Deliverable 3: Semantic Robotics - IoT Remote Lab

Gorka Vila Pérez
Chair of Embedded Systems and Internet of Things
Technical University of Munich
Munich, Germany
gorka.vila@tum.de

***Abstract*—This report details the engineering process involved in implementing an end-to-end cube sorting workflow for Deliverable 3: Semantic Robotics of the IoT Remote Lab. The system integrates heterogeneous Web of Things (WoT) devices discovered through a Thing Directory: two Uarm robots, an UR3 robot, two conveyor belts, infrared distance/presence sensors, and a color sensor. The final pipeline executes three stages: (i) Uarm1 picks a cube from a spawn location and places it on ConveyorBelt1 (CB1), (ii) Uarm2 stops CB1 using infrared sensing, adapts the pickup location based on measured distance, and transfers the cube onto ConveyorBelt2 (CB2), and (iii) UR3 stops CB2, picks the cube with a distance-based pose correction, performs color detection at a dedicated sensing station while holding the cube, and sorts it by color. The work emphasizes practical robustness techniques for non-deterministic cyber-physical integration: TD-driven discovery, semantic unit normalization across randomized TDs, explicit synchronization between stages, bounded waiting for asynchronous actuation, sensor-derived corrective offsets, and graceful shutdown coordination between processes.**

***Index Terms*—Web of Things, Thing Description, Node-WoT, Thing Directory, semantic heterogeneity, sensor-driven control, industrial robotics, CoppeliaSim**

I. Discovery & Methodology

A. Initial Exploration ("Step 0")

The Thing Directory at `http://localhost:8081/things` lists two UR3 robots, one UR3, two conveyors, four infrared sensors, a color sensor, and two virtual lights. The Thing Description (TD) files were already present in the project folder, enabling a thorough review of device functionalities.

Initial analysis identified significant heterogeneity: Uarms provide `goTo(x,y,z)` for position control, whereas the UR3 requires `goToPosition(x,y,z,rx,ry,rz)` for six-degree-of-freedom pose specification. Gripper actions also differ, with `gripOpen/gripClose` for Uarms and `openGripper/closeGripper` for UR3. These differences necessitated device-type-specific dispatching within the control logic.

B. Component-Level Testing

A bottom-up integration strategy was adopted:

1) **Robots:** Commanded single moves to safe coordinates and verified `currentPosition` convergence. Initial attempts failed because actions returned immediately. Adding a closed-loop wait that polls `currentPosition` every 100ms until all axes are within 20mm tolerance (with a 30s timeout and retry/backoff on transient HTTP errors) fixed this.
2) **Conveyors:** Started/stopped CB1 and observed cube motion in CoppeliaSim to estimate transport time.
3) **Sensors:** Polled `objectPresence` and `objectDistance` to characterize behavior. In the final pipeline, both pickup stations use a calibrated infrared trigger distance of 0.275m with a ±0.01m window, plus a stabilization dwell after belt stop to reduce variability.
4) **Color sensor:** Read RGB tuples and implemented a simple threshold-based classifier consistent with the code: red if R$>$100 and G,B$<$100; blue if B$>$100 and R,G$<$100; green if G$>$100 and R,B$<$100; otherwise unknown.

This incremental approach ensured that each subsystem was validated prior to pipeline integration, thereby reducing the scope of debugging.

II. Implementation Process

A. Evolution from Prototype to Full System

Initial attempts used a single sequential script to pick and place a cube, but it proved brittle when a second cube was introduced, causing workspace collisions between Uarm1 and Uarm2. The system was restructured into three concurrent stages, coordinated through explicit Boolean handshakes. Each stage operates in an independent asynchronous loop, enabling continuous operation.

In the final implementation, the handshakes are explicit "ready/busy" flags between adjacent stages: Stage 1 and Stage 2 coordinate with `stage2ReadyForCube`, and Stage 2 and Stage 3 coordinate with `stage3ReadyForCube`. This keeps ownership of shared resources (especially conveyor start/stop) unambiguous.

B. Architecture

The final system consumes WoT Things via Node-WoT's HTTP binding and runs three parallel control loops for a continuous pipeline:

Stage 1 (Uarm1 → CB1): Picks cube at spawnPickPosition $[-0.147, 1.320, 1.016]$, lifts to spawnLiftPosition $[-0.147, 1.320, 1.131]$, moves to conveyor1AbovePosition $[0.135, 1.550, 1.225]$, lowers to conveyor1PlacePosition $[0.135, 1.550, 1.105]$, releases after a 500ms settling delay, returns above the belt, and then back toward the spawn lift

waypoint. Stage 1 does *not* start CB1; Stage 2 owns CB1 start/stop to avoid race conditions. Handshake: Stage 1 waits for `stage2ReadyForCube==true` and then signals cube availability by setting `stage2ReadyForCube=false`.

