

## Read the Manual: Grounding Behavior Tree Synthesis for Multi-Machine Factory Operation

Shek, Chak.Lam; Wang, Ye; Liu, Jing; Suzuki, Kei; Tokekar, Pratap; Koike-Akino, Toshiaki

TR2026-149    October 03, 2026

### Abstract

Industrial robot deployments are typically scripted for specific machines/tasks, an approach that does not scale to wide floors where robots must operate multiple machines with various procedures. Moreover, fixed scripts cannot readily adapt when a machine deviates from the expected procedure, whereas human operators can flexibly follow manuals that specify ordered steps, preconditions, and recovery actions. We present an agentic framework that grounds robotic machine operation in such manuals. The framework parses a manual into a memory representation and synthesizes a Behavior Tree whose condition nodes evaluate procedural preconditions against live machine state and whose action nodes invoke robot primitives. This representation enables the robot to detect failed preconditions and execute the corresponding manual-prescribed recovery actions while preserving the remaining procedure. We evaluate the approach in a factory environment containing three machines with physically actuated control panels, distinct operating procedures, and faults requiring physical recovery actions. The proposed method outperforms an ungrounded language-model baseline.

*IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) Workshop 2026*



# Read the Manual: Grounding Behavior Tree Synthesis for Multi-Machine Factory Operation

Chak Lam Shek<sup>1,2</sup>, Ye Wang<sup>2</sup>, Jing Liu<sup>2</sup>, Kei Suzuki<sup>2</sup>, Pratap Tokekar<sup>1</sup>, and Toshiaki Koike-Akino<sup>2</sup>

**Abstract**—Industrial robot deployments are typically scripted for specific machines/tasks, an approach that does not scale to wide floors where robots must operate multiple machines with various procedures. Moreover, fixed scripts cannot readily adapt when a machine deviates from the expected procedure, whereas human operators can flexibly follow manuals that specify ordered steps, preconditions, and recovery actions. We present an agentic framework that grounds robotic machine operation in such manuals. The framework parses a manual into a memory representation and synthesizes a Behavior Tree whose condition nodes evaluate procedural preconditions against live machine state and whose action nodes invoke robot primitives. This representation enables the robot to detect failed preconditions and execute the corresponding manual-prescribed recovery actions while preserving the remaining procedure. We evaluate the approach in a factory environment containing three machines with physically actuated control panels, distinct operating procedures, and faults requiring physical recovery actions. The proposed method outperforms an ungrounded language-model baseline.

## I. INTRODUCTION

Industrial robot deployments are typically mono-function: e.g., a manipulator loads a single press; an automated guided vehicle shuttles parts along one fixed route. They are hard-scripted once for the particular job and rarely repurposed [1], [2], [3]. This pre-programmed, single-task paradigm simplifies validation/certification, but it does not scale to shop floors where product mix changes frequently and a single robot must operate several machines with diverse procedures [4], [5], [6]. In this work we target this *multi-task* setting, in which a robot handles multiple machines and multiple job types rather than a single fixed routine.

Operating a machine is rarely a single action. It is a sequence of operations: check that a precondition holds, release an interlock, perform the actuation, and confirm the resulting state, in an order set by the machine’s operating manual. Human operators are trained to run unfamiliar equipment by reading and following this manual. This suggests a question for robot autonomy: rather than hand-coding each machine’s procedure into a robot controller, one job at a time, can this manual (written for a human operator) guide a robot through the same sequence of operations?

Answering this question requires a control structure that, given a manual and a task, can (1) determine whether a

machine’s current operating condition permits a given action (e.g., checking if the machine idle, whether the guard closed, or if the upstream buffer is empty), and (2) order the resulting sub-tasks so that the full operation sequence executes correctly/safely. Behavior Trees (BTs) fit this structure: their condition/action nodes map onto a manual’s preconditions and operating steps, and their reactive ticking semantics accommodate state changes at run time [7], [8], [9].

To generate such BTs from unstructured manuals without hand-encoding procedures, we use an LLM agent that reads the manual to emit the corresponding behavior, backed by a memory system that tracks machine/task state across long-running operation. A single flat BT does not scale to multiple robots acting concurrently, so we further propose a task allocation scheme extension that decomposes the generated behavior into subtrees assignable to individual robots and coordinates their access to shared machines and resources.

The main contributions of this work are:

- 1) An Isaac Sim-based multi-robot industrial task simulation and benchmark for long-running machine operation, covering three machines under multi-task and multi-robot requirements.
- 2) An agentic system that reads operating manuals and produces executable Behavior Trees, grounding robot behavior in manual-specified operating conditions and procedures. It completes 83.7% of the benchmark procedures without a hand-written code per-machine.
- 3) A task allocation scheme that decomposes a generated BT into coordinated subtrees for individual robots.

## II. PROBLEM FORMULATION AND PRELIMINARIES

### A. Multi-Robot Industrial Operation Setup

We consider a shop floor with a *homogeneous* pool of  $m$  robots  $R = \{r_1, \dots, r_m\}$ , each with an identical action repertoire (e.g., navigation, pick, place, and physical actuation of a machine interface), and a set of  $k$  industrial machines  $E = \{e_1, \dots, e_k\}$  (e.g., presses, conveyors, inspection stations). Unlike the robots, the machines are heterogeneous: each machine  $e_j$  is accompanied by its own, machine-specific manual  $M_j$ , since different machines are operated according to different procedures. Given a high-level task instruction  $T$  referencing one or more machines, robots from  $R$  must be dispatched – via the scheduling mechanism of Sec. III-D – to carry out the operation sequence specified by the corresponding manual(s), while correctly reacting to any machine alarm that may arise during execution.

