Logic-Based Multi-Robot Coordination in Digital Twins: A DALI2 + CoppeliaSim Case Study for Cooperative Search and Rescue

Lorenzo De Lauretis[1], Giovanni De Gasperis[1,*] and Stefania Costantini[1]

[1]Department of Information Engineering, Computer Science and Mathematics (DISIM), University of L'Aquila, Via Vetoio, 67100 L'Aquila, Italy

Abstract

We present a case study of declarative multi-robot coordination using DALI2, a SWI-Prolog reimplementation of the DALI agent-oriented logic programming language, integrated with the CoppeliaSim robot simulator as a digital twin. Three mobile rescuers must locate victims scattered among obstacles in an arena and bring them to a safe zone; victims marked as *heavy* can only be moved by two robots cooperating via a coordinator-mediated synchronisation protocol. Each robot carries a vision sensor whose images are periodically analysed by a two-phase pipeline: a fast green-pixel pre-filter that skips the LLM when no victim-coloured pixels are present, and a vision-capable LLM (local Qwen3.5-9B via JAN [1] or OpenRouter gpt-4o) that returns structured JSON objects which the bridge parses into Prolog events driving the agent's decision-making, including autonomous exploration and obstacle avoidance. We show how DALI2's reactive rules, internal events, FIPA-style proposals, told/tell priority queues, multi-event synchronisation, and a built-in LLM oracle map naturally onto a layered cognitive architecture in which symbolic deliberation drives an embodied simulation through a thin Python bridge. Robots autonomously explore the arena via patrol waypoints, stop to photograph their surroundings, wait for the vision model's analysis, and then decide whether to report a victim, avoid an obstacle, or continue exploring. The bridge translates DALI2's Redis-based LINDA channel into CoppeliaSim's ZeroMQ Remote API, preserving the framework's distributed star topology and making the twin a first-class testbed: agents do not know whether their bodies are simulated or physical. The full source code, scene generator, agent specification, and reproducibility instructions are publicly available.

Keywords

multi-agent systems, logic programming, robot coordination, digital twin, CoppeliaSim, DALI, cognitive architectures, vision-language models

1. Introduction

Modern autonomous systems are increasingly expected to combine the flexibility of data-driven models with the safety, auditability and formal guarantees offered by symbolic reasoning. In multi-robot applications – search and rescue, environmental monitoring, warehouse logistics – this tension is particularly visible: low-level perception and motion are well served by learned controllers, yet *coordination* (who does what, when, with whom) demands explicit representations of obligations, capabilities and temporal constraints that are hard to extract from a neural network.

This paper revisits a logic-based answer to that question. DALI [2, 3] is an agent-oriented programming language designed around an event-based operational semantics: every external event, internal deliberation, goal commitment, action and constraint is a first-class syntactic construct, exposed to the programmer as a Prolog clause. We have recently rebuilt the platform on top of SWI-Prolog as DALI2[1] [4], keeping the original surface syntax while moving from a TCP-based Linda server to a Redis

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.

*Corresponding author.

lorenzo.delauretis@univaq.it (L. De Lauretis); giovanni.degasperis@univaq.it (G. De Gasperis); stefania.costantini@univaq.it (S. Costantini)

https://www.disim.univaq.it/GiovanniDeGasperis (G. De Gasperis); https://www.disim.univaq.it/StefaniaCostantini (S. Costantini)

0000-0002-8796-3672 (L. De Lauretis); 0000-0001-9521-4711 (G. De Gasperis); 0000-0002-5686-6124 (S. Costantini)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://github.com/AAAI-DISIM-UnivAQ/DALI2.

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/CARLA_regular_01.pdf

star topology, packaging the runtime as a process-per-agent architecture, and adding a built-in LLM oracle accessible through the OpenRouter API.

Our contribution is threefold:

1. We describe a layered cognitive architecture in which DALI2 agents reason at the symbolic level, while a thin Python *bridge* grounds high-level commands into a CoppeliaSim digital twin via the ZeroMQ Remote API [5]. The bridge is transparent in both directions: agents send the same `send/2` messages as in any DALI2 program, and the simulated robots feed events back through the same LINDA pub/sub channel.
2. Around that architecture we develop a non-trivial case study – cooperative search and rescue with light *and* heavy victims scattered among obstacles – that exercises features characteristic of the DALI family: belief-based synchronisation for the joint lift (with a clear upgrade path to DALI2's bounded-temporal-window mechanism), FIPA-style auctions for task allocation, told/tell priority queues for handling critical messages out of order, internal-event deliberation, and an optional LLM-driven triage step that consults the AI oracle without surrendering the coordination guarantees of the symbolic layer.
3. We close the perception loop by equipping each robot with a vision sensor and feeding its images through a two-phase pipeline: a fast green-pixel pre-filter (sub-millisecond, eliminates >90 % of frames) and a vision-capable LLM (local Qwen3.5-9B via JAN [1] or OpenRouter gpt-4o). The bridge parses the structured JSON response (with keyword-matching fallback and Qwen3.5-9B think-block stripping) and publishes typed Prolog events such as `vision_result(victim(light))`, `vision_result(obstacle)`, or `vision_result(clear)`. Robots autonomously explore the arena using a grid of patrol waypoints; at each waypoint the robot pauses, photographs its surroundings, and – only if green pixels are detected – waits for the LLM analysis, then decides whether to report a victim to the coordinator, perform a clearance-based obstacle-avoidance manoeuvre, or move on to the next waypoint. This demonstrates that DALI2's event model can absorb multimodal, neural-network-driven perception – including closed-loop decision-making – without any changes to the coordination logic.

