The Six-Room Survey

An EPDDL Domain Where the Map Supplies the Knowledge,
Executed on ePlanSys

Haniel Ulises

Epistemic Robotics

August 26, 2026

A single robot, six rooms, and a goal that no classical planner can state: not to reach the east wing, but to *know* whether it is passable. The observation is not imagined by the planner: it is produced by `plansys2_epistemic_perception` from an occupancy grid the robot's own laser builds while it drives.

1 What the domain has to express

A survey mission over a building is a poor fit for classical planning, and the reason is not the geometry. A robot can be told to drive to the far side of a building and a metric planner will route it there. What it cannot be told is that the mission succeeds only once the robot *knows* whether the far side is passable, a condition about the robot's information rather than about the world.

The distinction has teeth when the answer is not determined in advance. Rooms nobody has entered are neither known to be clear nor known to be blocked, and a plan that assumes either is unexecutable in half of the situations it will meet. What the mission needs is a policy: drive somewhere the question can be answered, answer it, and continue accordingly.

This report describes such a domain, `building-rooms`, together with its solution, and its execution on ePlanSys, together with the defect that currently stops the execution half short.

2 The domain

The building has six rooms. Rooms 3 and 6 lie east of a cross corridor and are the subject of the survey; the robot starts in room 1 at the west end. Four propositions suffice:

`blocked-r3`, `blocked-r6`	whether each east room is obstructed
`at-door3 ?i`, `at-door6 ?i`	whether `?i` stands where that room can be seen into

Two ontic actions drive to the doorways, and are public: in a fleet every robot tracks the others' positions, so the effect is common knowledge. Two sensing actions look in. Their type is what carries the epistemic content:

```
(:action inspect3
  :parameters (?i - agent)
  :action-type (semi-private-sensing
    (e-inspect3-blocked ?i)
    (e-inspect3-clear ?i) )
  :observability-conditions
    (:and
      (?i Fully)
      (default Partially) ) )
```

Semi-private sensing says that the agent which looks learns the answer, while anyone else present sees it look without learning what it saw. With one robot the distinction is invisible; modelling it as fully public would nevertheless be a lie, and one that stops being harmless the moment a second robot is added.

Each sensing action carries two events, one per answer, and neither is known to apply when the plan is built. That is precisely what makes the solution branch.

2.1 The instance

What the initial state does *not* say is whether either room is obstructed. The theory leaves both open, so the model designates four worlds rather than one, since a building nobody has surveyed is genuinely undetermined:

```
(:init
  (:and
    (:forall (?i - agent) ([C. All] (not (at-door3 ?i))))
    (:forall (?i - agent) ([C. All] (not (at-door6 ?i))))
    (:forall (?i - agent) ([C. All] (<Kw. ?i> (blocked-r3))))
    (:forall (?i - agent) ([C. All] (<Kw. ?i> (blocked-r6)))) ) )

(:goal (and ([Kw. r1] (blocked-r3)) ([Kw. r1] (blocked-r6))))
```

The goal is Kw_{r_1} *blocked-r*$_3 \wedge Kw_{r_1}$ *blocked-r*$_6$: knowing *whether*, not knowing *that*. A mission that succeeds only when the rooms turn out to be clear is not a mission, and a plan for it would be unexecutable in the half of the worlds where they are not.

3 Grounding and search

Grounding with PLANK yields one agent, four atoms, four grounded actions and an initial model of four worlds, all designated:

```
agents: ['r1']
atoms: ['at-door3_r1', 'blocked-r6', 'at-door6_r1', 'blocked-r3']
actions: ['goto-door3_r1', 'goto-door6_r1', 'inspect3_r1', 'inspect6_r1']
initial worlds: 4 designated: 4
```

ALETHEIA selects its heuristic and strategy from the task's own features rather than from a flag, a goal of `Kw` conjuncts over a small designated set, and solves it at depth four:

```
[main] Heuristic: knowledge-spread (auto, rule 'kw-only-goal')
[main] Strategy: AO* (auto, rule 'sensing-small-designated')
[aostar] Solution found at depth 4 Expanded=32 Generated=35 Memo=6/6
[validator] OK -- 4 leaves, 6 branches checked
```

The policy is a tree, not a sequence:

```
goto-door3_r1
  ev0: inspect3_r1
    ev0: goto-door6_r1 -> inspect6_r1 # room 3 seen blocked
    ev1: goto-door6_r1 -> inspect6_r1 # room 3 seen clear
```

Both branches continue to the second room, and they are distinct nodes rather than a shared subtree: the executor runs one of them, and which one is decided by an observation that has not been made when the plan is built.

4 Where the observation comes from

The point of the demonstration is that nothing in the stack invents the answer. The perception node watches an occupancy grid, classifies a named region of it, and reports what it found as the outcome of the sensing action the policy branched on:

```
epistemic_perception:
  ros__parameters:
    regions: ["room3", "room6"]
    room3:
      boxes: [4.25, 0.25, 11.8, 7.8]
      atom: "blocked-r3"
      atom_true_when_clear: false
      sensing_action: "inspect3_r1"
      outcome_when_clear: "e-inspect3-clear"
      outcome_when_blocked: "e-inspect3-blocked"
```

A region counts as *clear* only when every cell in it has been observed and settled, and as *blocked* when any one cell is occupied; anything else is undecided. Occupied beats unobserved, because no further looking removes an obstacle, and unobserved beats free, because a region is clear only when all of it is.