\*All work was performed during the internship of Chak Lam Shek while he was employed at MERL.

<sup>1</sup>The authors are with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA.

<sup>2</sup>The authors are with Mitsubishi Electric Research Laboratories (MERL), Cambridge, MA, USA.

### B. Behavior Tree Preliminaries

A Behavior Tree (BT) organizes the switching structure of an autonomous agent as a tree of nodes that are *ticked*, starting from the root. A tick is a control signal propagated down through the tree that triggers each visited node’s execution logic and returns one of three statuses: *success*, *failure*, or *running* [7], [10]. This section reviews the node types and execution semantics used in this work.

**Root node.** The root is an entry point of the tree. It initiates the periodic ticking process and generates the successive ticks that drive the evaluation of the entire behavior.

**Leaf nodes.** Leaf nodes are the terminal nodes of the tree; they perform a specific function when ticked. *Condition* nodes evaluate a predicate over the agent’s state or environment (e.g., checking sensor data, battery level, or whether a machine’s safety guard is closed) and return success or failure. *Action* nodes trigger an action in the environment, such as moving a joint, picking up an object, or sending a command to a machine. In addition to success or failure, an action node may return running when execution has not yet completed, reflecting that some actions, such as a robot motion, unfold over multiple ticks.

**Control nodes.** Control nodes specify the execution logic of their children, i.e., how and in what order they are ticked. A *Sequence* node ticks its children from left to right: if any child fails, the Sequence fails and halts further execution; if the last child succeeds, the Sequence succeeds. If a child returns running, the Sequence also returns running, and on the next tick either resumes at the last running child or restarts from the first child, depending on whether it is implemented as a Sequence or a ReactiveSequence. A *Fallback* node (a mirror of Sequence, also called a Selector) ticks its children from left to right and succeeds as soon as one child succeeds, skipping the rest; if a child fails, the next child is ticked, unless it was the last one, in which case the Fallback fails. As with Sequence, a running child causes the Fallback to return running, resuming at either the last ticked child or the first child on the next tick, depending on whether it is a Fallback or ReactiveFallback. A *Parallel* node ticks multiple children simultaneously and succeeds or fails based on a success threshold, e.g., requiring either all or only some subset of its children to succeed.

Fig. 1 depicts these node types on a minimal example: before placing an object into a drawer, the agent must ensure the drawer is open, retrying the open action if needed, and only then execute the placement action. This is the condition-action structure that our agent (Sec. III) extracts from an operating manual and composes into an executable BT.

### III. METHODOLOGY

Figs. 2 and 3 show the two halves of our system: a simulated industrial environment that exposes machines through a physical-interface and finite-state abstraction, and an agentic system that reads the operating manual, maintains it in memory, and generates the BT that drives the simulator. At a high level, the operating manual is parsed and stored in a memory module; the agentic system’s Behavior Tree

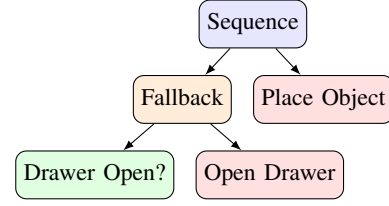


Fig. 1. A minimal Behavior Tree example. The root Sequence first ticks a Fallback that checks whether the drawer is open (a Condition node) and, if not, executes the Open Drawer action; only once this succeeds does the Sequence tick the Place Object action.

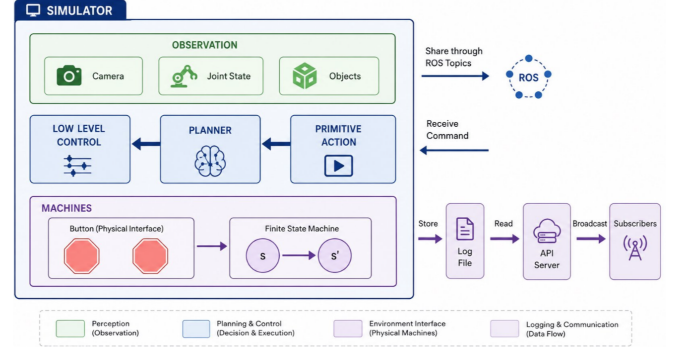


Fig. 2. Simulation environment: Observations (camera, joint state, objects) are shared over ROS topics; a Planner turns received commands into low-level control acting on machines modeled as a physical interface coupled to a finite-state machine; machine states are logged, served, and broadcast.

retrieves context from this memory to decide which condition and action nodes to instantiate, compiles the resulting tree into code that issues primitive commands to the simulator, and closes the loop by monitoring the simulator’s state — both directly, through shared topics, and indirectly, through a logged, served, and periodically polled system status that feeds the BT’s Condition nodes on every tick.

#### A. Simulation Environment and Machine Interface

As shown in Fig. 2, the simulator exposes three layers. The *Observation* layer provides perception of the environment through camera images, joint state, and object information, shared with the rest of the system through ROS topics. The *Planning & Control* layer receives a primitive command from outside the simulator, which a Planner turns into a motion or actuation carried out by Low-Level Control. These primitive commands act on the *Machines* layer, where each industrial machine is modeled as a physical interface (e.g., buttons) driving an underlying finite-state machine with states  $s$  representing the machine’s operating condition (idle, running, faulted, etc.). Every state transition is stored to a log file, which is read by an API server and broadcast to subscribers so that the current status of every machine is observable without querying the simulator directly.