The combination targets the goals of the CARLA workshop: the agents' reasoning is logic-based and reproducible, while the digital twin serves as a controllable testbed where embodied behaviour can be benchmarked before any deployment to physical hardware.

The remainder of the paper is organised as follows. Section 2 summarises DALI2 and CoppeliaSim. Section 3 presents the three-tier architecture and the bridge protocol. Section 4 develops the cooperative search-and-rescue scenario, with annotated DALI2 code snippets. Section 5 explains the LLM oracle and vision-based perception. Section 6 reports observations from running the system. Sections 7, 8 and 9 discuss limitations, related work, and ongoing extensions.

2. Background

2.1. DALI and DALI2

DALI [2, 3, 6] is a declarative agent-oriented language belonging to the *event-condition-action* (ECA) family. An agent is a Prolog program extended with a small set of operators that the language runtime intercepts:

`eventE(X) :> Body` reacts to an external event `event(X)` (the suffix `E` marks the head as external);

`eventI(X) :> Body` together with `internal_event(event, P, Rep, Start, Stop)` declares an *internal* event that the agent fires proactively, subject to a period `P`, a repetition policy, a start condition (a Prolog goal) and a stop condition;

`Cond :< Action` specifies a condition-action rule that fires once when `Cond` transitions from false to true (edge-triggered);

`Head ~ / Past` (and the variants `</` and `?/`) reacts to patterns on past memory and consumes them;

`:~Cond` expresses an *integrity constraint* that the engine checks every cycle and that, on violation, is reported to the agent;

`ev1E, ev2E, within(N) :> Body` is a *multi-event*: the body fires when all listed events appear in the agent's past memory within an `N`-second window (the original DALI's `deltat`).

DALI further provides goals (`obt_goal` and `test_goal`), beliefs (`believes/1` and `assert_belief/retract_belief`), named actions (suffix `A`), past-event lifetimes (`past_event/2`, `remember_event/2`), and a *told/tell* mechanism that filters incoming and outgoing messages by pattern, with optional Prolog body conditions and a priority that the receiver uses to sort its message queue.

DALI2 preserves the surface syntax verbatim while modernising the runtime. Each agent is a separate SWI-Prolog process; all communication flows through a Redis star topology with two pub/sub channels (`LINDA` for inter-agent messages, `LOGS` for monitoring) and a Redis `SET` replacing the original Linda blackboard. A web UI on port 8080 exposes per-agent beliefs, past memory, learned associations, goal status, and a live event log. An *AI Oracle* module wraps the OpenRouter API, allowing the predicate `ask_ai(Context, Result)` to invoke an arbitrary LLM and bind `Result` to a Prolog term that has been validated against the agent's `told` rules. The architectural shift from TCP/Linda to Redis pub/sub also means that running the same program in distributed mode is a configuration change, not a code change: each node only needs to be told the same `REDIS_HOST`.

2.2. CoppeliaSim

CoppeliaSim (formerly V-REP) is a multi-physics robot simulator widely used for behaviour-based and reinforcement-learning research [7]. It exposes a ZeroMQ-based Remote API client [5] that allows external processes to seamlessly control the simulation. We rely on this API exclusively, as it keeps the digital-twin layer easily replaceable (e.g. by Webots, Gazebo, or a real-robot driver with the same surface API).

3. Architecture

The system is layered (Fig. 1). At the top, DALI2 agents reason about tasks, beliefs and constraints. At the bottom, a CoppeliaSim scene contains the embodied robots and the environment. In between, a Python *bridge* translates symbolic commands into ZMQ Remote API calls, and translates simulator state into events published back on the LINDA channel.

3.1. The bridge as a virtual agent

Crucially, the bridge does *not* extend DALI2. It is a regular Redis pub/sub client that registers itself as the agent `sim` and consumes messages addressed to that name. All agents continue to use the standard DSL primitives:

```
%% From any rescuer:
send(sim, set_target(Me, X, Y)).
send(sim, attach(Me, victim_2)).
send(sim, go_to_depot(Me)).
```

In the other direction, the bridge publishes events to robot agents exactly as another DALI2 agent would:

