movement along predefined trajectories. In other cases, reference trajectories are generated dynamically. For instance, [37] presented a wheelchair control combining user control variables with controller variables following curvilinear trajectories. This offers another perspective on adjusting control variables considering the robot's actual trajectory and variable references, such as when obstacles appear suddenly. In our case, state machines controlling robot wheels include states with quasi-continuous PID control modes that divide trajectories into two parts: orientation change and straight movement to the destination.

Although the present work focuses on differential-drive robots, the proposed CEFSM-based architecture is largely independent of the underlying kinematic model. Extension to other platforms, such as omnidirectional or tracked robots, mainly requires adapting the motion-control equations while preserving the same CEFSM structure and communication mechanisms.

3. Controller Stack

The robots must navigate their environment to complete a given mission. A typical mission involves reaching a specific point to, for example, pick up or deliver materials.

Programming the corresponding controller requires first designing its behavioral model, preferably independent of implementation details. This work uses a network of interacting state machines, each modeling a controller component's behavior.

Before detailing state machines, we must define the controller architecture, describing system components and their communication. We represent components with large rectangles, signals with squares, and communications' directions with arrows.

Signals are updated *continuously*, meaning that in this discrete system, they are computed at every state machine transition. Signals may have memory, in which case they need not be recomputed at each cycle; conversely, they may also be updated spuriously.

We use squares or rectangles to represent memoryless signals and circles to represent signals with memory, i.e., state variables. All module inputs are memoryless signals, whereas outputs may be either memoryless or bound to variables (in the latter case, represented by circles inscribed within squares).

Since pure output signals may create undesired combinatorial feedback loops causing signal value determination issues, all output signals derive from variables.

A combinational feedback loop occurs when an input signal's value at a given time depends on a pure output signal from the same module at the same time, directly or indirectly. This situation makes signal values difficult to predict, as they depend on system implementation specifics.

We use a layered architecture (Figure 1) for the robot controller to progressively decompose general missions into detailed actions. The simplest mobile robot mission involves moving from its current position to a target location by following given path steps. More complex missions provide only the destination, with or without complete environment maps, requiring exploration and mapping.

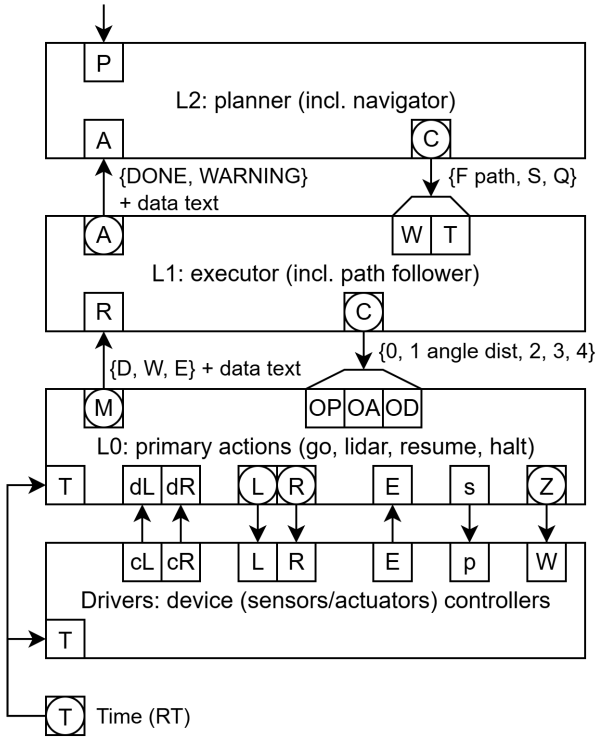

Figure 1. Four-layer controller architecture. Each layer communicates with the layer below and the layer above through input and output signals. All output signals come from variables (boxes with the name enclosed in a circle), whereas all input signals are memoryless (boxes with the name, without a circle). Each layer may consist of multiple modules; in particular, layer L0 includes two modules: one responsible for communication with the other layers and for locomotion, and another dedicated to lidar management.

The highest level of the controller stack (L2) receives a path P and guides the vehicle along it until either the path is fully traversed or a failure occurs, in which case the vehicle returns to the initial point. While more complex behaviors could be incorporated, they are beyond the scope of this paper.

To achieve this, L2 issues commands to L1 through the output signal C and monitors their successful execution by inspecting the input signal A, which carries L1's responses to the issued commands.

In L1, commands received from L2 are decomposed into an operation code W and an optional path T. The input opcode W may be 'F' to execute a follow-path command, 'S' to perform a lidar sweep, or 'Q' to abort the current operation and enter a waiting state. For the mission considered, L1 processes the path T and issues action commands to the lower level via the output signal C. This signal contains either the empty string or "1 αr", which instructs L0 to move the robot to the polar coordinates (α, r).

In L0, these actions take the form of triplets $\langle OP, OA, OD \rangle$, which contain the opcode (or 0 if no command has been received at that time), the angle α, and the radius r, respectively.

L0 transforms each GO (opcode $OP = 1$) command into corresponding left (L) and right (R) motor control values until the vehicle reaches the destination. These values, along with whether the robot has reached its final pose, depend on wheel encoder data dL and dR. Upon completing an action or encountering unexpected events, L0 sends an appropriate message M to the upper level.

Additionally, L0 communicates with the drivers to direct the lidar to a specific angular position Z and retrieve the distance to objects E (from echo). Beyond obstacle detection during forward movement, this communication enables scanning distances to objects across the 180-degree front field of view.

The controller's lowest level contains device drivers, which take DC motor control values for left and right wheels and the servomotor adjusting lidar orientation, providing L0 with encoder (cL, cR) and lidar (E) data.

All components integrate into a framework including an Arduino-compatible board and Bluetooth communication module (Figure 2). Front and rear caster wheels provide stability during movement. The kinematic center is located at the robot's geometric center to simplify position computations. Similarly, the lidar is positioned above the platform center.

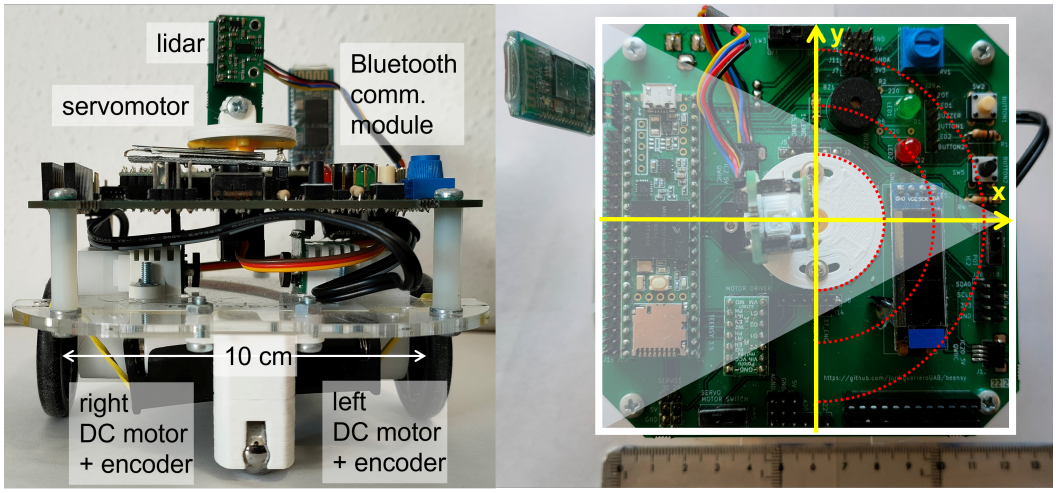

Figure 2. The mobile robot integrates essential hardware for communication (Bluetooth), environment sensing (lidar), and closed-loop motion control (motors and encoders). The geometric and kinematic centers coincide to simplify position computations (right side).

Although the present work focuses on differential-drive robots, the proposed CEFSM-based architecture can be adapted to different robotic platforms with minimal changes in the motion-control layer. In particular, the decomposition of the reactive controller into interacting CEFSMs, as well as the communication and synchronization mechanisms between modules, is not tied to a specific locomotion system.

Consequently, the approach can be extended to other mobile platforms, such as omnidirectional robots or tracked vehicles, by adapting only the motion-control layer described in Section 5. In these cases, the kinematic relationships used to compute pose evolution and control inputs must be reformulated according to the corresponding platform model, while the overall CEFSM structure and interaction protocols remain unchanged.

This separation of concerns allows the proposed methodology to maintain its simplicity and modularity while supporting different robotic platforms with minimal redesign effort.

4. L0 Model

This layer executes four basic actions: GO, LIDAR, RESUME, and HALT. The GO (instruction with opcode $OP = 1$) action moves the vehicle to a specified position in polar coordinates relative to its kinematic center, while LIDAR ($OP = 2$) sweeps a 180° front field of view. For simplicity, GO and LIDAR commands cannot be executed in parallel. However, the lidar is used to detect obstacles directly ahead during movement. The LIDAR command initiates a lidar sweep but requires L1 to RESUME after each reading.

HALT stops any ongoing command execution and, during lidar sweeps, returns the lidar sensor to forward orientation. Accurate lidar readings require the servomotor to be stationary, but the L0 controller can communicate with higher levels regardless of whether it is rotating or not. Although this parallel behavior is less critical during forward movement, it becomes useful during lidar sweeps, where sending data to L1 and rotating the servomotor to target orientation can occur simultaneously. Thus, L0 divides into two controllers: L0Main and L0Lidar (Figure 3). The former handles communication with L1 and the wheels' drivers, while the latter manages lidar orientation and readings.