#### B. Manual Parsing and Memory

Operating manuals arrive as prose, tables, and numbered procedure lists in heterogeneous formats. We first convert each manual to Markdown, a uniform representation that

preserves the document’s own hierarchy: the section headings become addressable units, and the state–action tables survive as tables rather than collapsing into flowing text.

That hierarchy makes the manual usable as memory. The handbooks in our benchmark share a recurring organization, including purpose and state inventory, one subsection per control button, the nominal start procedure, the state flow, one subsection per alarm with its recovery steps, safety prohibitions, a state–action summary table, escalation criteria, and a general recovery rule. Each of these serves a different node type: button subsections and the summary table supply the action vocabulary and its effects, the state flow and the alarm sections supply the preconditions that become Condition nodes, and the safety and escalation sections supply the cases in which the correct behavior is to stop rather than to retry.

The Markdown file is therefore loaded into a memory module (Fig. 3) that the agent queries by section, rather than re-reading the whole document at every decision point. When the agent instantiates a node it retrieves only the relevant unit, such as the recovery procedure for the alarm at hand, or the row of the summary table matching the current state, so that each node is grounded in a specific, quotable passage.

### C. Agentic Behavior Tree Generation

The agentic system (Fig. 3) builds a BT using the node types introduced in Sec. II-B: a Root node driving Sequence and Fallback control nodes, which govern Condition nodes. Sequence and Fallback nodes retrieve context from memory (e.g., what sub-procedure the manual specifies for a given task, and in what order) to determine how the tree should branch, while Condition nodes are grounded in the manual’s stated preconditions (e.g., machine states or interlocks that must hold before an action may proceed). Once instantiated, the tree is compiled into executable code, which issues primitive commands to the simulator described in Sec. III-A.

The loop is closed in two complementary ways. First, the simulator streams its system topics directly to the agentic system, giving immediate access to raw observations. Second, the logged machine states are served by an API server, consumed by an API client, and used to update a status module; this status is what the BT’s Condition nodes check on every tick. This separation lets condition checks reflect the machine’s actual, persisted operating state rather than only the most recent topic message. It is useful when a condition (e.g., “has the machine finished its current cycle”) depends on state that changes more slowly than the topic stream.

### D. Multi-Robot Composition and Coordination

The architecture in Sec. III-C describes a single tree acting through a single robot. To distribute this tree across a multi-robot team, we introduce a scheduling queue  $Q$  that holds pending individual action nodes or entire Behavior (sub)Trees alongside a pool of robots  $R$ , as shown in Fig. 4.

The queue is processed FIFO: the node at its head is popped and checked against a *Ready to run?* gate, corresponding to the manual-grounded preconditions of Sec. II-B; if the node is not yet ready, it is returned to the back of

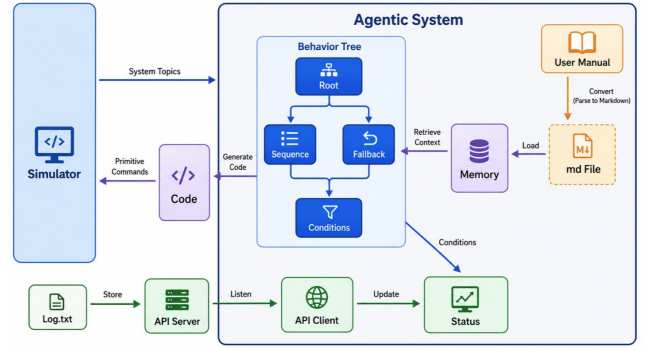


Fig. 3. Agentic system architecture. The user manual is parsed to Markdown and loaded into memory; the Behavior Tree retrieves context from memory to instantiate Sequence, Fallback, and Condition nodes, and is compiled into code that issues primitive commands to the simulator. Machine status, served through a logging/API pipeline, feeds back into the tree’s Condition nodes, closing the loop.

the queue rather than blocking scheduling of other work. Once a node clears this gate, it is resolved against the currently executing tree rooted at *Root*: a *Sequence* node continues ticking its currently active leaf, which is assigned to a specific robot  $r$ ; a *Parallel* node, by contrast, is requested as independent branches (*Branch A*, etc.), allowing each branch to be scheduled and dispatched to a different idle robot concurrently, rather than serialized at a single robot.

Algorithm 1 formalizes this process. At each iteration, the node at the front of  $Q$  is popped; if its preconditions are not satisfied it is pushed to the back of  $Q$  and scheduling continues with the next node. Otherwise, an idle robot is allocated; if none is available, the node is instead returned to the *front* of  $Q$  so it is retried first once a robot becomes free. Once a robot is reserved, if the dequeued item is itself a Behavior Tree, we resolve its currently active work node and, if it contains a *Parallel* node, additionally requeue each of its branches so that they may be picked up by other idle robots independently; the resolved work node is then dispatched to the reserved robot. This decouples *which* node runs next from *which robot* executes it based on the tree’s own Sequence/Fallback/Parallel semantics and the manual-derived preconditions, allowing the same manual-grounded tree to be transparently distributed across a homogeneous multi-robot team without being re-authored per robot.