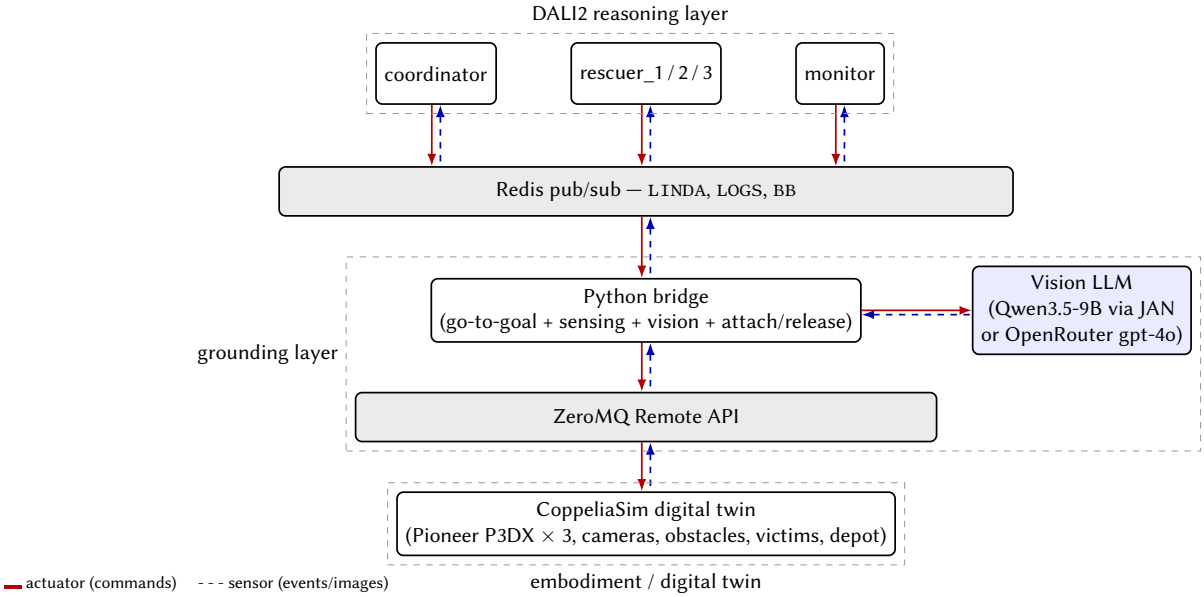


Figure 1: Three-tier architecture with double arrows. Solid red arrows carry actuator commands (set_target, attach, release); dashed blue arrows carry sensor data (pose, screenshots, vision analysis). The bridge forwards camera images to a local vision LLM and publishes the result back on the LINDA channel.

```
%% From the bridge:
LINDA  <- "rescuer_1:position(2.31,-0.74,0.18):sim"
LINDA  <- "rescuer_1:victim_in_range(victim_2,-3.50,2.50,heavy):sim"
LINDA  <- "rescuer_1:at_target:sim"
LINDA  <- "rescuer_1:delivered(victim_3):sim"
LINDA  <- "rescuer_1:vision_result(clear):sim"
```

Because LINDA is a single broadcast channel filtered by the addressee field, swapping the bridge for a real-robot driver only requires the new driver to keep the same protocol. Conversely, a DALI2 program written for the simulator runs unchanged on top of any backend that respects the protocol.

3.2. Bridge internals

The bridge runs at a fixed control rate (default 10 Hz). At each tick, for every robot it: (i) reads the world pose via `sim.getObjectPosition` and `sim.getObjectOrientation`; (ii) runs a proportional unicycle controller toward the current target with saturation on linear and angular velocity; (iii) integrates the unicycle model one step and writes the new pose back with `sim.setObjectPosition`/`setObjectOrientation` (*kinematic* mode – the Pioneer's built-in dynamics are disabled by the scene builder, so the bridge controls the robot's pose directly); (iv) emits LINDA events for telemetry, target arrival, and the first time a victim falls within the detection radius (3.0 m by default). The bridge also performs *reactive obstacle avoidance*: when a scene obstacle falls within a 1 m frontal cone (35° half-width), the bridge computes candidate waypoints to the left and to the right of the robot, scores each by summing the distance to nearby obstacles (a simple clearance heuristic), and steers toward the less obstructed side before resuming the original target. A 1.5 s cooldown timer prevents oscillation between repeatedly triggering the same obstacle.

Pickup and release are implemented through `sim.setObjectParent` so that, once attached, a victim's pose is rigidly tied to the carrier; release at the depot marker triggers a `delivered/1` event back to the responsible robot. Upon delivery, the victim cuboid is removed from the scene entirely (`sim.removeObject`). Heavy victims are still attached to a single carrier physically, but the DALI2 layer enforces that two robots be in position before any attach command is even issued (Section 4.2). During the cooperative carry, the second robot enters *formation-follow* mode: the bridge kinematically

Table 1
Spatial layout of the default scenario. Distances are Euclidean from each robot's spawn to the nearest victim and to the depot.

Entity	x (m)	y (m)	Notes
rescuer_1	-5.0	-5.0	faces NE ($45°$)
rescuer_2	5.0	-5.0	faces NW ($-45°$)
rescuer_3	0.0	5.0	faces S ($180°$)
victim_1 (light)	5.0	3.0	nearest robot: r2 (8.0 m)
victim_2 (heavy)	-4.0	-3.0	nearest robot: r1 (2.2 m)
victim_3 (light)	-3.0	4.0	nearest robot: r3 (3.2 m)
depot	0.0	4.7	safe zone

places it 0.6 m beside the leader at every tick, so the pair moves in lockstep.