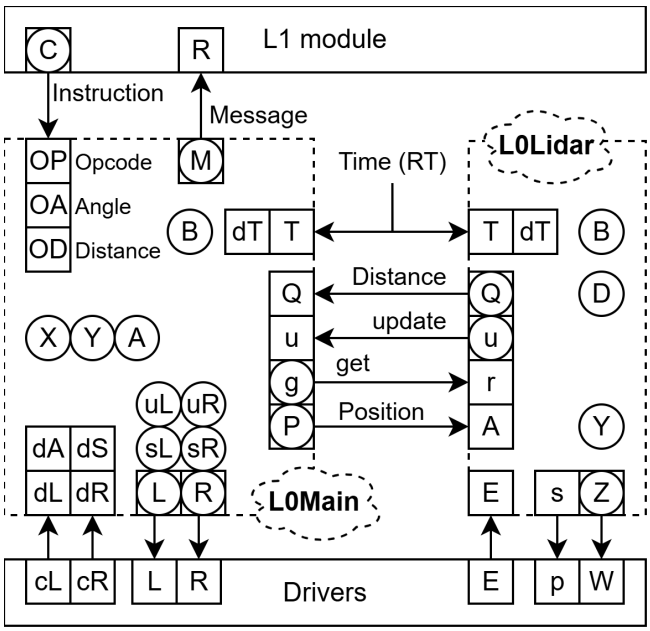

Figure 3. The L0 layer consists of a system of two CEFSMs that interact with the upper L1 execution layer and the driver layer. Internal communication between the L0Main and L0Lidar modules takes place within the same execution process and can operate either in continuous mode (with signal Q updated at every execution cycle) or asynchronously, with the update cycle explicitly indicated by signal u.

Communication between L0Main and L0Lidar is relatively straightforward. L0Lidar provides L0Main with the distance Q to an object at angle A, where A is supplied by L0Main (output P). For lidar sweeps, this communication is governed by a handshake

protocol initiated by L0Main when it sets g (request for a new measurement) to true. The protocol completes when L0Lidar computes a new distance Q and asserts u (update) to *true*. In this case, since both L0Main and L0Lidar belong to the same subsystem and are executed within the same process, data transmission between them is inherently guaranteed, as it simply involves copying values from one program variable to another. Consequently, it is sufficient to verify that the transmission sequence (request–acknowledgment) is correct, while additional mechanisms for detecting missing acknowledgments (e.g., cycle counters to determine communication failures) are not required in this context.

In this work, the behavior of all modules is described using CEFSMs and is therefore independent of their implementation. The advantage of this approach is that the CEFSM representation is formal and, consequently, verifiable not only with respect to functional aspects but also non-functional aspects, particularly in terms of cycle time.

The graphical description of a CEFSM (see Figure 4 for reference), as with any other state machine, associates each node of the graph with a main state, identified by a name inside the corresponding circle. The arcs between nodes represent transitions that are triggered when the logical expression on their labels evaluates to true. Thus, the arrows indicate the next state when the associated condition is satisfied. In this context, the activation of multiple conditions or the absence of any activated condition are considered error cases, corresponding to indeterminacy and incompleteness of the specification, respectively.

The variables that extend the main state are updated whenever a delayed assignment occurs, once the transition has been completed. Delayed assignments are denoted by a plus sign as a superscript to the variable name (or, in plain text, appended at the end) on the left-hand side of the assignment.

Output signals that change immediately as functions of the extended state (Moore model) and of the inputs (Mealy model, which is not used due to issues related to combinational feedback) are shown using conventional assignments with the equal sign as the operator.

In the case of the L0Lidar, the output signal s, corresponding to the emission of laser pulses, depends on the main state, while the remaining outputs are derived from variables Q, u, and Z.

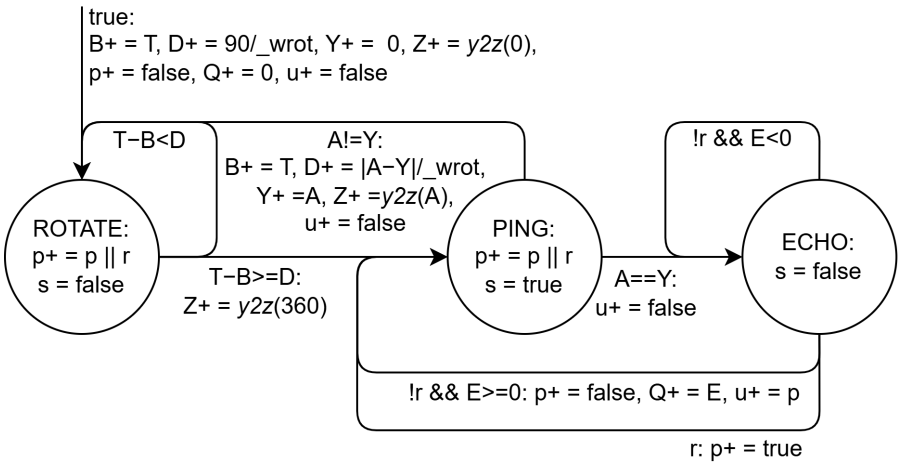

Figure 4. EFSM model of the lidar controller behavior.

The L0Lidar state machine (Figure 4) starts in the initial state, ROTATE, which is responsible for orienting the lidar to the required angular position. The initialization is performed through the actions associated with the start arrow that leads to this state. These actions set the initial target orientation of the lidar to 0°, after which the ROTATE state ensures that the sensor reaches this position. Specifically, the begin time B is set to the

current real time T; the servomotor delay D, required to point to 0°, is set to $90/_wrot$, where *_wrot* denotes the motor angular speed in degrees per second; the target angle Y (yaw) is set to 0; the control value Z is set to the value corresponding to 0° obtained from the function **y2z()**; the pending request flag p is set to *false*; the obstacle distance Q is initialized to 0 (the default value for undetected objects); and the update flag u is set to *false*. After initialization, it performs continuous measurements, acquiring new values as rapidly as possible.

Each state associates with a specific output signal s value, *true* for only one cycle to signal the lidar driver to wait for a new reading. The state machine can interact with external modules with or without explicit communication of detected distances. The former case involves input signal r (request) and variables p (pending request) and u (update acknowledgement), described next.

In the ROTATE state, the machine waits for the servomotor to turn $|A - Y|$ degrees (or 90 degrees initially). It waits D seconds, computed using the servomotor's average rotation speed *_wrot*, and sets servomotor control output Z. The value assigned to Z comes from function **y2z()**, which translates angles to appropriate control values. Argument 360 returns 360, indicating no control signal sent to the servomotor.

At PING, it checks if the input orientation angle A matches the lidar's current position Y. If they match, it moves to ECHO to wait for distance reading E and sets output s (send ping) to *true*. If Y differs from A, the state machine returns to ROTATE to orient the lidar to angle A, and Y is set to A.

When using the handshake protocol, the client module sets r to *true* and A to the desired angle, then waits for u to become *true*. The L0Lidar state machine stores r into p if the request occurs during rotation (ROTATE) or when requesting a new lidar reading (PING), and sets u to *true* only if p is *true*, i.e., only the first reading after u becomes *true*.

The L0Main behavior (Figure 5) involves listening for L1 commands $\langle OP, OA, OD \rangle$ and replying with execution results. For clarity of representation, outgoing transitions from states are expressed using decision trees, where each diamond node contains a logical expression and splits the incoming arc into two outgoing arcs corresponding to the values *false* and *true*.

The initial state is LISTEN, which also acts as the idle state waiting for incoming commands. From this state, transitions may occur toward states corresponding to GO and LIDAR commands, depending on the instruction opcode OP. A GO command may initiate either in the TURN or GO states, depending on whether the polar angle (input signal OA) is nonzero or zero, respectively. The RESUME and HALT commands have no effect at this level, other than generating error messages.

When $OP = 2$ (lidar sweep), L0Main transitions from LISTEN to LIDAR (Figure 6). It then checks whether the next scan angle in the *_scan* table is equal to 360, which indicates the end of the sweep. If this is not the case, the state machine moves to ECHO, where it waits for an update of the input value Q, signaled by u. The *_scan* table is a constant list of angles that define the positions to which the lidar sensor must be directed during a sweep. The sequence is terminated by the value 360, which serves as a sentinel indicating completion. For instance, in the real robot, the table is defined as $\langle 15, 30, 45, 60, 75, 90, 40, 0, -15, -30, -45, -60, -75, -90, -40, 360 \rangle$.

Upon receiving a positive pulse (*true*) on u, the state machine transitions to RAY to send data to the upper level via output M. From RAY, it proceeds to RESUME to await a RESUME command (OP = 3) from the upper level. From RESUME, it transitions to ECHO or DONE, depending on whether the next angle differs from 360.

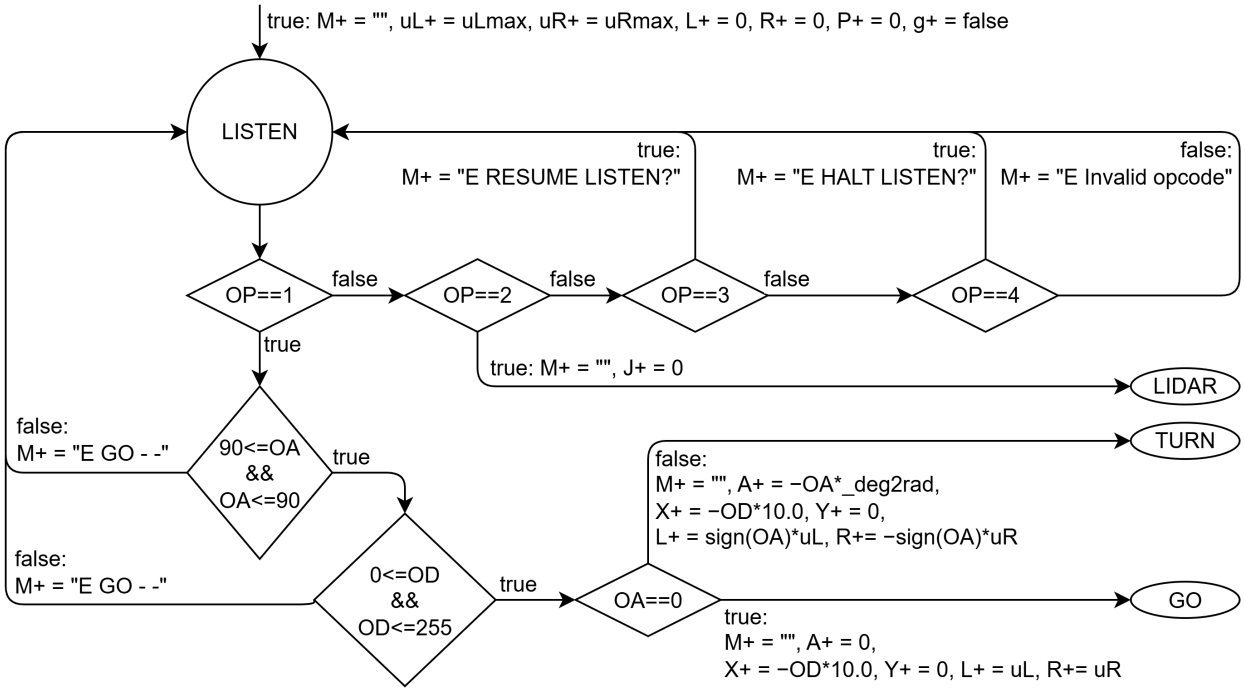

Figure 5. The L0Main controller starts in the LISTEN state, which also serves as its idle state. Upon reception of a valid command, it transitions either to LIDAR to initiate a lidar sweep or to GO or TURN to initiate motion toward a relative polar position.