## IV. SIMULATION AND BENCHMARK

We evaluate on a factory environment in which three machines, each an explicit finite state machine with its own operating manual, are operated by a simulated dual-arm mobile manipulator through their physical control panels. Its defining property is visible in Fig. 5: Machines 1 and 2 carry the same four coloured buttons under different configurations, so the same button means different things on the two machines. Appearance alone is therefore insufficient to act, and the mapping is recoverable only from the manual.

### Algorithm 1 Dynamic Multi-Robot Behavior Tree Scheduling

```

1: procedure SCHEDULE( $Q, R$ )
2:   while  $Q \neq \emptyset$  do
3:      $n \leftarrow \text{POPFRONT}(Q)$   $\triangleright$  Select the next queued task
4:     if not PRECONDITIONSSATISFIED( $n$ ) then  $\triangleright$  Wait until the task is ready
5:       PUSHBACK( $Q, n$ )
6:     continue
7:   end if
8:    $r \leftarrow \text{FINDIDLEROBOT}(R)$   $\triangleright$  Find an available robot
9:   if  $r = \emptyset$  then
10:    PUSHFRONT( $Q, n$ )  $\triangleright$  Retry when a robot is available
11:  continue
12: end if
13: RESERVE ROBOT( $r$ )
14: if ISBEHAVIORTREE( $n$ ) then
15:    $w \leftarrow \text{RESOLVEACTIVETASK}(n)$   $\triangleright$  Find the active
    executable node
16:   if CONTAINS PARALLEL NODE( $n$ ) then
17:     for all  $b \in \text{PARALLELBRANCHES}(n)$  do
18:       PUSHFRONT( $Q, b$ )  $\triangleright$  Queue branches concurrently
19:     end for
20:   end if
21:   DISPATCH( $w, r, n$ )  $\triangleright$  Execute the active BT node
22: else
23:   DISPATCH( $n, r, \emptyset$ )  $\triangleright$  Execute a non-BT task
24: end if
25: end while
26: end procedure

27: procedure ONCOMPLETE( $w, r, n$ )  $\triangleright$  Called when dispatched work terminates
28:   RELEASE ROBOT( $r$ )  $\triangleright$  Make the robot available again
29:   if  $w = \text{success}$  and  $n \neq \emptyset$  then
30:     if HAS REMAINING WORK( $n$ ) then
31:       PUSHBACK( $Q, n$ )  $\triangleright$  Schedule the BT's next active node
32:     end if
33:   end if
34: end procedure

```

TABLE I

FUNCTIONAL SCENE CONTENT IN  $s_1$ . COORDINATES ARE IN METRES.

| Content          | Count | Location        | Role              |
|------------------|-------|-----------------|-------------------|
| Work pieces      | 8     | (2.6–3.6, –5.1) | material          |
| Spare batteries  | 3     | (3.9–4.3, 0.16) | replacement stock |
| Seated batteries | 2     | M2, M3 modules  | consumable        |
| Machine M1       | 1     | (–1.8, –1.3)    | colour panel      |
| Machine M2       | 1     | (0.0, 1.7)      | colour panel      |
| Machine M3       | 1     | (9.0, –3.0)     | numeric panel     |

#### A. Environment

The environment builds on `at_factory` [11], an MIT-licensed Isaac Sim 5.0.0 [12] and ROS 2 Humble [13] integration distributed as a Docker image. The embodiment is a Galaxea R1: a wheeled base, a liftable torso, and two arms with parallel grippers [14]. Base motion, arm motion, and gripper actuation are all simulated, and the agent never receives a privileged teleport.

The scene supplies what the task consumes: eight work pieces on a staging rack, which the robot loads into a machine to be tuned, and spare batteries on a table across the floor, which replace the modules on Machines 2 and 3 when one fails. Both are far from the machines they serve.

#### B. Machines and Manuals

Each machine clamps a work piece, measures it, tunes it against an operator-supplied parameter, and verifies the result, running as a finite state machine in its own process. The three differ in control panel and in fault availability (Table II). Machine 3's manual, the most demanding, defines 18 states, 11 panel commands, and seven recovery procedures.

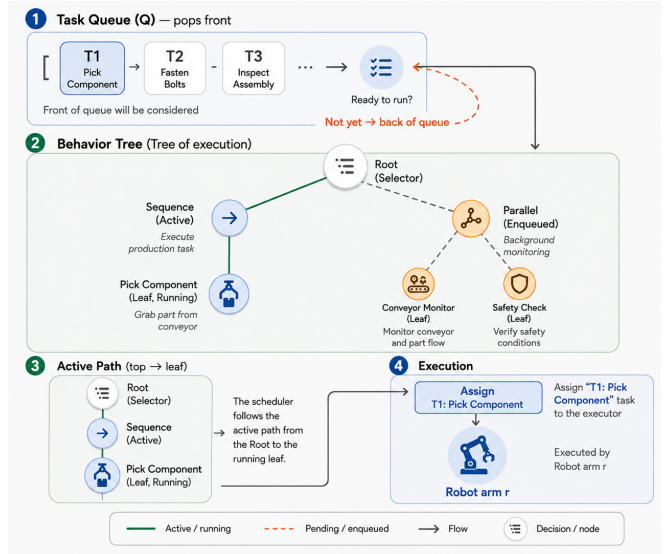


Fig. 4. Multi-robot scheduling, illustrated end to end on one queued task. (1) The scheduler pops the task at the front of queue  $Q$  and checks whether it is ready to run; a task that is not ready returns to the back of  $Q$ . (2) The executing Behavior Tree: a Selector root chooses between the active production Sequence, whose running leaf is PICK COMPONENT, and a Parallel branch of background monitoring leaves (CONVEYOR MONITOR, SAFETY CHECK), which remain enqueued rather than blocking the active path. (3) The scheduler follows the active path from the root to the currently running leaf. (4) That leaf is dispatched as an assignment — here, “T1: Pick Component” — to an idle executor, robot arm  $r$ .