4. Case study: cooperative search and rescue

The scenario takes place in an 11×11 m flat arena cluttered with eight static obstacles (cuboids of varying size and colour, representing rocks, crates and debris). Three victims are scattered among the obstacles: two light (fluorescent-green cuboids, 0.30 m wide) and one heavy (fluorescent-green cuboid, 0.45 m wide – same colour, larger size). A blue 3-D cylinder marker denotes the depot (safe zone) at coordinates $(0.0, 4.7)$. Three Pioneer P3DX robots, instantiated by the scene-builder script from CoppeliaSim's bundled `pioneer p3dx.ttm` model, start from opposite corners and edges of the arena. The stock Pioneer model ships with a built-in Lua child script implementing an obstacle-avoidance wander; the scene builder removes every such script from the model tree (`sim.getObjectsInTree` + `sim.removeObject`) so that only the bridge's commands drive the wheels. Each robot is fitted with a forward-facing *vision sensor* (256×256 pixels, $90°$ FOV), whose images the bridge captures every second.

Table 1 details the spatial layout. Robots spawn at the arena's periphery (corners and edge), facing inward; victims are placed 2–8 m from the nearest robot (detection radius is 3 m), so discovery order depends on the patrol grid shuffle. The eight obstacles (rocks, crates, and debris fragments of 0.25–0.50 m size) are distributed so that no straight-line path exists between any spawn point and the depot, forcing the reactive avoidance behaviour described in Section 3.

4.1. Agents and message protocol

The four DALI2 agents are declared in a single file `agents/rescue.pl`. Each `:- agent(Name).` directive opens a context for subsequent rules. We highlight three excerpts.

Rescuer telemetry. The bridge sends position readings every second; the rescuer simply forwards them to the coordinator with low told-priority while updating its own beliefs:

```
positionE(X, Y, Th) :>
    retract_belief(pose(_,_,_)),
    assert_belief(pose(X, Y, Th)),
    believes(robot_id(Me)),
    send(coordinator, position_update(Me, X, Y)).
```

Coordinator told/tell rules. The coordinator declares an explicit priority for every message it accepts. Critical messages overtake routine telemetry, which arrives much more frequently:

```
told(_, victim_seen(_,_,_,_),     200) :- true.
told(_, victim_rescued(_),        180) :- true.
```

```
told(_, ready_to_lift(_,_),       140) :- true.
told(_, bid(_,_,_),               120) :- true.
told(_, position_update(_,_,_),    10) :- true.
told(_, heartbeat(_),               5) :- true.
```

One-shot auction. A new sighting becomes an open task and triggers a request for bids; each rescuer answers with a cost equal to its Euclidean distance to the victim. The coordinator collects bids during one cycle and awards the task to the lowest bidder, or to the two lowest bidders for a heavy victim:

```
solicit_bidsI :>
    believes(open_tasks(Tasks)),
    member(Vid, Tasks),
    \+ believes(assigned(Vid, _)),
    \+ believes(assigned_pair(Vid, _, _)),
    \+ believes(soliciting(Vid)),
    believes(victim_info(Vid, X, Y, W)),
    assert_belief(soliciting(Vid)),
    broadcast(request_bid(Vid, X, Y, W)).
internal_event(solicit_bids, 2, forever, true, forever).
```

The auction-award helper dispatches on `Weight` and on the number of usable bids:

```
helper(award(Vid, Weight, Bids, X, Y)) :-
    ( Bids == [] -> true
    ; Weight == light, Bids = [_-Robot|_] ->
        assert_belief(assigned(Vid, Robot)),
        send(Robot, rescue_assignment(Vid, X, Y, light)),
        helper(cleanup_bids(Vid))
    ; Weight == heavy, Bids = [_-R1, _-R2 | _], R1 \== R2 ->
        assert_belief(assigned_pair(Vid, R1, R2)),
        send(R1, rescue_assignment(Vid, X, Y, heavy(partner(R2)))),
        send(R2, rescue_assignment(Vid, X, Y, heavy(partner(R1)))),
        helper(cleanup_bids(Vid))
    ; true   %% heavy task, only one bidder yet -- wait
    ).
```

This is, semantically, a degenerate Contract Net protocol [8]: the manager (coordinator) calls for bids, every contractor evaluates its own cost locally, and the manager selects the winner(s). Because cost is a simple closed-form expression (Euclidean distance d), the auction terminates in a single cycle with the constraint that at least two bids exist for heavy tasks.

4.2. Cooperative lift synchronisation

A heavy victim cannot be moved unless both assigned rescuers are *simultaneously* in position. In the original DALI literature, the canonical encoding for such joint events is the multi-event [3] with a temporal window:

```
e1E(R1, V), e2E(R2, V), within(5) :> Body.
```

DALI2 supports the same `within(N)` construct. In our current implementation we use a simpler, belief-based encoding. Each rescuer that reaches the victim signals readiness to the coordinator, which stores it as a belief and checks every cycle whether both partners have reported:

```
%% Each rescuer signals readiness when it physically reaches a
%% heavy victim.
ready_to_liftE(Robot, VictimId) :>
    assert_belief(ready_at(Robot, VictimId)),
    assert_belief(ready_at_t(Robot, VictimId)).

%% Both partners ready -> issue lift_now to both.
sync_liftI :>
```

```
    believes(assigned_pair(Vid, R1, R2)),
    believes(ready_at_t(R1, Vid)),
    believes(ready_at_t(R2, Vid)),
    send(R1, lift_now(Vid)),
    send(R2, lift_now(Vid)),
    ( retract_belief(ready_at(R1, Vid)) ; true ),
    ( retract_belief(ready_at(R2, Vid)) ; true ),
    ( retract_belief(ready_at_t(R1, Vid)) ; true ),
    ( retract_belief(ready_at_t(R2, Vid)) ; true ).
internal_event(sync_lift, 0, forever, true, forever).
```

Because `ready_at_t/2` is stored as a persistent belief, the current version does *not* enforce a temporal deadline: once a rescuer reports readiness the belief stays until the lift fires or the task is re-auctioned. Adding a bounded temporal window is a one-line change – declaring `past_event(ready_at_t(_,_), 5)` and replacing `believes` with `has_past` in `sync_liftI` – and is planned for the next iteration. The coordinator's beliefs are still inspectable at runtime via `/api/beliefs?agent=coordinator`.