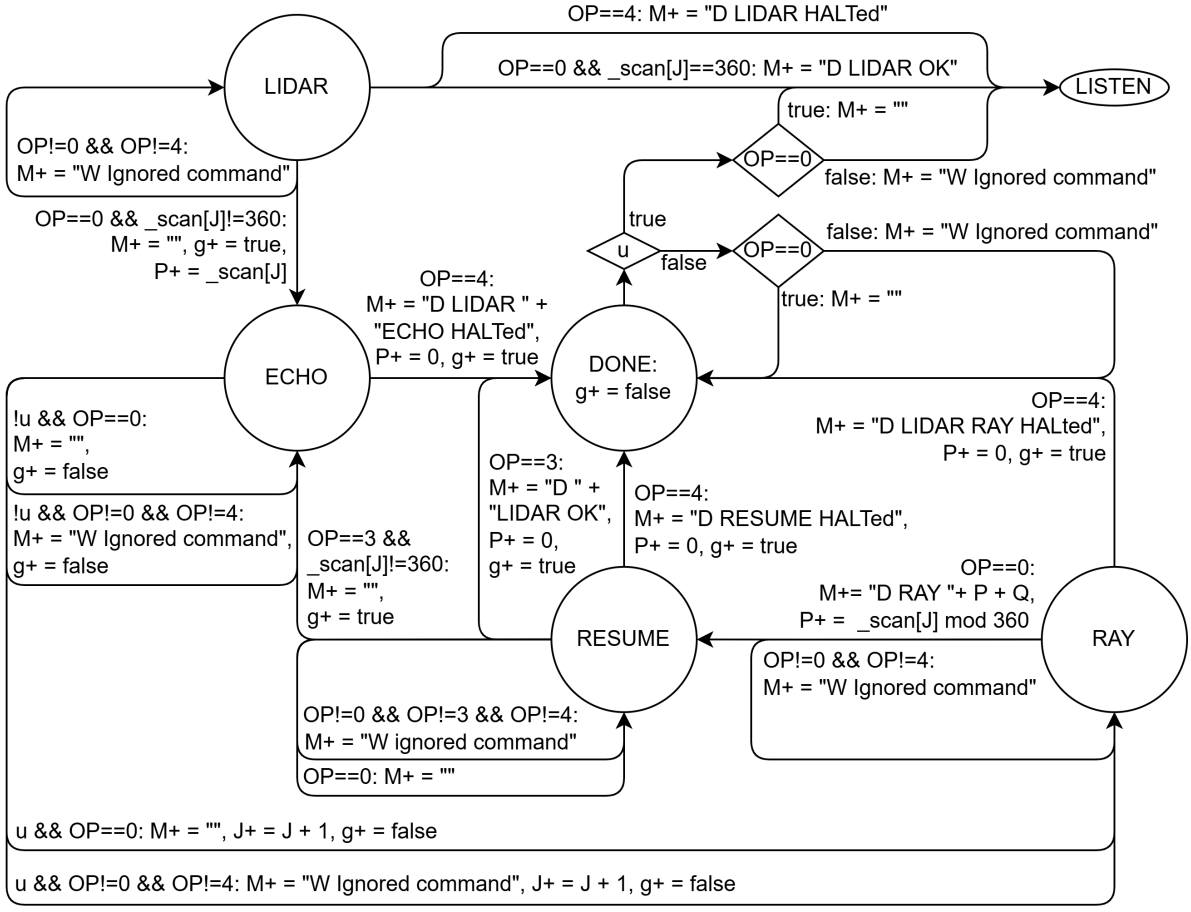

Figure 6. Section of the L0Main controller EFSM for LIDAR sweep operations.

Since the angle list in *_scan* may not end with 0, if *_scan[J]* is 360, L0Main instructs L0Lidar to point forward by setting P to 0 and transitioning to DONE, waiting for the next lidar update confirming forward orientation. Note that $P^+ = 0$ in transitions from ECHO, RAY, and RESUME to DONE ensures the lidar returns to its initial setting.

The other L1 command is GO, causing L0Main to switch to TURN or GO states (Figure 7). For GO with non-zero angle, it first goes to TURN, then to GO if distance is positive. For GO with only non-zero distance, it moves directly to GO. Transitions to TURN and GO also set variables L and R to uL and uR values for left/right turns or forward movement.

Initially, uL and uR are set to *uLmax* and *uRmax* and remain unchanged in this on–off controller mode. However, Section 5 presents a better control strategy where these values adjust to refine robot movement and minimize final pose errors.

In the TURN state, it updates the angle A and the (X, Y) position errors with respect to the goal, which is defined at the polar coordinates $(0, 0)$. The function **dist0**(X, Y, A, dA, dS) computes the distance error to the goal. The inputs dA and dS represent the angular and linear displacement increments since the previous cycle, and are obtained from the left and right encoder count variations, dL and dR. Because state updates take effect in the subsequent cycle, and because assignments and conditions may be evaluated in any order in the model, the values corresponding to the next cycle, such as $(A + dA)$ for the angle A, must be explicitly computed in every expression that relies on them. Following this example, the term $(A + dA)$ is therefore used not only in the expressions that compute the next values of X, Y, and A, but also in the conditions associated with the outgoing arcs from the TURN state.

The state machine leaves TURN when the next angle $A + dA$ magnitude falls below angle tolerance *_atol*. At that point, it returns to LISTEN if the remaining distance to the goal is within the tolerance *_stol*, or moves to GO to start forward movement otherwise. In the first case, L and R are set to 0 to stop the vehicle; in the second, they are set to uL and uR for forward motion.

In GO, variables A, X, and Y are updated, and next values determine if the final position is reached within tolerance *_stol* on X (negative). To prevent collisions, the distance to front obstacles Q must exceed safe distance *_sdist*. If _sdist $\geq Q$, the state machine transitions from GO to BLOCKED, checking if the front obstacle moves away within *_tout* seconds. If Q exceeds *_sdist* before timeout, it returns to GO; otherwise, it moves to LISTEN.

From TURN, GO, and BLOCKED, the state machine returns to LISTEN when $I[0] = 4$ (HALT), setting L and R to 0 and M to an appropriate text message. Commands other than HALT receive warning messages indicating command ignorance.

To simplify the above explanation, L and R are constant in both TURN and GO states. In practice, they would have to be modified every cycle to account for motion errors, especially when moving forward. Thus, L and R would have to be updated every cycle with the same values of uL and uR, which would be calculated to compensate for any motion errors. Obviously, in transitions to BLOCKED or LISTEN, L and R would become 0. The next section will show how to update uL and uR in discrete control modes that simulate continuous control.

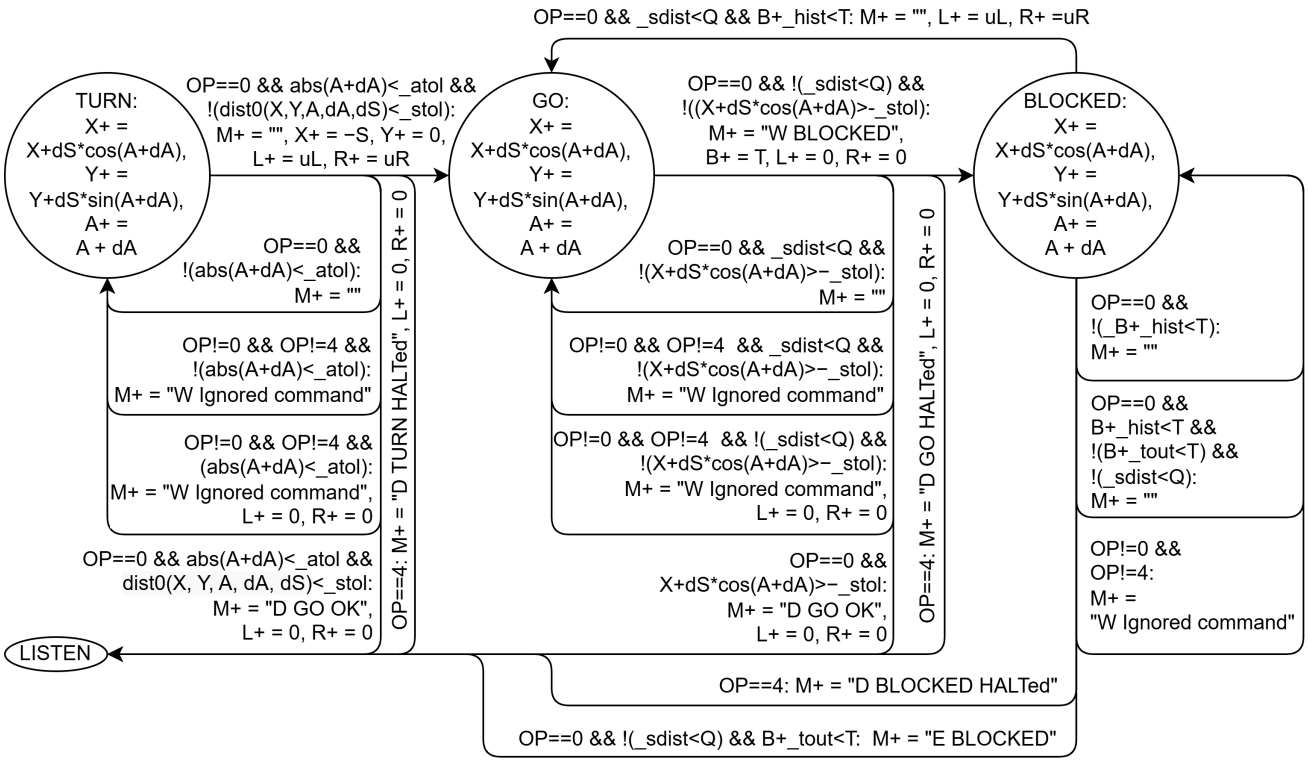

Figure 7. Section of the L0Main controller CEFSM for locomotion.