Fig. 5. The factory scene: the Galaxea R1, the three machines (M1–M3), the staging rack holding the work pieces, and the table of spare batteries. Panel close-ups are in Table. I.

#### C. Interfaces and Tasks

The system provides four robot-mounted image streams at  $224 \times 224$ , including a head-mounted RGB–depth pair and RGB streams from the left and right wrists. These observations are available for physical interaction with the environment but are not required for machine-state estimation. Machine state and active alarms are obtained through the API interface described in Sec. III-C and are used by the Behavior Tree’s Condition nodes for procedural execution and recovery. Actions are parameterised primitives, such as navigation, button presses, grasping and placing, rather than joint torques, which keeps episodes of dozens of decisions tractable. Presses remain physical, so a mis-executed press is a failure and not a silent no-op.

The suite has two task families: nominal operation and

TABLE II  
THE THREE MACHINE CONFIGURATIONS

| Machine | Panel                         | Battery module |
|---------|-------------------------------|----------------|
| M1      | Four colour buttons, config A | —              |
| M2      | Four colour buttons, config B | ✓              |
| M3      | Twelve-key numeric panel      | ✓              |

Long-horizon operation so one robot interleaves two color mappings and a keypad within a single episode.

## V. EXPERIMENTS

### A. Baselines

We compare against four prompting-based baselines that generate robot behavior directly, without our memory module or tree synthesis: **ReAct** [15], which interleaves reasoning and acting with no explicit plan; **Code as Policies** [16], which emits a Python program over our primitive API; **ProgPrompt** [17], which generates an action sequence from a programmatic task scaffold; and **ReCAP** [18], which plans ahead and repairs recursively on failure. We also evaluate a **zero-shot** ablation that withholds the manual, isolating what grounding contributes. All conditions share our primitive interface and evaluation protocol (Sec. IV-C); only the intermediate representation differs from our Behavior Tree.

### B. Metrics

We report two metrics per condition: **task success rate**, the fraction of episodes in which the machine reaches the required terminal state, counted on the state transition itself so a machine already at that state cannot be recounted; and **execution time**, the time to execute the generated tree in simulation from dispatch to episode termination.

### C. Simple Short-Horizon Task Results

On the simple task set, the baselines achieve comparable success rates and wall-clock time (Fig. 6a), so the primary axis of separation among them is token cost. Code as Policies and ProgPrompt exhibit substantially lower token usage, as both express actions as short function calls or executable programs rather than free-form text, requiring the model to emit compact code rather than a verbose plan at each step. ReAct incurs higher cost, as it performs local planning at each step, re-reasoning over the current state on every turn rather than committing to a compact program upfront, which adds tokens with each additional step. ReCAP incurs the highest cost of all: its prompt tokens exceed ReAct’s substantially even though its completion tokens are smaller, a pattern consistent with the additional cost arising from re-sending accumulated context at each step rather than from generating more content per step.

### D. Practical Long-Horizon Task Results

On the long-horizon set the axis of separation moves from token cost to task success. The simple tasks left the baselines closely matched and separated primarily by token expenditure. On this set, every baseline degrades sharply,

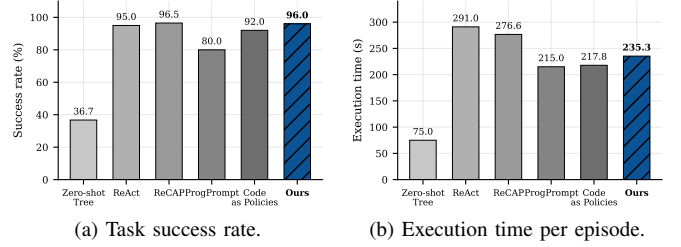


Fig. 6. Task success rate and execution time for short-horizon tasks.

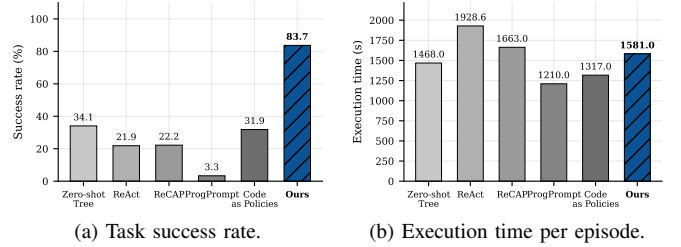


Fig. 7. Task success rate and execution time for long-horizon tasks.

and Ours is the only one to complete a majority of tasks (Fig. 7). Execution time does not track success here: ReAct and ReCAP take longer than Ours yet succeed far less often, so their extra time reflects wasted reasoning rather than progress, while ProgPrompt terminates fastest and least successfully, consistent with an upfront plan failing early rather than adapting.