4.3. Integrity constraints (language capability)

DALI's `:~` operator lets the programmer declare invariants that the engine checks every cycle. For instance, `:~(believes(open_tasks([])) ; \+all_unavailable)` would assert that at least one rescuer is available whenever open tasks exist. A violation is logged on the LOGS channel and made visible on the web UI; the behavioural reaction is decoupled, handled by a separate reactive rule or `when/2` monitor. The current scenario does not yet declare integrity constraints; adding them is planned as part of the temporal-window and failure-handling extensions described in Section 9.

4.4. Delivery and immediate resumption

When a rescuer releases a victim within the depot radius, the bridge publishes `delivered(VictimId)`. The Prolog handler immediately transitions the robot back to the `exploring` state and sends a new `explore` command to the bridge, bypassing the five-cycle wait of the `exploreI` internal event:

```
deliveredE(Id) :>
    believes(robot_id(Me)),
    ( retract_belief(carrying(_)) ; true ),
    ( retract_belief(target(Id, _, _, _)) ; true ),
    ( retract_belief(state(_)) ; true ),
    assert_belief(state(exploring)),
    send(coordinator, victim_rescued(Id)),
    send(sim, explore(Me)).
```

For heavy pairs, the bridge additionally clears the follower's `target_xy` and marks its arrival as published, preventing a spurious `at_target` event on the next tick. The coordinator cleans up the corresponding `assigned_pair` belief and removes the victim from `open_tasks` upon receiving `victim_rescued/1`. Because the auction loop runs as a periodic internal event (every 2 cycles), any victim that was assigned to a robot that becomes unavailable can be re-auctioned by simply pushing it back onto `open_tasks`.

5. LLM-augmented perception and triage

5.1. Text-based triage (AI Oracle)

DALI2's AI oracle is invoked through a single Prolog predicate, `ask_ai/2`. When configured with an OpenRouter API key, it forwards the context to the selected model and returns a Prolog term that has been validated against the agent's `told` rules; otherwise it transparently falls back to

`suggestion(no_ai_available)`. In the rescue scenario we use it as a soft advisor on incoming sightings:

```
victim_seenE(VictimId, X, Y, Weight) :>
    ...,
    ( ai_available ->
        ask_ai(triage(victim(VictimId), location(X,Y),
                      weight(Weight)), Advice),
        log("AI triage advice: ~w", [Advice])
    ; true ).
```

The advice is logged but never overrides the deterministic auction result. This is a deliberate design choice that mirrors the neuro-symbolic "advisor + safety net" pattern [9, 10]: the LLM contributes soft preferences (e.g. "vital signs were already detected for victim_2, escalate"), while the symbolic auction enforces correctness properties that an LLM cannot guarantee out of the box – single assignment per task and two-robot coverage of heavy victims. Because `told` rules filter the model's response, malformed or out-of-vocabulary advice is quietly rejected without affecting the rest of the system.

5.2. Vision-based perception and exploration

Beyond text-based triage, each rescuer possesses a forward-facing vision sensor and autonomously patrols the arena. When a rescuer is idle, an internal event fires periodically and commands the bridge to drive it to the next waypoint in a shuffled grid covering the 11×11 m arena:

```
exploreI :>
    believes(state(idle)),
    believes(robot_id(Me)),
    retract_belief(state(_)),
    assert_belief(state(exploring)),
    send(sim, explore(Me)).
internal_event(explore, 5, forever, believes(state(idle)), forever).
```

The bridge generates a shuffled grid of patrol waypoints (spacing 2 m), and on each `explore` command drives the robot to the next waypoint. At each waypoint the robot pauses, the bridge captures a 256×256 screenshot from the robot's vision sensor and runs a two-phase analysis pipeline:

Phase 1: fast green-pixel pre-filter. All victims in the scene are fluorescent-green cuboids. Before invoking the (expensive) LLM, the bridge scans a random 10 % sample of the image's pixels for the fluorescent-green signature ($G \geq 200$, $R \leq 100$, $B \leq 100$, $G - \max(R, B) \geq 100$). If fewer than six pixels match, the bridge immediately publishes `vision_result(clear)` and skips the LLM entirely. This takes $\sim$1 ms per frame and avoids the 1–5 s round-trip for the vast majority of screenshots that show only floor, obstacles, or walls.