Formal Verification of EFSM Models

The presented EFSM models are constructed to ensure determinism, completeness, and absence of deadlocks and livelocks.

In the case of CEFSM networks, formal validation becomes more challenging, as the resulting composed EFSM exhibits a state-space explosion. Consequently, validation is primarily carried out through simulation-based analysis. For this reason, the discussion in this section focuses on individual EFSMs rather than on their concurrent composition into CEFSM networks.

Determinism is guaranteed by defining mutually exclusive transition conditions for each outgoing arc from a state. Completeness is achieved by ensuring that at least one transition is enabled under all valid input conditions. Potential indeterminacy or incompleteness conditions correspond to specification errors and are handled through explicit fallback transitions leading to safe states (e.g., __STOP__), which cause a reset in the system.

Deadlock freedom is enforced by construction: every state has at least one outgoing transition. Livelock conditions are avoided by ensuring progress, either through reduction in control error (in locomotion) or completion of handshake protocols (in lidar operations). It should be noted, however, that errors in sensor signal values may lead to livelock situations not captured by the EFSM diagrams. For example, the system may remain indefinitely in the ECHO state if the input E never becomes positive for L0Lidar, or if there are no changes in the rotary encoder inputs and dL and dR remain equal to zero in L0Main. To address these cases, timeout conditions could be introduced in all CEFSM loops, although this would significantly complicate their representation. A more suitable alternative is to introduce an additional CEFSM responsible for monitoring the progress of the other state machines and reinitializing them if repeated state patterns are detected over an excessive period. In this regard, it should be noted that, while the timeout for L0Lidar may be fixed, the timeout for L0Main should depend on the expected amount of motion (either rotational or linear) determined by L and R within a given dt.

Reachability analysis is performed by inspecting the connectivity of the EFSM graph and ensuring that all operational states are reachable from the initial state. Additionally, unreachable or sink states are treated as specification errors.

In terms of practical validation, the following methods are applied:

- Static inspection of EFSM diagrams to verify completeness and determinism.
- Runtime assertions embedded in the implementation to detect invalid transitions, for example by counting the number of transitions activated per cycle.
- Execution trace analysis to detect unexpected state or signal sequences, including verification of handshake signals to ensure correct synchronization between modules.

This validation approach, while lightweight, provides sufficient guarantees for embedded applications and aligns with the design goal of minimizing framework complexity.

5. Movement Control

Commands $\langle GO(OP = 1), \alpha, r \rangle$ transform into an α-degree rotation (positive to the right) followed by an r-cm forward movement. This simplifies robot position tracking and control. Variables A and X store angle and X-distance errors in radians and mm, respectively. Initially zero, they add $-\alpha$ and $-r$ upon receiving a GO command. Variable Y stores Y-distance error.

The control strategy drives A and X to zero and maintains Y at zero within a specified tolerance by adjusting the output variables L and R, which represent the DC control signals applied to the left and right motors, respectively.

For clarity of representation, the L0Main model presented in the previous section adopts an on–off control strategy in which the values of L and R are either $uLmax$ and $uRmax$, respectively, or zero. The variables uL and uR are initially set to their maximum values and remain unchanged throughout the operation. As a consequence, during the forward-motion phase (state GO) it is not possible to correct vehicle drift, nor to adjust the velocity to reduce the final positioning error.

This section enhances the on–off control system into a discrete control strategy by updating uL and uR at each cycle, and by setting the outputs L and R to identical values in the GO state.

Most control strategies aim to compute control inputs that drive the system error to zero as quickly as possible while minimizing energy consumption. In the present case, this requirement translates into selecting the control inputs uL and uR to be as large as possible, subject to stability constraints that prevent error growth and ensure that neither the target position nor the target orientation is overshot by more than half of the allowable tolerance.

Accordingly, it is necessary to characterize the relationship between the control inputs uL and uR and the resulting evolution of the state variables X, Y, and A.

In the remainder of the section, we first derive the incremental position and orientation updates based on the motion measured during the previous discrete-time control cycle, as obtained from the left and right wheel rotary encoders. Once the vehicle pose—expressed relative to the target frame and therefore representing the state error to be regulated—is known, suitable values of the control inputs uL and uR can be computed either analytically or experimentally, for instance by identifying the parameters of a PID controller model.

For the sake of simplicity, the rotational component of the motion (TURN state) is addressed using a conventional control strategy, whereas the residual orientation error is incorporated into the forward-motion phase (GO state). During this phase, closed-loop corrections are applied by modulating the control inputs uL and uR.

To determine their values based on pose error each iteration, left and right wheel movements in mm, dL and dR, are computed as:

$$(dL, dR) = (cL, cR) \cdot 2\pi \cdot rW/CPR \tag{1}$$

where (cL, cR) denote the positive or negative increments in encoder counts since the previous reading, rW represents the wheel radius, and CPR denotes the counts per revolution, which are 100 mm and 1440, respectively, in our case.

These values transform into dA and dS, containing average angle variation in radians and average forward movement in mm:

$$dA = (dR - dL)/dW \tag{2}$$

$$dS = (dR + dL)/2 \tag{3}$$

where dW is the distance between wheels (20.1 mm, in our case).

At each cycle, the current position (X^+, Y^+), the yaw angle (A^+), and the target distance (S^+) are calculated from the last position of the vehicle with respect to the target (X, Y), the last yaw angle A, and dA and dS (see Figure 8):

$$A^+ = A + dA \tag{4}$$

$$(X^+, Y^+) = \left(X + dS \cdot \cos(A^+), Y + dS \cdot \sin(A^+)\right) \tag{5}$$

$$S^+ = \sqrt{(X^+)^2 + (Y^+)^2} \tag{6}$$

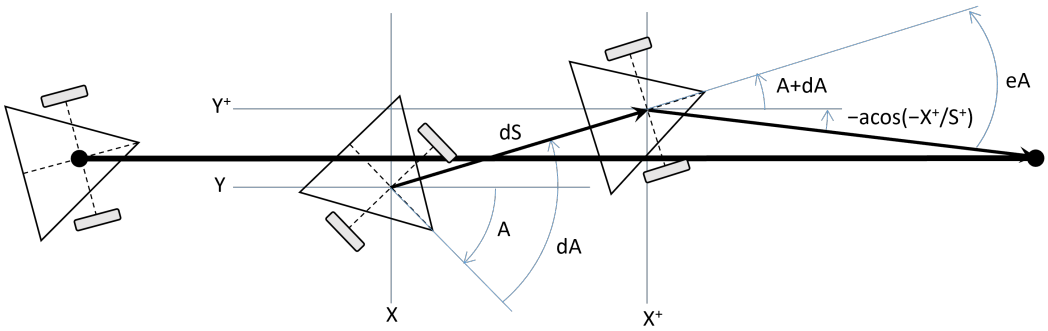

Figure 8. Computation of the yaw angle error eA from the known position (X, Y), orientation A, and the increments in angle dA and linear displacement dS. The point (X^+, Y^+) and the angle $A^+ = A + dA$ represent the updated pose, while S^+ denotes the updated distance to the goal.

Using these values, the left and right motor control signals, uL and uR, are computed to approach the target position and correct the heading, defined by the yaw angle A relative to the target vector.

The first phase of the motion (rotation) sets uL and uR to opposite values until A^+ from Equation (4) reaches zero within a specified tolerance (*_atol*).

The second phase moves the vehicle forward, progressively reducing the distance to the target, S^+ (Equation (6)). However, since A may not be exactly zero at the beginning of this phase and small differences in motor speeds can cause drift, an error correction mechanism is required.

The angular error eA, which compensates for yaw deviations with respect to the target direction, can be computed from the updated pending yaw A^+ in Equation (4) and from the geometric relationship between X^+ and S^+:

$$eA = A^+ - \cos^{-1}\left(-\frac{X^+}{S^+}\right) \tag{7}$$

Given that the motor speeds are proportional to uL and uR, and that these signals have produced a pose variation (dS, dA) expressed in polar coordinates, the updated control inputs (uL^+, uR^+) are computed so as to generate a forward displacement dS and an orientation change that compensates for the heading error eA:

$$(uL^+, uR^+) = \left(\frac{uL}{dL} \cdot (dS + dW \cdot eA),\ \frac{uR}{dR} \cdot (dS - dW \cdot eA) \right) \tag{8}$$

where uL^+ and uR^+ are limited to range $[uMin, uMax] = [40, 120]$, the minimum for DC motors to move the robot and the maximum to limit vehicle speed.

By applying Equation (2) to Equation (8), the latter can be rewritten as:

$$(uL^+, uR^+) = \left(uL + dW \cdot \frac{uL}{dL} \cdot \left(eA + \frac{dA}{2} \right),\ uR - dW \cdot \frac{uR}{dR} \cdot \left(eA + \frac{dA}{2} \right) \right) \tag{9}$$

Note that when $eA > 0$, the robot should turn right, implying that the right motor should run slower than the left one, and vice versa for $eA < 0$. This behavior is reflected in uL^+ and uR^+, which include a term proportional to eA and dA with opposite signs.

Special care must be taken when the linear motion increments dL or dR approach zero, as this may lead to numerical instability. In practice, since the ratios uL/dL and uR/dR can be assumed approximately constant and equal, they are replaced by fixed representative values to avoid this issue.

With this control strategy, the error would be canceled within a single control cycle, which would drive uL and uR to out-of-range values or, at best, to excessively aggressive corrections. To smooth the control action, the corrections should be distributed and applied over multiple cycles. One way to achieve this is by introducing a scaling (dividing) factor γ to eA. For example, with $dt \approx 0.03$ s (31 ms), the robot can change its orientation by approximately 1° ($\pi/180$ rad) per control cycle. Consequently, the angular error must be corrected over a number of cycles proportional to its magnitude. For instance, an error of 8° requires at least 8–9 control cycles for correction, since, in addition to the nominal correction rate, the effect of accumulated error during operation must also be taken into account.