## VI. CONCLUSION

We presented an agentic framework that grounds robot behavior in a machine’s operating manual, parsing the manual into memory and synthesizing an executable Behavior Tree whose condition nodes evaluate the manual’s stated preconditions against live machine state. A scheduling scheme extends a single generated tree across a multi-robot team by dispatching nodes to idle robots as their manual-derived preconditions are satisfied, without re-authoring the tree per robot. The approach was evaluated on a factory benchmark in which two machines share an identical control panel under different key mappings and faults require physical rather than informational recovery, conditions under which appearance alone does not determine the correct action.

## VII. FUTURE WORK

Two limitations remain. First, our experiments cover a limited set of tasks and conditions; broader evaluation is needed to assess additional aspects of the approach, including a wider range of fault alarms beyond those considered here, and to determine how well the generated tree generalizes to conditions not observed in this study. Second, all experiments were conducted with a single robot; multi-agent settings, in which multiple robots execute a shared tree or coordinate across trees, have not yet been evaluated and remain an important direction for future work.

## REFERENCES

- [1] Z. Pan, J. Polden, N. Larkin, S. Van Duin, and J. Norrish, “Recent progress on programming methods for industrial robots,” *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 2, pp. 87–94, 2012.
- [2] J. Arents and M. Greitans, “Smart industrial robot control trends, challenges and opportunities within manufacturing,” *Applied Sciences*, vol. 12, no. 2, p. 937, 2022.
- [3] G. Fragapane, R. de Koster, F. Sgarbossa, and J. O. Strandhagen, “Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda,” *European Journal of Operational Research*, vol. 294, no. 2, pp. 405–426, 2021.
- [4] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel, “Reconfigurable manufacturing systems,” *CIRP Annals*, vol. 48, no. 2, pp. 527–540, 1999.
- [5] H. ElMaraghy, G. Schuh, W. ElMaraghy, F. Piller, P. Schönsleben, M. Tseng, and A. Bernard, “Product variety management,” *CIRP Annals*, vol. 62, no. 2, pp. 629–652, 2013.
- [6] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen, “Robot skills for manufacturing: From concept to industrial deployment,” *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.
- [7] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.
- [8] M. Iovino, E. Scukins, J. Styrd, P. Ögren, and C. Smith, “A survey of Behavior Trees in robotics and AI,” *Robotics and Autonomous Systems*, vol. 154, p. 104096, 2022.
- [9] M. Colledanchise and P. Ögren, “How Behavior Trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees,” *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, 2017.
- [10] P. Ögren and C. I. Sprague, “Behavior trees in robot control systems,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 5, no. 1, pp. 81–107, 2022.
- [11] Momoi Open Source Organization, “at\_factory: Factory simulation environment with Isaac Sim 5.0.0,” [Online]. Available: [https://github.com/momoiorg-repository/at\\_factory](https://github.com/momoiorg-repository/at_factory), 2026, commit 7a2b3c4, accessed August 10, 2026.
- [12] NVIDIA, “Isaac sim,” 2025, accessed: 2026-08-10. [Online]. Available: <https://github.com/isaac-sim/IsaacSim>
- [13] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.
- [14] Galaxea AI, “R1: Dual-arm mobile manipulator,” <https://userguide-galaxea.github.io/>, 2025, product hardware specification and user guide; accessed 2026.
- [15] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, “ReAct: Synergizing reasoning and acting in language models,” in *Int. Conf. on Learning Representations (ICLR)*, 2023.
- [16] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2023, pp. 9493–9500.
- [17] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, “ProgPrompt: Generating situated robot task plans using large language models,” in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2023, pp. 11 523–11 530.
- [18] Z. Zhang, T. Chen, W. Xu, A. Pentland, and J. Pei, “Recap: Recursive context-aware reasoning and planning for large language model agents,” *arXiv preprint arXiv:2510.23822*, 2025. [Online]. Available: <https://arxiv.org/abs/2510.23822>

## APPENDIX

This appendix describes more details of Sec. IV.

### A. Scene

Scene  $s_1$  contains 138 USD prims within a walled  $15.5\text{m} \times 9.7\text{m}$  floor. Structural content, such as walls, storage racks, and a loading ramp, is not interacted with but occludes and obstructs. Table I lists the functional content, and Fig. 8 shows the staging rack. Named location labels (`navigate_to_workgroups`,



Fig. 8. The staging rack: eight work pieces are the material loaded into the machines; spare batteries are held on a separate table across the floor.

`new_battery_location`, and per-machine approach poses) provide symbolic navigation targets, so navigation and manipulation can be exercised independently.

The staging rack, the battery table, and the three machines are mutually distant, so a fault raised mid-cycle forces a traversal of the floor rather than a repair in place: recovery costs horizon, which is what makes it worth planning rather than reflexively attempting.

Two further scene variants,  $s_2$  and  $s_3$ , share the same machine and robot models and differ in floor plan and inter-station distance. They change navigation cost, not task semantics.

Every panel element, including each key, button, screen, door, and battery holder, is an individually addressable prim with its own collision geometry. Keys on the numeric panel are 4.2 cm across.

### B. Machine Interfaces

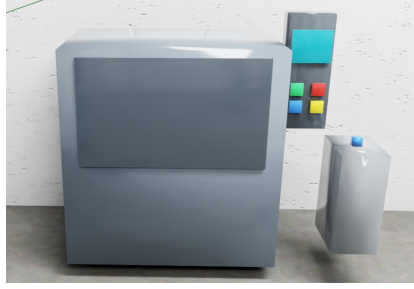
Machines 1 and 2 use four colored buttons under different configurations. A configuration maps red to battery replacement, green to door open/close, blue to start, and yellow to restart. Machine 3 uses a twelve-key numeric panel: digits 0–9, Enter, and Backspace. Commands are a digit followed by Enter. Backspace deletes the last digit entered and, when the machine is in a parameter-related alarm, returns it to parameter setup.