Phase 2: vision LLM. When green pixels *are* detected, the bridge sends the image (base64 JPEG) to a vision-capable LLM via the standard OpenAI-compatible `/v1/chat/completions` endpoint. The default backend is a local Qwen3.5-9B model (IQ4_XS quantisation) served through JAN [1]; alternatively, the launcher accepts an `–openrouter` flag that routes requests to OpenRouter's `gpt-4o` with a longer timeout (30 s). The system prompt requests a structured JSON response:

```
{"victim": false, "victim_type": null,
 "obstacle": false, "path_clear": true}
```

Because Qwen3.5-9B wraps its reasoning in `<think>...</think>` tags, the bridge strips these blocks with a regular expression before parsing. The JSON is parsed first; if that fails, a keyword-matching fallback scans for terms such as "victim", "obstacle", "green cube", etc. The bridge publishes a typed event to the agent: `vision_result(victim(Type))`, `vision_result(obstacle)`, or `vision_result(clear)`. The agent reacts accordingly:

```
vision_resultE(victim(Type)) :>
    believes(robot_id(Me)),
    believes(pose(X, Y, Th)),
    Vx is X + 2.0 * cos(Th), Vy is Y + 2.0 * sin(Th),
    send(coordinator, victim_seen(vision_detect, Vx, Vy, Type)),
    ( believes(state(exploring)) ->
        retract_belief(state(_)), assert_belief(state(idle))
    ; true ).

vision_resultE(obstacle) :>
    believes(robot_id(Me)),
    send(sim, avoid_obstacle(Me)).

vision_resultE(clear) :>
    log("Vision: path clear").
```

When an obstacle is detected, the bridge computes candidate waypoints to the left and right of the robot's current heading, scores each by clearance from nearby obstacles, and steers toward the less obstructed side (see Section 3). When a victim is spotted, the robot estimates the victim's position $\sim$2 m ahead along its heading and reports it to the coordinator; the sighting enters the standard auction pipeline.

Critically, vision-based sightings are indistinguishable from proximity sightings at the coordination level. The coordinator cannot tell whether a `victim_seen` event originated from the proximity sensor or the vision LLM, and does not need to: the auction's correctness guarantees hold regardless of the sensing modality. If JAN is offline or the LLM call times out, the bridge publishes `vision_result(clear)` by default and the robot continues with proximity-based detection alone.

6. Reproducibility and observations

We packaged the case study as a self-contained repository `DALI2-Robot-Coordination/`, available under the same licence as DALI2. Setting up the demo requires:

1. a CoppeliaSim Edu install (≥ 4.5);
2. a Redis instance (locally or via the included `docker-compose.yml`);
3. the Python bridge dependencies (`pip install -r bridge/requirements.txt`), now including Pillow for vision-sensor image handling;
4. SWI-Prolog ≥ 9.0 for DALI2;
5. (optional) JAN [1] (or any OpenAI-compatible local server) with a vision-capable model for the image-analysis loop.

The end-to-end startup is now a single command (after opening CoppeliaSim with an empty scene):

```
python launch.py      # starts Redis, builds scene, DALI2, bridge
```

The launcher accepts flags such as `–skip-redis`, `–config`, and `–openrouter KEY` (to route VLM requests to OpenRouter instead of a local server) for incremental use. Alternatively, the four steps can still be run in separate terminals as before.

Qualitative observations. With the default scenario (2 light + 1 heavy victim, 8 obstacles):

- All three victims are typically delivered to the depot in $\sim$60–100 s, depending on the order in which sightings reach the coordinator. The heavy victim accounts for the majority of the wall-clock time, since it requires a rendezvous and a formation-follow carry phase.
- After delivering a victim, every rescuer resumes exploration within one agent cycle (the `deliveredE` handler transitions directly to `state(exploring)` and sends `explore`), ensuring that robots never stall at the depot.

Table 2
Mission performance (5 runs per condition, 300 s timeout). Values are mean ± std. dev.

Metric	VLM on	VLM off
Success rate	100 % (5/5)	80 % (4/5)
Mission time (s)	95.1 ± 16.1	155.8 ± 81.3
First sighting (s)	4.6 ± 0.6	10.4 ± 4.7
Pre-filter skip rate	76 %	n/a
LLM calls per run	39.8 ± 12.1	n/a
Heavy-lift sync (s)	7.8 ± 3.7	10.0 ± 7.8

- The green-pixel pre-filter eliminates >90 % of LLM calls: in a typical run only a handful of screenshots contain green pixels, so the local Qwen3.5-9B is invoked rarely, keeping CPU/GPU load manageable even on modest hardware.
- The told/tell priority queue keeps the coordinator responsive: thousands of `position_update` messages per minute (priority 10) never starve the rare but critical `victim_seen` (priority 200) and `ready_to_lift` (priority 140) messages.
- The web UI's `/api/beliefs` endpoint makes the coordinator's state inspectable: at the moment `sync_lift` fires, both `ready_at_t/2` beliefs are visible, confirming that both partners were in position.

Effect of the vision pipeline. The bridge exposes a `VISION_LLM_ENABLED` flag that disables the vision pipeline entirely, leaving only the 3 m proximity sensor as the sole detection modality. We ran five missions per condition (5 min timeout); Table 2 summarises the results.