Considering that the corrections applied to uL and uR can be both positive and negative and that, during the second phase, the robot must move forward, the reference value is set to $(uMax + uMin)/2$ in order to allow the maximum variation in both directions. These values are therefore used as the initial conditions to compute the first updated control inputs (uL^+, uR^+) according to Equation (9).

Although the control law in Equation (9) is derived from incremental pose variations, it can be interpreted as a proportional–derivative (PD) controller acting on the angular error with respect to the target direction. In particular, the term proportional to the angular error eA provides the proportional action, whereas the inclusion of the incremental orientation change dA introduces an implicit derivative action that damps oscillations and improves stability. Consequently, the proposed control strategy can be regarded as a discrete-time PD controller formulated directly in terms of measurable pose errors, without requiring an explicit state-space representation.

The expressions for the PD controller resemble those of Equation (9) and can be written as:

$$(uL^+, uR^+) = \left(\frac{uMax + uMin}{2} + u,\ \frac{uMax + uMin}{2} - u \right) \tag{10}$$

where

$$u = Kp \cdot eA + Kd \cdot \frac{\Delta eA}{dt} \tag{11}$$

$$Kp \approx \frac{1}{\gamma} \cdot dW \cdot \frac{uL}{dL} \tag{12}$$

$$Kd \propto dW \cdot \frac{uL}{dL} \tag{13}$$

Here, ΔeA denotes the variation in the angular error with respect to the previous control cycle, and dt is the control period. The controller gains were determined experimentally, yielding $Kp = 150$ and $Kd = 0.001$.

The initial estimate for Kp was obtained from Equation (12), under the assumption that both motors exhibit similar characteristics and operate in a linear region, such that the ratios uL/dL and uR/dR can be considered constant and approximately equal. The sensitivity of the controller to the derivative term, which depends on dt, is regulated by the gain Kd. To ensure controller stability, dt must remain constant. In our implementation, the computation time is supplemented with idle time up to dt seconds to enforce a fixed control period. The gains Kp and Kd have been identified to minimize the overall error for movements in the range of 10 to 40 cm and turns of up to $\pm 90^{\circ}$, which correspond to the typical operating conditions.

An integral term could be incorporated, if required, to compensate for steady-state errors; however, for the considered application, the proportional–derivative structure was found to be sufficient to ensure convergence to the target within the prescribed tolerances.

The stability of the proposed controller is analyzed under the assumption of a constant sampling period dt, together with actuator saturation limits and limited wheel slip. Under these conditions, the controller behaves as a discrete-time PD system with damping, resulting in a stable evolution of the pose error. From the discrete-time kinematic model, the heading error dynamics under the proposed control law can be approximated by a second-order linear difference equation. For positive Kp and non-negative Kd, and under bounded actuator saturation, this formulation leads to a progressive reduction in the angular error. Since the lateral and positional errors are directly coupled to the heading error, this behavior implies convergence of the full pose error to zero within the specified tolerances. While a formal proof of stability has not been included, the assumption of a constant sampling period simplifies the system dynamics, and experimental observations consistently confirm stable behavior for the selected gains.

To minimize sudden stops, *uMax* should reduce as the robot approaches the goal or obstacles. Here, *uMax* decreases linearly with the distance to nearby obstacles between $2 \cdot$ *_sdist* (braking distance) and *_sdist* (safety distance), from its nominal maximum value down to *uMin*. This corresponds to a proportional scaling between these two limits. Values below *uMin* cannot move the vehicle, so the transition from *uMin* to 0 is seamless.

This behavior has been consistently confirmed through experimental validation, both during the development phase and over several years of continuous operation of multiple robots in the laboratory.

6. Programming

Implementing the controller stack involves synthesizing software from its computational model. In this case, a network of concurrent EFSMs organized in three layers is transformed into Lua and Arduino Sketch programs. The full controller stack is implemented in the CoppeliaSim simulator, and L0 is implemented in Arduino Sketch for the robot's Arduino-compatible processor.

The embedded L0 controller (shaded part in Figure 9) communicates with upper levels via a Bluetooth serial profile link. An additional layer, L0L1Link, makes L1 communication transparent regardless of whether one robot (virtual or real) or both are used. L1 messages are sent to both, and L0 messages are synchronized with priority given to real-world messages. (More sophisticated strategies are possible but beyond this work's scope.)

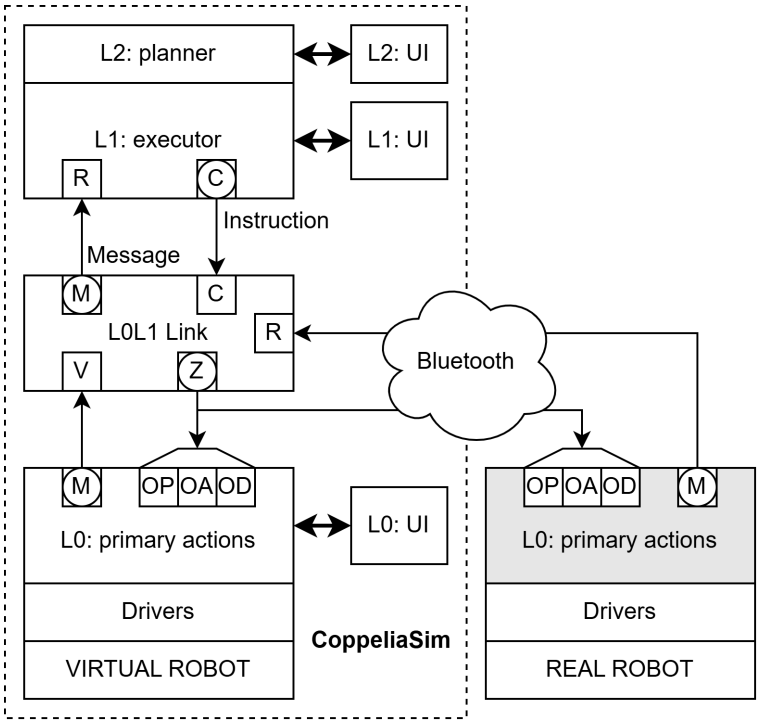

Figure 9. Controller stack architecture, including a robot digital twin and the L0L1Link interface layer for synchronizing messages between virtual and real L0 instances. The L0L1Link module broadcasts commands from L1 (input signal *C*) to the robots (output *Z*) and merges inputs *V* and *R* into messages *M* for L1. The driver layer and L0 are embedded in the physical robot and also have a virtual counterpart, in which the drivers interact with the simulated environment.

The L0 level is made up of two modules, L0Main and L0Lidar, whose software is synthesized separately. Each one contains the definition of the corresponding class, with private variables for EFSM input signals, state, model variables, and outputs (lines 17 to 28 in Listing 1) and methods to execute each of the operations it must perform to evolve in time (lines 4 to 12 in Listing 1): **set** instantiates the object according to its configuration parameters; **init** initializes the extended state and the immediate outputs; **write_outputs** transfers the contents of the output signals to the devices they have connected, be it a program window in the case of an interactive simulation or microcontroller ports in the case of the real robot; **read_inputs** obtains the data for the input signals, with the values of the internal signals as parameters, obtained via **get_*signal_name*** methods of the objects where the signals originate; **assign_delayed** determines the values of the extended state (main state and complementary variables) for the next state; **assign_immediate** makes the next state the current one, thus completing the assignment of the values of the next extended state to the current extended state and determines the values of the immediate outputs, and, finally, **is_active** returns whether the current state is valid and not a sink one. (A *sink state* is a state corresponding to a node with no outgoing arcs.)

Listing 1. EFSM header file for L0Lidar.

```
1  class L0Lidar {
2    public:
```

```
3       L0Lidar( ... )
4       void set() { ... }
5       void init();
6       bool is_active();
7       void write_outputs();
8       long get_Q(); // distance
9       bool get_u(); // update
10      void read_inputs(float angle, bool request);
11      void assign_delayed();
12      void assign_immediate();
13    private:
14      char _SERVOPIN; Servo _Servo; VL53L0X  _Sensor;
15      float    _wrot; // Rot. speed [deg/s]
16      int      A;     // Target angle [deg]
17      unsigned E;     // Obstacle dist. [mm]
18      enum {S_ROTATE, S_PING, S_ECHO, S_SINK} state[2];
19      int   Y[2]; // Yaw
20      float Z[2]; // Servo position
21      float B[2]; // Begin time
22      float D[2]; // Delay
23      float Q[2]; // Distance to obstacle
24      bool  u[2]; // Distance update
25      bool  s[2]; // Send ping
26      int   Y2Z(int A);
27  };
```

All variables representing the extended state (the state itself and other model variables) are two-element arrays, the first holding the current iteration value (_CURR_) and the second the next cycle value (_NEXT_). Thus, delayed assignments first set the _NEXT_ item value, and the immediate assignment phase copies this value to the _CURR_ field. (Immediate outputs are set according to current model variable values, including the state variable.)

Distinguishing _CURR_ and _NEXT_ values prevents errors from incorrect instruction execution order in **assign_delayed**. For example, swapping lines 9 and 10 in Listing 2 would make variable *D* always zero because *Y* would be set to *A* before computing *D*'s new value. This guarantee requires a second method to update all EFSM variables and output signals: **assign_immediate**, partially shown in Listing 2.