Each machine has its own manual, written in the register of industrial operator documentation: numbered procedures, state and command tables, a per-alarm recovery section, and maintenance-escalation criteria. Machine 3’s defines 18 states, 11 panel commands, a 26-row transition table, and seven recovery procedures. Under it, nominal operation is six panel interactions bracketing four automatic stages : close the door, confirm the battery, load material, enter parameter setup, supply a tuning parameter in  $[10, 100]$ , and, after the machine has run, a sixth press to release the finished piece. The manuals are released with the benchmark.

Each machine runs as a finite state machine in its own process, serving state at `/state` and accepting presses at `/press?key=.`, with a human-readable log at `/messages`. Ports are assigned per machine so all three run concurrently.



(a) Machine 1



(b) Machine 2



(c) Machine 3

Fig. 9. The three machines. (a) Machine 1 and (b) Machine 2 differ in body but carry the same four-button color panel under different key mappings, so the panel does not determine the correct action. (c) Machine 3 uses a twelve-key numeric panel. The pedestal beside (b) and (c) is the external battery module; Machine 1 has none.

TABLE III  
OBSERVATION TOPICS. WRIST STREAMS ARE OPT-IN.

| Topic suffix    | Mount       | Encoding | Default |
|-----------------|-------------|----------|---------|
| head.rgb        | upper torso | rgb8     | on      |
| head.depth      | upper torso | 32FC1    | on      |
| left.wrist.rgb  | left wrist  | rgb8     | off     |
| right.wrist.rgb | right wrist | rgb8     | off     |

Actuating a key and the machine consuming that key are separate events, and only the second advances the machine: a press that misses, or that arrives in a state where the key is not accepted, leaves the machine where it was.

The state set has 18 members in four groups: idle and ready (IDLE DOOR OPEN, READY), setup (PARAMETER SETUP, PARAMETER ERROR), checking (PRECHECK, VIOLATION CHECK), running (CLAMP, MEASURE, TUNE, VERIFY), terminal (FINISHED, QUALITY FAIL), and alarm (DOOR, BATTERY, MATERIAL, SENSOR FAULT, VIOLATION, EMERGENCY STOP).

Three branch points matter. PRECHECK diverts to the corresponding alarm if the door is open, the battery is not ready, the material is missing, or a sensor is invalid. VIOLATION CHECK diverts to VIOLATION if the tuning parameter falls outside  $[10, 100]$ . VERIFY returns to TUNE on failure until the retry count reaches three, then enters QUALITY FAIL. An emergency stop is accepted from any running state and is cleared only by an explicit restart.

### C. Observation Interface

Four robot-mounted cameras publish as ROS 2 `sensor_msgs/Image` under the `/rl.pro` namespace (Table III). All render at  $224 \times 224$  and publish on a fixed interval which keeps rendering off the critical path of physics stepping. Head RGB-depth pairs are enabled by default; wrist streams are enabled per experiment.

### D. Primitive Actions

Primitives are invoked as separate processes that either run to completion or fail, returning a status on which the caller branches (Table IV). Arm-using primitives take a left/right argument, and all primitives take a robot identifier,

TABLE IV  
PRIMITIVE ACTIONS.

| Primitive    | Arguments       | Effect                                          |
|--------------|-----------------|-------------------------------------------------|
| nav          | pose or label   | base motion to a pose                           |
| nav-object   | object          | base motion to an approach pose for that object |
| touch-button | prim, arm       | reach and press a key                           |
| gripper      | open/close, arm | actuate a gripper                               |
| manipulate   | object, arm     | grasp a graspable object                        |
| put          | target, arm     | place the held object                           |
| reset        | arm             | return an arm to home                           |

allowing the same action set to support multi-robot episodes. Operating at this level of granularity, rather than directly controlling joint torques, is a deliberate design choice: episodes span dozens of decisions, and requiring end-to-end joint control over this horizon would emphasize low-level control performance rather than procedural reasoning. Button presses remain physically grounded: an arm must reach the target key and make contact. A mis-executed press is therefore treated as a task failure rather than a no-op.

One execution detail affects interpretation. Navigation reports a soft failure in which the planner halts outside its position tolerance; the harness treats this as success so that a marginal approach does not abort an otherwise correct long episode. Its frequency is reported with our results.

### E. Behavior-Tree Generation Prompt

Fig. 10 gives the prompt template issued once per manual. Bracketed slots are the only parts that vary across the suite; the constraint block is fixed, and it is what makes a generated tree executable by construction rather than after repair.

### F. Manual Retrieval Example

Fig. 11 reproduces an excerpt from one of the operating handbooks: the recovery procedures for the *Battery not ready*, as they appear after the Markdown conversion of Sec. III-B. We trace these excerpts through the agent to illustrate what is retrieved and what each retrieved passage contributes.

The excerpt supplies the recovery itself. Its ordered steps become a Sequence of `nav`, `manipulate`, `put`, and

**ROLE.** You operate a mobile manipulator in a simulated factory.

**TASK.** Emit a behavior-tree JSON that drives  $\langle \text{machine} \rangle$  through  $\langle \text{goal} \rangle$ .