With the vision pipeline *enabled*, all five missions complete successfully in 95.1 ± 16.1 s on average. First sightings occur at 4.6 s, roughly half the time required by the proximity-only baseline (10.4 s), confirming that the VLM acts as an effective early-warning channel. The green-pixel pre-filter skips ∼76 % of frames, limiting LLM queries to ∼40 per mission and keeping computational overhead low on commodity hardware. With the pipeline *disabled*, one run out of five fails to rescue all victims within the 5 min budget (2/3 rescued) and mean mission time increases to 155.8 s with much higher variance. Overall, the vision-augmented configuration improves both reliability (100 % vs. 80 % success) and efficiency (∼39 % shorter missions).

Replay and inspection. Because every DALI2 event is recorded in an append-only past-memory store, an entire mission can be replayed off-line by injecting the same sequence through `/api/inject`. Combined with CoppeliaSim's deterministic fixed-step solver, this gives a reasonable degree of reproducibility: the same scenario file plus the same injection trace yields nearly identical timings on a single machine, modulo OS scheduling jitter and Redis publish latency.

7. Discussion

Where logic helps. The auction, the cooperative lift and the priority queue together amount to roughly one screen of DALI2 code. Each of those constructs maps directly to a single linguistic primitive of the language; the resulting program is short enough to be reasoned about manually, and the engine's introspection facilities (past memory, beliefs, learned associations, goal status) make it inspectable at runtime. This is especially valuable in safety-critical settings, where one must be able to explain *why* a given action was taken.

Where LLMs help. Conversely, the deterministic logic layer is poorly suited to rich, partially observable scene understanding ("is this body still breathing?"; "does this debris look unstable?"). Our system uses LLMs in two complementary roles: (a) a *text-based triage advisor* invoked via `ask_ai/2`,

and (b) a *vision-based perception module* that analyses camera screenshots through a local Qwen3.5-9B (or, optionally, OpenRouter's gpt-4o) and returns structured JSON objects parsed into typed Prolog events. A fast green-pixel pre-filter avoids the LLM call entirely for the majority of frames. Both channels are kept strictly advisory: the symbolic auction enforces correctness properties, while `told` rules act as a typed firewall on what advice is even accepted. The exploration loop adds a further dimension: robots proactively patrol the arena, photograph their surroundings, and act on the analysis (obstacle avoidance, victim reporting), closing the sense-think-act cycle entirely within the DALI2 event model. This is a strict generalisation of the pattern proposed in [10] for grounding natural-language plans in a robot's action vocabulary: in our case the action vocabulary is itself a set of FIPA proposals, which the language already knows how to validate.

From simulation to the real world. Because the bridge is a separate process speaking a small, well-typed protocol, replacing CoppeliaSim with a real-robot back-end is, in principle, mechanical: a ROS 2 node would subscribe to the same LINDA messages on `sim` and command real wheels. We have not yet performed this transfer, but the architecture is intentionally agnostic to the embodiment, in line with CARLA's emphasis on simulation-first methodology. Detection radius, navigation gains, arena bounds and victim detection are all parameters of the bridge: tightening them brings the digital twin closer to a noisy real-world deployment without changing a single line of agent code. Similarly, the green-pixel pre-filter – effective in simulation where victims are uniformly coloured – would be replaced in a physical deployment by a task-specific detector (e.g. a lightweight CNN or YOLO model trained on victim appearance); the downstream VLM triage and the coordination logic would remain unchanged, since the bridge's contract with the agents is limited to the typed `vision_result/1` event.

Limitations. We position this work as a *cognitive-architecture case study* rather than a general robotics solution: the emphasis is on validating the coordination logic in isolation from low-level dynamics. Our navigation controller is a kinematic unicycle model integrated directly via `setObjectPosition`/`setObjectOrientation`; while this simplifies development, it bypasses CoppeliaSim's physics entirely, so the robots cannot collide with each other or push objects. The clearance-based obstacle avoidance picks the less obstructed side but does not consider dynamic obstacles or other robots; a real deployment would need a proper local planner. The auction is non-iterative: once a heavy task has only one bidder it simply waits for the next solicitation cycle, which can cause delays if the team is small. The cooperative-lift synchronisation currently relies on persistent beliefs rather than a bounded temporal window (`past_event` lifetime); adding the window is a one-line change and is planned for the next iteration. The vision LLM's latency (typically 1–5 s for the local Qwen3.5-9B) means that the robot pauses briefly at each waypoint; the green-pixel pre-filter mitigates this by skipping most calls, but adaptive capture intervals or motion-triggered snapshots would help further. Finally, both the text and vision LLM oracles remain advisory – a more aggressive integration (e.g. letting the model issue `propose/1` terms) would require a sandboxing strategy that we leave to future work.

8. Related work

DALI joins a long line of declarative agent platforms. Among the most influential, AgentSpeak/Jason [11] formalises BDI agents in a Prolog-like language; JaCaMo [12] layers Jason agents over the CArtAgO artefact-based environment and the Moise organisational model. GOAL [13] pursues a similar goal-oriented agent specification. Compared to these, DALI's distinctive feature is the fine-grained event ontology – external, internal, present, past, multi-event with delta-t – and integrity constraints as first-class language primitives.