Listing 2. Delayed and immediate assignments in L0Lidar.

```
1  void L0Lidar::assign_delayed() {
2    if (state[_CURR_]==S_ROTATE) {
3      if (T-B[_CURR_]>=D[_CURR_]) {
4        Z[_NEXT_] = Y2Z(360); state[_NEXT_] = S_PING;
5      } // if
6    } else if (state[_CURR_]==S_PING) {
7      u[_NEXT_] = false;
8      if (A!=Y[_CURR_]) {
9        B[_NEXT_] = T; D[_NEXT_] = abs(A-Y[_CURR_])/_wrot;
10       Y[_NEXT_] = A; Z[_NEXT_] = Y2Z(A);
11       state[_NEXT_] = S_ROTATE;
12     } else { // A==Y[_CURR_]
```

```
13        s[_NEXT_] = true; state[_NEXT_] = S_ECHO;
14      } // if
15    } else if (state[_CURR_]==S_ECHO) {
16      s[_NEXT_] = false;
17      if (E>=0 ) {
18        u[_NEXT_] = true; Q[_NEXT_] = E; state[_NEXT_] = S_PING;
19      } // if
20    } else { // error handling
21    } // if chain
22  } // L0Lidar::assign_delayed()
23  void L0Lidar::assign_immediate() {
24    state[_CURR_] = state[_NEXT_];
25    Y[_CURR_] = Y[_NEXT_]; Z[_CURR_] = Z[_NEXT_];
26    // (... omitted ...)
27  } // L0Lidar::assign_immediate()
```

The methods **read_inputs** and **write_outputs** are platform-specific and must be programmed to read data into state machine inputs or take outputs from them and perform actions like sending serial port messages. Outputs intended for other state machines in the same system require **get_*signal_name*** methods, called to supply arguments to other state machines' **read_inputs**. In this case, L0Lidar has two inputs from L0Main specifying the lidar target angle and an explicit request for the lidar sweeping command execution, and L0Main has two inputs from L0Lidar: obstacle distance (Q) and update status (u).

This code organization enables direct binding between graph elements and program instructions, making programming systematic and easily automatable.

The version of L0 running on CoppeliaSim takes advantage of the simulation engine, which calls each of the methods specified in the different callback functions (sysCall_init, sysCall_actuation, sysCall_sensing, sysCall_cleanup) it calls at each simulation phase. Thus, as shown in Figure 10, during the initialization phase, **set** and **init** are called; during the actuation phase, **write_outputs** is called; and during the sensing phase, the remaining methods are called in sequence. In this case, if any **is_active** function returns false, either the modules are reinitialized or the simulation is stopped, depending on CO (for "continuous operation") being initially set to *true* or *false*, respectively.

Listing 3 shows an Arduino Sketch fragment simulating L0 state machines following this procedure. After instantiating both objects, the setup function calls their **set** and **init** methods to create necessary data structures and objects and initialize them appropriately.

Listing 3. Simulation engine in Arduino Sketch.

```
1  #include ''L0Lidar.h''
2  #include ''L0Main.h''
3  L0Lidar Lidar(LIDAR_SERVO);
4  L0Main Main(M1DIR, M1PWM, /* ... */);
5  void setup() {
6    Lidar.set(); Main.set();
7    Lidar.init(); Main.init();
8  } // setup
9  unsigned long dt = 31; // [ms] measured average execution time
10 void loop() {
11   unsigned long starttime = millis();
12   Lidar.write_outputs(); Main.write_outputs();
13   Lidar.read_inputs(Main.get_P(), Main.get_g());
```

```
14    Main.read_inputs(Lidar.get_Q(), Lidar.get_u());
15    Lidar.assign_delayed(); Main.assign_delayed();
16    Lidar.assign_immediate(); Main.assign_immediate();
17    if(!Lidar.is_active()||!Main.is_active()) {
18      Lidar.init(); Main.init();
19    } // if
20    while (millis()-starttime<dt) {;} // idle wait
21  } // loop
```

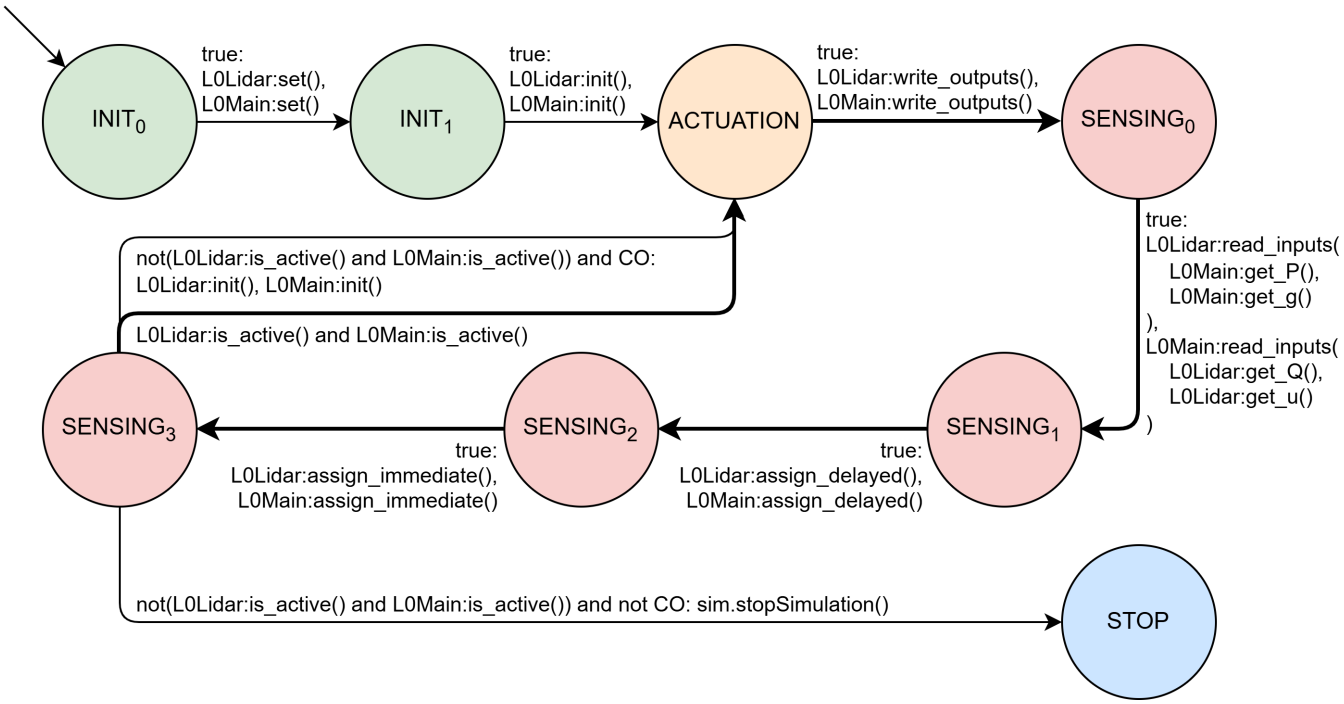

Figure 10. Execution model of L0 on CoppeliaSim. State names correspond to the simulation engine callback functions, and subscripts indicate the order of method calls within each phase. The thick arrows depict the simulation cycle. Commands are expressed in the Lua scripting language used by the simulator. This EFSM is executed by the CoppeliaSim engine which, in turn executes the L0 network of CEFSMs.

The loop function implements the cycle from Figure 10 for each CEFSM. Like the figure's behavior, if any state machine becomes inactive, it resets internal data by calling its **init** method and resumes the loop. More sophisticated actions like activating error handling could be taken when state machines become inactive due to failures.

The final *while* ensures no cycle runs shorter than *dt* milliseconds, preventing over-sampling issues like excessive power consumption from input/output operations. The value of *dt* has been experimentally determined as the average execution time of the control cycle, specifically considering the transitions within the TURN and GO states and between them. This selection facilitates the identification and tuning of the PD controller parameters, ensuring consistent dynamic behavior and contributing to the stability of the control loop.

The controller evolution is inherently described as a discrete-time sequence with fixed sampling period *dt*, where all state variables and control signals are synchronously updated at each cycle.

The code resulting from this process operates correctly in both the Lua implementation and the embedded C++ version. Although high precision is not a strict non-functional requirement, robot motion using on–off controllers proved to be poorly accurate, leading to excessive accumulation of positioning and orientation errors over time.

In contrast, the PID-based implementation provides significantly higher accuracy and is sufficient for the intended robot applications. Nevertheless, it must be noted that the initial, unoptimized codebase consisted of approximately 700 lines of code (LoC), occupying about 60 KB of program memory and 8 KB of data memory on an Arduino-compatible PJRC Teensy board, with an average execution cycle of 31 ms.

Although the PID controller was tuned to achieve satisfactory performance over movement ranges of 10 to 40 cm with a nominal *dt* of 31 ms, this average cycle time may occasionally increase due to execution-time variations, for instance when handling interrupts unrelated to the control process. For this reason, a shorter control cycle time is preferable, allowing idle time to be added as needed to enforce a stable period of 31 ms, which motivated an optimization effort. The optimized version resulted in a 600-line program, without arrays or global variables, using 58 KB of program memory and 7 KB of data memory, and achieving an average execution cycle time of 12 ms.

The reduction in execution time also provides additional computational margin within each control cycle. This margin has been exploited to incorporate lightweight online monitoring of selected internal variables and state transitions, enabling improved observability of the controller behavior during operation.

Experimental Evaluation

To illustrate the differences between the control methods and assess the trade-off between complexity and movement accuracy, a simple experiment was designed in which the robot was required to follow a trajectory describing a square with sides of 20 cm and return to its initial position and orientation. Figure 11 shows the result for each control method. As expected, the PD controller achieves superior performance. The figure presents only the final positions and orientations reached after each command in the sequence, which corresponds to $\langle\langle 1\ 0\ 20\rangle, \langle 1\ 90\ 20\rangle, \langle 1\ 90\ 20\rangle, \langle 1\ 90\ 20\rangle, \langle 1\ 90\ 0\rangle\rangle$.

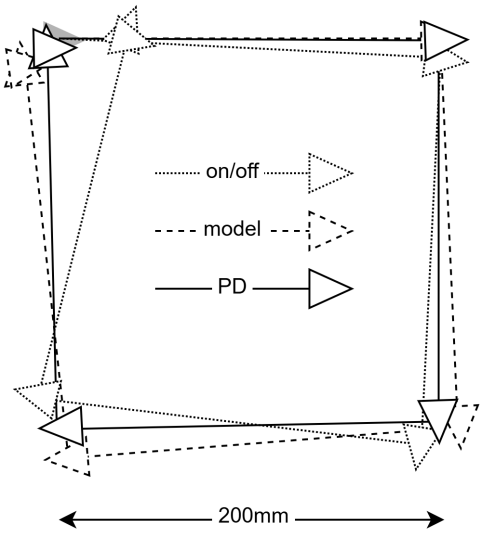

Figure 11. The experiment consisted of commanding the robot to follow the sides of a 20 cm square and return to its initial position.