**GIVEN.** *template* — node types, APIs, interactive commands, primitives, conditions, object names; *locations* — `location_label.sl`; *procedure* — the operating manual retrieved from memory.

**CONSTRAINTS.**

*C1 Schema.* Follow the template’s nesting and field names. Every action node supplies primitive, args, robot\_num, timeout, and arm, machine ID or FSM key where required.

*C2 Closed vocabulary.* Use only what the template exposes. Invent no API, primitive, command, condition, object, argument, or node type.

*C3 Grounding.* Take every navigation, button, placement, work-object, and battery target from `location_label.sl`, by its exact label.

*C4 Procedure.* Follow the manual’s order of operations.

*C5 Scope.* Act on  $\langle \text{machine} \rangle$  alone.

*C6 Robustness.* Primitives may fail. Handle every fault case, using only the conditions and actions the template lists.

**OUTPUT.** Derive the tree internally, then output only valid, directly executable JSON with no surrounding prose.

Fig. 10. Prompt template for behavior-tree generation. For the single-machine cycle of Sec. IV, the slots bind to  $\langle \text{machine} \rangle = \text{Machine 1}$  and  $\langle \text{goal} \rangle = \text{one complete operation cycle followed by a reset for the next round}$ .

## 5.2 Battery Not Ready Alarm

*Alarm message:* Battery not ready

*Cause:* The battery is missing, low, disconnected, or not confirmed after replacement.

*Required button:* Replace Battery

*Battery location:* Lower-right side of the machine.

*Recovery procedure:*

- Go to the lower-right side of the machine.
- Open the battery cover.
- Remove the old or failed battery.
- Install a ready battery.
- Make sure the battery is fully seated and connected.
- Close the battery cover.
- Press the Replace Battery button.
- Confirm the display message: Battery Ready.

*Expected recovery result:* The battery condition is cleared and the machine is ready to operate.

Fig. 11. Excerpt from a machine handbook, abridged. Each alarm is documented in this form: message, cause, required button, physical location, ordered recovery steps, and expected result.

touch-button primitives, and its expected result becomes a Condition node checking the machine state after the press.

Two further passages constrain that Sequence rather than populating it. The handbook’s general recovery rule requires the physical correction to precede the button press, which fixes the order of the Sequence’s children; without it, an agent may press *Replace Battery* without having replaced the battery, producing a state trace indistinguishable from correct recovery until the next precheck. The escalation criteria specify that an alarm persisting after a completed replacement is a maintenance condition, which becomes the Fallback branch taken when the confirming Condition fails. That branch terminates the task rather than retrying it.

This is deliberately the simplest case in the suite. Recoveries spanning several alarms, or interleaved with an in-progress operation cycle, retrieve correspondingly more passages, but the correspondence between documented pro-

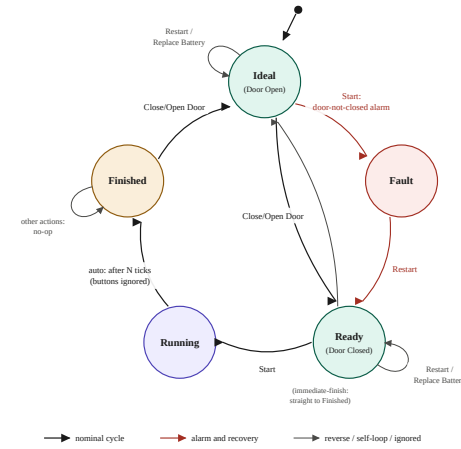


Fig. 12. An example machine FSM (Machines 1 and 2). Black edges are the nominal cycle; red edges are the alarm and its recovery; gray edges are the reverse door toggle, self-loops, and ignored inputs.

cedure and node type is unchanged.

## G. Example of a Machine Finite State Machine

Every machine in the benchmark is modeled as a finite state machine  $(S, A, \delta)$ : a state set  $S$ , the panel’s input alphabet  $A$ , and a transition function  $\delta : S \times A \rightarrow S$  that is partial by design. Inputs with no defined transition leave the state unchanged, and operation itself advances autonomously rather than by input. This appendix illustrates the construction on the simplest instance, the machine shared by Machines 1 and 2; Machine 3’s numeric-panel machine has a larger  $S$  and  $A$  but is built the same way.

For this instance (Fig. 12),  $S$  has five members: IDEAL (DOOR OPEN), READY (DOOR CLOSED), RUNNING, FINISHED, FAULT.  $A$  is the four panel buttons: Close/Open Door, Replace Battery, Start, Restart. The nominal cycle traverses the first four states and returns; the door button toggles the first two; Start pressed with the door open raises the door-not-closed alarm and enters FAULT, from which Restart recovers.

Three properties of the construction hold across all machines and are what the benchmark exercises. *The same input means different things in different states*: here, Start begins operation from READY and raises an alarm from IDEAL, so the panel alone does not determine the correct action. *Operation is autonomous*: RUNNING accepts no input and completes after a configurable number of ticks, so pressing through it gains nothing. *Undefined transitions are silent*: an input with no transition in the current state leaves the machine where it is, so a wrong press produces no observable effect and a policy must read the resulting state rather than infer success from having acted.

Each machine runs as its own process, serves its current state and active alarm as JSON over HTTP, and accepts inputs on the same interface.