The grid is built by `slam_toolbox` from the robot's own laser as it drives. This is what makes the demonstration honest rather than arranged: a room the robot has not entered is unobserved in the grid and undetermined in the Kripke model at the same time, and those are two views of one fact. Nothing publishes a prepared map.

5 Execution

The six-node epistemic bringup comes up on this domain (domain expert, problem expert, planner, executor, epistemic state, epistemic perception) and the planner answers with the policy above. The action mapping joins the two vocabularies, `inspect3_r1` on the epistemic side being `(look_into r1 door3)` in PDDL, and the executor dispatches it:

```
[planner] [epistemic] policy with 6 nodes, branching
[executor] Accepted new goal
[perception] 'room3' is blocked: inspect3_r1 observed e-inspect3-blocked
[epistemic] applied inspect3_r1 -> e-inspect3-blocked
[bt] (look_into r1 door3) observed e-inspect3-blocked
[epistemic] applied goto-door6_r1
[perception] 'room6' is clear: inspect6_r1 observed e-inspect6-clear
[bt] (look_into r1 door6) observed e-inspect6-clear
[bt] [goal] (and (Kw r1 blocked-r3) (Kw r1 blocked-r6))
Successfully finished
```

The two rooms answer differently, so the branch is exercised rather than implied: the policy takes its blocked continuation out of room 3 and its clear one out of room 6, and the goal holds because the agent has come to know both propositions, from a map it built itself while driving.

5.1 What the run cost

Measured against Gazebo's own poses for the robot rather than against its odometry, which keeps integrating while a base is held against a wall and so cannot answer the question at all:

Distance driven	27.88 m
Closest approach to any obstacle	0.307 m
Robot body radius	0.105 m
Contacts	0
Recoveries from a stall	0

6 Five defects the demonstration found

Building it was the test. Each of these stopped the run, and each is fixed.

A dangling reference in the executor. `EpistemicBTBuilder` terminated with `SIGSEGV` whenever a policy was rendered. `ExecutorNode` iterated `get_parameter("bt_node_plugins").as_string_array()` directly in a range-`for`. That accessor returns a reference *into* the `rclcpp::Parameter` the call returned by value; the Parameter is a temporary and dies at the end of the full expression, so the loop walked a destroyed vector and handed `dlopen` a freed pointer, faulting inside `strchr`, as the loader scanned the name for a dynamic string token. It bites only when `bt_node_plugins` is non-empty, which is exactly the epistemic configuration and never the classical one, which is why no existing test caught it.

A parameters file that could not be loaded. The shipped `plansys2_epistemic_params.yaml` wrote `regions: []`. An empty list in a parameters file has no element type, so the override reaches the node with no value and construction throws: the very trap the same file documents twenty lines earlier for `epddl_libraries`. Launching with `epistemic_perception:=True` therefore aborted every time.

Rolling-only APIs on a Humble target. Five packages used `rclcpp::experimental::executors::EventsExecutor` and `rclcpp::ServicesQoS()`, neither of which exists in Humble, so the fork could not be built on the distribution its own CI names. `plansys2_core/Compat.hpp` now selects between them by asking whether the header exists rather than what the distribution is called.

A refusal that was only a race. Perception reports the moment a region resolves; the epistemic state learns the robot reached the doorway only when the executor applies the preceding action. Observed gap: 78 ms. The state refused the early observation as a divergence between model and

world, and perception, by design, never retried a refusal. A reading the model was about to be ready for was thrown away permanently. Perception now distinguishes *not applicable yet* from a real disagreement and retries the first for a bounded number of grids.

An update applied twice. With the observation landing first, the behaviour tree's own `ApplyEpistemicUpdate` then applied the same sensing action again and was refused, because its `Kw` precondition no longer held because it had succeeded. The state now answers a re-applied action with the outcome it already produced, and repeats no update. Every other inapplicable action is still refused exactly as before.

7 What a sensing action turned out to be

The demonstration also settled a modelling question the domain alone does not answer: what a robot should physically *do* when the policy says `inspect3`.

A timed pause is the obvious reading and the wrong one: it cannot tell success from failure, and it ends on a stopwatch rather than on knowledge. A rotating sweep works but only as a workaround: `slam_toolbox` integrates nothing from a base that never moves, so the robot was spinning to keep its own mapper alive, which reads as confusion rather than intent.

What the action does now is drive into the room, turn to face the region, hold still, and end exactly when `(Kw r1 blocked-r3)` holds in the epistemic state, or fail and say so if the region never settles. A sensing action exists to acquire knowledge, so knowledge is what ends it.

One consequence is worth recording, because it is a fact about sensors and not about the model. A region is *blocked* as soon as one cell in it is occupied, and *clear* only when every cell has been observed and settled. The first is decided in an instant; the second cannot be decided at all for a region lying in open floor, because a 3.5 m laser returns nothing in a 7.5 m room and free space is only traced along beams that come back. The region a robot can settle is the region its beams cross on the way to a wall.

8 Reproducing

```
# ground and solve, without ROS
plank export -d building-rooms.epddl -p instances/problem_1.epddl \
     -l intermediate.epddl -o out
epistemic_planner --task out/problem_1.json --plan out/plan_1.json --explain

# the full system
ros2 launch eplansys_rooms_demo rooms_demo_launch.py
```

The domain and instance are in `epddl-workspace/building-rooms`; the ROS side, including the two action nodes, the parameters and the launch file, is in `ros2_ws/src/eplansys_rooms_demo`.