Although our robots are intended for educational purposes or applications where topological location is more relevant than geometric precision, and therefore no strict accuracy requirements were imposed, it is desirable for the robot to operate with centimeter-level positional accuracy and degree-level orientation accuracy, consistent with the resolution of the command instructions. Under these criteria, only the PD-based strategy approaches the desired performance.

Table 1 summarizes the results obtained with the three control strategies. The model-based approach uses Equation (9) with central values for uL and uR, $(uMin + uMax)/2$. As expected, this approach improves upon the on–off strategy; however, since it does not account for non-ideal robot behaviors, its performance remains inferior to that of the PD-based strategy, in which the parameters have been tuned to achieve near-optimal performance across different operating conditions.

Table 1. Comparison of control strategies.

Control Strategy	Position Error [mm]	%	Orientation Error [deg]	%
Required	≤ 40	≤ 0.5	≤ 4.0	≤ 1.0
On–off	40.7	5.1	15.1	4.1
Model-based	20.4	2.6	7.6	2.1
PD-based	4.4	0.6	1.4	0.4

To further illustrate the temporal evolution of the control process, Figure 12 shows the evolution of the control signals (uL and uR), the state transitions (with a fixed sampling period of dt = 31 ms), and the error variables over time during a representative maneuver. Such detailed inspection does not compromise real-time constraints, as the execution time remains safely below dt.

The figure focuses on the detailed chronological behavior of the proposed control strategy and is intended to illustrate the internal dynamics of the CEFSM-based control structure and its discrete-time operation. The timing analysis is based on the PD controller adopted in the final implementation. Comparative performance between control strategies is provided separately in Table 1. For the representative 20 cm motion used during evaluation, the PD controller produced less variable control signals than the model-based controller while achieving higher positioning accuracy, resulting in a clearer representation of the controller dynamics.

It should also be noted that variations in execution time are not associated with short or fast maneuvers, as these only affect the magnitude of the control signals without modifying the computational structure of the control loop. Additional execution steps may occur only during abrupt stops, which involve transitions to states such as BLOCKED or LISTEN. These transitions correspond to motion interruption and, therefore, do not affect the real-time requirements of the PD control loop. In contrast, consecutive GO-to-GO cycles follow identical execution paths, ensuring a constant and predictable cycle time during normal operation.

The relationship between the discrete control cycle, the L0Main states, and the progressive reduction in the control error can be clearly observed during a representative maneuver. For a $\langle 1\ (\text{GO}), 45, 20 \rangle$ instruction, the initial angular and distance errors are 45° and 200 mm, respectively. The two phases of this operation are clearly visible: in the TURN state, the control variables uL and uR have opposite values, the distance error S remains constant, and the angular error eA decreases toward zero. Then, in the GO state, uL and uR take positive values to drive the vehicle forward, and S decreases progressively. During this phase, XR and YR also decrease proportionally, as the robot approaches the goal along a 45° direction. The control signals become less aggressive as the vehicle approaches the goal and drop to zero when L0Main returns to LISTEN.

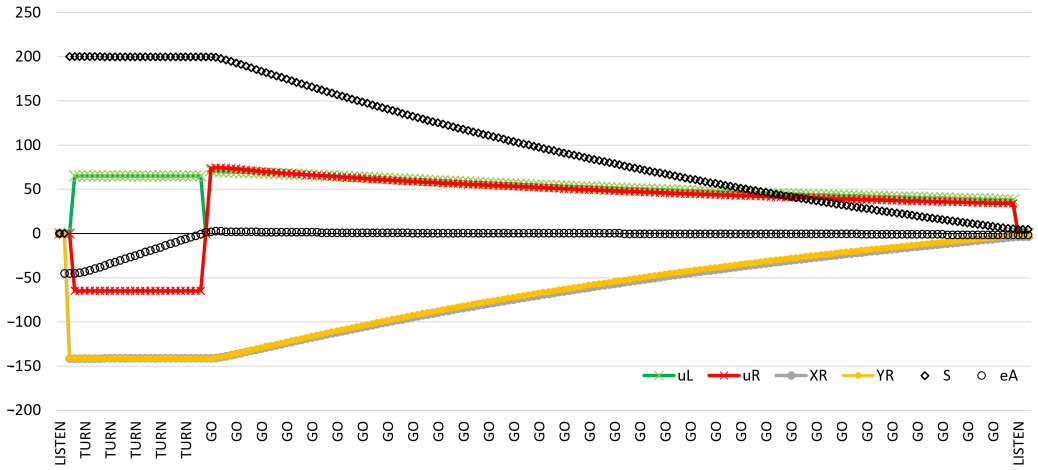

Figure 12. Timing diagram showing the evolution of the control process for a $\langle 1\,(\mathrm{GO}), 45, 20\rangle$ instruction, including the L0Main states, control signals (uL, uR), and error variables (S, distance to the goal, and eA, angular error) over time. Variables XR and YR represent the robot position with respect to the goal. The instruction is completed in 3008 control cycles of 31 ms.

7. Conclusions

A network of CEFSMs has been shown to offer a clear, modular, and implementation-friendly approach for structuring the reactive control layer of a differential-drive mobile robot. By explicitly modeling control logic as interacting state machines with minimal and well-defined semantics, the proposed approach enables clear behavioral specifications and a direct, systematic translation from diagrams to executable procedural code.

The controller architecture decomposes the reactive layer into two CEFSMs dedicated to locomotion and lidar operation, respectively, which interact through simple and deterministic protocols. This separation of concerns improves modularity, facilitates reuse, and allows individual components to be developed, tested, and refined independently. Unlike more complex hierarchical frameworks, the proposed solution avoids heavy runtime infrastructures and is therefore well suited for resource-constrained embedded platforms.

A key contribution of this work is the integration of a hybrid, discretized motion-control strategy within the CEFSM framework. The proposed approach approximates continuous behavior by performing discrete updates at each control cycle and incorporates a simplified PD controller derived from incremental pose measurements. This strategy enables closed-loop correction of motion errors during forward movement while remaining computationally lightweight and implementation-friendly. Experimental results show that this approach significantly improves positioning and orientation accuracy compared with simple on–off control schemes, while respecting actuator constraints and real-time execution limits.

Although the present work has been validated on a differential-drive platform, the proposed EFSM-based methodology is not restricted to this particular robot configuration. The separation between behavioral modeling and motion-control formulation allows the architecture to be extended to other mobile platforms, including omnidirectional and tracked robots.

In such cases, only the kinematic and control expressions used in the motion-control layer need to be adapted, while the EFSM structure, communication mechanisms, and synchronization protocols remain unchanged. This highlights the generality of the proposed approach and its applicability across different robotic systems with minimal redesign effort.

The methodology also demonstrates strong portability across platforms. The same behavioral models and programming templates are used to control both physical Arduino-compatible robots and their digital twins in the CoppeliaSim simulation environment, with only thin input/output layers requiring adaptation.

This portability, together with the clarity of the state-machine representations, has been validated through more than a decade of classroom deployment in an undergraduate Embedded Systems course. Long-term teaching experience shows that the method significantly reduces conceptual ambiguity and accelerates student learning, particularly with respect to concurrency, signal management, and time-sensitive control loops. Students gain practical experience implementing controllers on both simulated and physical robots, and course feedback has consistently highlighted the clarity of the approach and the usefulness of the programming templates.

Although translation from graphical representations into executable programs can be performed manually using the provided templates and examples, or automatically through specialized tools, we are currently exploring how AI can leverage these programming patterns to generate reliable code through appropriate prompting. We believe that AI-based tools can play an increasingly important role in supporting the design and development of robot controllers and that preparing future engineers to work effectively with these technologies is essential.

Limitations and Future Work

The scalability of the proposed approach is primarily constrained by the computational complexity of the low-level controllers integrated within the CEFSM framework. In particular, the inclusion of additional sensors or actuators, or components with higher degrees of freedom, increases the dimensionality of the control problem and the amount of computation required within each control cycle. As a result, the worst-case execution time (WCET) of the control loop may approach or exceed the prescribed control cycle time dt, thereby affecting the temporal determinism required for stable closed-loop operation.

From a real-time systems perspective, this limitation is not intrinsic to the CEFSM modeling paradigm but reflects a general constraint associated with embedded control systems. Addressing this issue depends on the specific application and may require algorithmic optimization, more efficient software architectures, or the use of more capable processing platforms.

From a modeling standpoint, the use of CEFSMs and the proposed implementation methodology do not restrict the structure or formulation of the underlying low-level controllers. In particular, the one-to-one mapping between diagrammatic elements and code constructs can, in principle, be preserved independently of the control strategy employed. However, as system complexity increases, the number of state variables, transition conditions, and auxiliary computational functions also grows. This growth is structural rather than conceptual, as the modeling paradigm itself remains unchanged. As a consequence, model compactness, readability, and maintainability may be negatively affected, especially in systems with tightly coupled dynamics or more complex non-holonomic constraints, where additional coordination between modules becomes necessary.

These observations suggest that scalability is not limited by the modeling formalism itself, but by the practical trade-offs between control complexity, execution-time constraints, and implementation clarity. Consequently, an important line of future work is the integration of more sophisticated low-level controllers within the reactive control layer, while preserving the clarity, modularity, and deterministic execution properties of the CEFSM-based representation.

Despite its advantages, the proposed approach also presents limitations at the system-integration level, particularly regarding scalability in distributed environments and sensitivity to timing assumptions, including actuator dynamics, sensor latencies, and communication delays. Extending the architecture to middleware-based platforms such as ROS2 would therefore require additional mechanisms to handle asynchronous communication and distributed execution.

Funding: This research received no external funding.

Data Availability Statement: Diagrams and software are available upon request.

Acknowledgments: The author thanks Jordi Guerrero for building the robot and conducting tests, and Ismael Chaile, Pragna Das, Joaquín Saiz, and Daniel Rivas for their contributions to various versions of the physical and virtual robots over the years.