On the robotics side, ROS and ROS 2 [14] are the de-facto middleware standard, and several behaviour-tree libraries (BehaviorTree.CPP, py_trees) provide a more declarative alternative to plain callbacks. Compared to behaviour trees, the DALI/DALI2 event model emphasises message-driven interaction over hierarchical control, and integrates naturally with multi-agent FIPA protocols. Recent work on

neuro-symbolic robot planning [9, 15] sits in the same broader space as our LLM oracle but typically targets single-agent task-and-motion planning rather than multi-robot coordination.

CoppeliaSim has been used as a digital twin in many published multi-robot studies [7, 5]; what we add here is a symbolic, declarative cognitive architecture that treats the twin as just another peer in a Redis-mediated star topology.

9. Conclusion

We have shown that DALI2, a modern reimplementation of the event-driven logic-programming language DALI, can drive a multi-robot search-and-rescue mission end-to-end inside a CoppeliaSim digital twin, including the non-trivial case of two robots cooperating to lift a heavy victim via a coordinator-mediated synchronisation protocol. The architecture is deliberately thin: a Python bridge routes messages between Redis pub/sub and the ZMQ Remote API using kinematic unicycle control, while the agent program is itself only a few hundred lines of DALI syntax. We extended the system with on-board vision sensors, a fast green-pixel pre-filter, and a local vision LLM (Qwen3.5-9B through JAN, or OpenRouter gpt-4o), closing the perception loop: robots autonomously patrol the arena, photograph their surroundings at each waypoint, skip LLM analysis when no green pixels are present, and then decide whether to report a victim, avoid an obstacle, or continue exploring – all without breaking the symbolic coordination guarantees.

Going forward, we plan to (i) replace the bridge's kinematic unicycle controller with a learned navigation policy, evaluating whether the symbolic layer can act as a safety filter on its outputs; (ii) add a bounded temporal window (`past_event` lifetime) to the cooperative-lift synchronisation and declare integrity constraints (`:~`) for safety invariants; (iii) lift the same agent file unchanged onto a fleet of TurtleBot 4 robots through a ROS 2 back-end; (iv) stress-test the vision LLM on noisy, cluttered scenes to characterise the degradation profile of the combined system; and (v) let the LLM propose navigation waypoints directly, subject to symbolic validation by the agent layer.

Acknowledgments

The author thanks the DALI community for their long-standing work on event-based logic programming and the CoppeliaSim team for maintaining a research-friendly digital twin.

Declaration on Generative AI

During the preparation of this work, the author used an LLM-based coding assistant in order to scaffold the repository and produce code skeletons for the DALI2/CoppeliaSim bridge. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] Homebrew Computer Company, Jan: Open-source ChatGPT alternative that runs 100% offline on your computer, https://jan.ai/, 2024. Accessed: 2026-01-20.

[2] S. Costantini, A. Tocchio, A logic programming language for multi-agent systems, in: Logics in Artificial Intelligence, 8th European Conference, JELIA 2002, volume 2424 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 1–13. doi:10.1007/3-540-45757-7_1.

[3] S. Costantini, A. Tocchio, The DALI logic programming agent-oriented language, in: Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, volume 3229 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 685–688. doi:10.1007/978-3-540-30227-8_57.

[4] AAAI-DISIM-UnivAQ, AAAI-DISIM-UnivAQ/DALI2, 2026. URL: https://doi.org/10.5281/zenodo.19858727. doi:10.5281/zenodo.19858727.

[5] Coppelia Robotics, CoppeliaSim ZeroMQ remote API, https://manual.coppeliarobotics.com/en/zmqRemoteApiOverview.htm, 2024. Accessed: 2026-01-15.

[6] S. Costantini, G. D. Gasperis, P. Migliarini, Multi-agent system engineering for emphatic human–robot interaction, 2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE) (2019) 36–42. doi:10.1109/AIKE.2019.00015.

[7] E. Rohmer, S. P. N. Singh, M. Freese, V-REP: A versatile and scalable robot simulation framework, in: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2013, pp. 1321–1326. doi:10.1109/IROS.2013.6696520.

[8] R. G. Smith, The contract net protocol: High-level communication and control in a distributed problem solver, IEEE Transactions on Computers C-29 (1980) 1104–1113. doi:10.1109/TC.1980.1675516.

[9] A. d'Avila Garcez, M. Gori, L. C. Lamb, L. Serafini, M. Spranger, S. N. Tran, Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning, Journal of Applied Logics 6 (2019) 611–632.

[10] W. Huang, P. Abbeel, D. Pathak, I. Mordatch, Language models as zero-shot planners: Extracting actionable knowledge for embodied agents, Proceedings of the International Conference on Machine Learning (ICML) (2022).

[11] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak using Jason, John Wiley & Sons, 2007.

[12] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, A. Santi, Multi-agent oriented programming with JaCaMo, Science of Computer Programming 78 (2013) 747–761. doi:10.1016/j.scico.2011.10.004.

[13] K. V. Hindriks, Programming rational agents in GOAL, Multi-Agent Programming: Languages, Tools and Applications (2009) 119–157. doi:10.1007/978-0-387-89299-3_4.

[14] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, ROS: An open-source robot operating system, in: ICRA Workshop on Open Source Software, 2009.

[15] M. Ahn, A. Brohan, N. Brown, et al., Do as I can, not as I say: Grounding language in robotic affordances, in: Proceedings of the 6th Conference on Robot Learning (CoRL), 2022.