Conflicts of Interest: The author declares no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ASM	Algorithmic State Machine
BT	Behavior Tree
CEFSM	Concurrent Extended Finite State Machine
CPR	Counts Per Revolution
EFSM	Extended Finite State Machine
FSM	Finite State Machine
HSM	Hierarchical State Machine
LoC	Lines of Code
PD	Proportional–Derivative
PID	Proportional–Integral–Derivative
ROS2	Robot Operating System 2
UML	Unified Modeling Language
WCET	Worst-Case Execution Time

References

1. Sauer, L.; Henrich, D.; Martens, W. Towards Intuitive Robot Programming Using Finite State Automata. In *Proceedings of the KI 2019: Advances in Artificial Intelligence: 42nd German Conference on AI, Kassel, Germany, 23–26 September 2019*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 290–298. [CrossRef]
2. Halim, J.; Eichler, P.; Krusche, S.; Bdiwi, M.; Ihlenfeldt, S. No-code robotic programming for agile production: A new markerless-approach for multimodal natural interaction in a human-robot collaboration context. *Front. Robot. AI* **2022**, *9*, 1001955. [CrossRef] [PubMed]
3. Girault, A.; Lee, B.; Lee, E.A. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **1999**, *18*, 742–760. [CrossRef]
4. Alur, R.; Kannan, S.; Yannakakis, M. Communicating Hierarchical State Machines. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*; Lecture Notes in Computer Science; Springer: Berlin, Germany, 1999; Volume 1644, pp. 169–178. [CrossRef] [PubMed]
5. Lee, E.A. Concurrent Models of Computation. Lecture Notes. Department of Electrical Engineering and Computer Sciences, University of California: Berkeley, CA, USA, 2007. Available online: https://embedded.eecs.berkeley.edu/concurrency/lectures/Lee07.pdf (accessed on 28 April 2026).
6. Samek, M.; Montgomery, P. State-oriented programming. *Embed. Syst. Program.* **2000**, *13*, 22–43.
7. Brunner, S.G.; Steinmetz, F.; Belder, R.; Dömel, A. RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Daejeon, Republic of Korea, 9–14 October 2016; pp. 3283–3290. [CrossRef]
8. Li, W.; Miyazawa, A.; Ribeiro, P.; Cavalcanti, A.; Woodcock, J.; Timmis, J. From Formalised State Machines to Implementations of Robotic Controllers. In *Distributed Autonomous Robotic Systems*; Springer Proceedings in Advanced Robotics; Springer: Cham, Switzerland, 2018; Volume 6. [CrossRef]

9. Miyazawa, A.; Ribeiro, P.; Li, W.; Cavalcanti, A.; Timmis, J.; Woodcock, J. RoboChart: Modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.* **2019**, *18*, 1–53. [CrossRef]
10. Tousignant, S.; Van Wyk, E.; Gini, M. An Overview of XRobots: A Hierarchical State Machine Based Language. In Proceedings of the ICRA-2011 Workshop on Software Development and Integration in Robotics, Shanghai, China, 9–13 May 2011.
11. Ribas-Xirgo, L. Simulator-Based Digital Twin of a Robotics Laboratory (Section on EFS2M Software Generation). *Machines* **2026**, *14*, 273. [CrossRef]
12. Rivas, D.; Das, P.; Saiz-Alcaine, J.; Ribas-Xirgo, L. Synthesis of Controllers from Finite State Stack Machine Diagrams. In *Proceedings of the IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*; IEEE: New York, NY, USA, 2018; pp. 1179–1182. [CrossRef]
13. Zhou, H.; Min, H.; Lin, Y.; Zhang, S. A Robot Architecture of Hierarchical Finite State Machine for Autonomous Mobile Manipulator. In *Intelligent Robotics and Applications*; Huang, Y., Wu, H., Liu, H., Yin, Z., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017; pp. 425–436. [CrossRef]
14. González-Santamarta, M.Á.; Rodríguez-Lera, F.; Fernández, C.; Martín, F.; Matellán, V. YASMIN: Yet Another State MachINe library for ROS 2. *arXiv* **2022**. [CrossRef]
15. Macenski, S.; Foote, T.; Gerkey, B.; Lalancette, C.; Woodall, W. Robot Operating System 2: Design, architecture, and uses in the wild. *Sci. Robot.* **2022**, *7*, eabm6074. [CrossRef] [PubMed]
16. Macenski, S.; Martín, F.; White, R.; Clavero, J.G. The Marathon 2: A Navigation System. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*; IEEE: New York, NY, USA, 2020. [CrossRef]
17. Colledanchise, M.; Ögren, P. *Behavior Trees in Robotics and AI: An Introduction*; CRC Press: Boca Raton, FL, USA, 2018. [CrossRef]
18. Poppinga, M.; Bestmann, M. DSD—Dynamic Stack Decider. *Int. J. Soc. Robot.* **2022**, *14*, 73–83. [CrossRef]
19. Levin, I.; Kolberg, E.; Reich, Y. Robot Control Teaching with a State Machine-based Design Method. *Int. J. Eng. Educ.* **2004**, *20*, 234–243.
20. Rustam, I.; Tahir, N.M.; Yassin, A.I.M.; Wahid, N.; Kassim, A.H. Linear Differential Driven Wheel Mobile Robot Based on MPU9250 and Optical Encoder. *TEM J.* **2022**, *11*, 30–36. [CrossRef]
21. Myint, C.; Win, N.N. Position and Velocity control for Two-Wheel Differential Drive Mobile Robot. *Int. J. Sci. Eng. Technol. Res. (IJSETR)* **2016**, *5*, 2849–2855.
22. Hirpo, B.D.; Zhongmin, W. Design and Control for Differential Drive Mobile Robot. *Int. J. Eng. Res. Technol. (IJERT)* **2017**, *6*, 327–334.
23. Manzo Hernández, J.P.; Solano Vargas, J.C.; Barrera Fernández, J.M.; Moreno Orduña, D.; Sánchez Saquín, C.H. A novel adaptive PID controller with new seesaw algorithms using alternative derivatives. *Syst. Sci. Control Eng.* **2024**, *12*, 2363625. [CrossRef]
24. Sun, Q.; Ren, H.; Duan, Y.; Yan, Y. The Adaptive PID Controlling Algorithm Using Asynchronous Advantage Actor-Critic Learning Method. In *SIMUtools*; Springer: Cham, Switzerland, 2019. [CrossRef]
25. Jafarzadeh, S.; Mirheidari, R.; Jahed Motlagh, M.R.; Barkhordari, M. Designing PID and BELBIC Controllers in Path Tracking Problem. In *Proceedings of the 2008 10th International Conference on Control, Automation, Robotics and Vision*; IEEE: New York, NY, USA, 2008.
26. Sharma, P.; Kumar, V. Design of Novel BELBIC Controlled Semi-Active Suspension and Comparative Analysis with Passive and PID Controlled Suspension. *Walailak J. Sci. Technol.* **2021**, *18*, 8989. [CrossRef]
27. Raju, G.; Srinivas, G.N. Intelligent Control Strategies for BLDC Motors in Electric Vehicles: Unveiling the Optimal Balance of Speed, Stability, and Efficiency. *Appl. Intell.* **2025**, *11*, 244. . [CrossRef]
28. Yamamoto, T.; Kurozumi, R.; Fujisawa, S. A Design of CMAC Based Intelligent PID Controllers. In *ICANN/ICONIP*; Springer: Berlin/Heidelberg, Germany, 2003.
29. Wakitani, S.; Ohnishi, Y.; Yamamoto, T. Design of a CMAC-Based PID Controller Using FRIT for Nonlinear Systems. In *SICE Transactions*; Springer: Berlin/Heidelberg, Germany, 2012.
30. Zhou, Q.; Liu, L.; Jiang, L.; Xu, Z. CMAC-PID Composite Control for the Position Control of a Fully Variable Valve System. *Int. J. Automot. Technol.* **2023**, *24*, 681–691. [CrossRef]
31. Goh, M.Q.; Ghazali, M.R.; Ahmad, M.A.; Mohd Shukri, M.S. Data-Driven Sigmoid Proportional-Integral-Derivative (SPID) Controller for Twin Rotor MIMO System. *J. Eur. Syst. Autom.* **2023**, *56*, 105–113. [CrossRef]
32. Mohd Tumari, M.Z.; Ahmad, M.A.; Suid, M.H.; Ghazali, M.R.; Tokhi, M.O. An Improved Marine Predators Algorithm Tuned Data-Driven Multiple-Node Hormone Regulation Neuroendocrine-PID Controller for Multi-Input–Multi-Output Gantry Crane System. *J. Low Freq. Noise Vib. Act. Control* **2023**, *42*, 1666–1698. . [CrossRef]
33. Ekinci, S.; Izci, D.; Tümen, V.; Jabari, M.; Çelik, E.; Elrashidi, A. Intelligent Control of Magnetic Ball Suspension Systems via a Novel Hyperbolic Tangent PID Controller Tuned by the Artificial Lemming Algorithm. *Biomimetics* **2026**, *11*, 205. [CrossRef] [PubMed]
34. Xiang, K.; Song, Y.; Ioannou, P. Nonlinear Adaptive PID Control for Nonlinear Systems. *IEEE Trans. Autom. Control* **2025**, *70*, 7000–7007. [CrossRef]

35. Dòria-Cerezo, A.; Biel, D.; Olm, J.M.; Repecho, V. Sliding mode control of a differential-drive mobile robot following a path. In *Proceedings of the 18th European Control Conference (ECC)*; IEEE: New York, NY, USA, 2019; pp. 4061–4066. [CrossRef]
36. Nurmaini, S.; Dewi, K.; Tutuko, B. Differential-Drive Mobile Robot Control Design based-on Linear Feedback Control Law. In *IOP Conference Series: Materials Science and Engineering, Volume 190, IAES International Conference on Electrical Engineering, Computer Science and Informatics, Semarang, Indonesia, 23–25 November 2016*; IOP: Bristol, UK, 2016. [CrossRef]
37. Tafrishi, S.A.; Ravankar, A.A.; Salazar Luces, J.V.; Hirata, Y. A Novel Assistive Controller Based on Differential Geometry for Users of the Differential-Drive Wheeled Mobile Robots. *arXiv* **2022**, arXiv:2202.01969v1. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.