Stage 2 (CB1 → Uarm2 → CB2): Takes control of CB1 as soon as Stage 1 signals a cube. It starts CB1, uses sensor2 (`objectPresence`) as early warning, then monitors sensor1 for `objectPresence=true` and `objectDistance` close to the trigger setpoint $d_{\text{target}} = 0.275\,\text{m}$ with tolerance ±0.01m. After stopping CB1 and waiting for stabilization (500ms + 1500ms), it measures the final stop distance d_s and adapts the pickup X coordinate around conveyor1PickPosition $[1.205, 1.5675, 1.080]$ using a direct correction:

$$x_{\text{pick}} = x_{\text{nominal}} - (d_s - d_{\text{expected}}), \quad d_{\text{expected}} = 0.16\,\text{m} \quad (1)$$

Uarm2 lifts and transfers the cube to conveyor2DropPosition $[1.460, 1.285, 1.100]$ (with a relaxed tolerance of 0.1m for the drop), waits for Stage 3 readiness (`stage3ReadyForCube==true`), releases, then starts CB2.

Stage 3 (CB2 → UR3 → Color Sort): Uses sensor4 (and sensor3 as a fallback) to detect an approaching cube, then in parallel moves UR3 above the pickup pose and monitors sensor3 distance to stop CB2 at $d_{\text{target}} = 0.275\,\text{m}$ ±0.01m. After stopping and stabilization (500ms + 1500ms), it applies a direct correction to the pickup Y coordinate of conveyor2PickPosition $[1.475, 0.090, 1.120]$ (with $ry = -90°$):

$$y_{\text{pick}} = y_{\text{nominal}} + (d_s - d_{\text{target}}) \quad (2)$$

UR3 then lifts, moves through ur3IntermediatePosition $[1.053, -0.0876, 1.174]$ (with $rx = 179.1966°, ry = -89.8671°, rz = -109.2019°$), holds the cube at colorSensorDropPosition $[0.800, 0.220, 1.145]$ (with $rx = 90°, ry = -90°$) and waits up to 2s for `objectPresence` at the color sensor. The final drop is computed from the sensor pose: red uses $y = 0.22 + 0.25 = 0.47$, blue uses $y = 0.22 - 0.25 = -0.03$, and green drops at the intermediate waypoint.

C. Position Setpoints (Code-Aligned)

Tables I and II list the exact world-frame setpoints used by the implementation (units: meters for x, y, z; degrees for rx, ry, rz).

TABLE I
UARM SETPOINTS USED IN STAGES 1–2

Waypoint	x	y	z
spawnPickPosition	-0.147	1.3200	1.016
spawnLiftPosition	-0.147	1.3200	1.131
conveyor1AbovePosition	0.135	1.5500	1.225
conveyor1PlacePosition	0.135	1.5500	1.105
uarm2HomePosition	1.205	1.5675	1.160
conveyor1PickPosition	1.205	1.5675	1.080
conveyor2DropPosition	1.460	1.2850	1.100

TABLE II
UR3 SETPOINTS USED IN STAGE 3

Waypoint	x	y	z	rx	ry	rz
conveyor2PickPosition	1.475	0.0900	1.120	0.0000	-90.0000	0.0000
ur3IntermediatePosition	1.053	-0.0876	1.174	179.1966	-89.8671	-109.2019
colorSensorDropPosition	0.800	0.2200	1.145	90.0000	-90.0000	0.0000

D. Key Control Patterns

Sensor-driven stopping: After initial failures with fixed timing (cubes arrived 1–2s early/late), I switched to distance-based triggers. Cubes are detected at ∼0.3m by presence sensors, tracked continuously, and the belt stops when the distance crosses into the target window.

Wait-for-position convergence: A helper `uarmGoToAndWait()` invokes movement then polls `currentPosition` every 100ms (with retry/backoff on transient HTTP failures) until *all* axes are within tolerance (L∞ check) or a 30s timeout. This prevents gripper actions mid-motion.

III. PROBLEMS ENCOUNTERED & TROUBLESHOOTING

A. Problem 1: Uarm2 Timeout Cascade

Symptom: During Stage 2, Uarm2 frequently logged "Timeout waiting to reach position" even though visual inspection showed the arm near the target. Subsequent cycles failed completely.

Investigation: I added debug logging of `currentPosition` every 100ms. Logs revealed that the action returned immediately (HTTP 200), but the position continued to change for 2-3 seconds afterward. My initial code invoked `gripClose` 200ms after `goTo` returned, causing the gripper to close mid-motion at the wrong height-cube missed.

Root cause: WoT action invocations are fire-and-forget; `goTo` starts motion but doesn't block until completion.

Solution: Implemented `waitForUarmAt(target, tolerance=0.02, timeout=30s)` that polls `currentPosition` every 100ms and waits until $|x - x_t|, |y - y_t|, |z - z_t| \leq 20\,\text{mm}$. Applied this after every critical move. This eliminated the "close mid-motion" failure mode.

B. Problem 2: Non-Deterministic Cube Stopping

Symptom: After switching to sensor-driven stopping, cubes still stopped at varying positions (sensor1 reading ranged 0.16–0.21m across runs). Uarm2 pickup succeeded only ∼60% of the time.

Investigation: I logged sensor1 distance every 50ms during approach. The data showed the cubes decelerated unevenly, sometimes stopping 20mm short and sometimes 30mm past the target, so belt inertia and cube friction may have varied per run in the simulation.

Solution iteration 1: Added a short settling delay after `stopBelt` (500ms) plus a longer stabilization dwell (1500ms) to let the cube settle.

Solution iteration 2: Captured the final stop distance d_s after stabilization and computed a direct 1:1 correction: $x_{\text{offset}} = -(d_s - d_{\text{expected}})$ with $d_{\text{expected}} = 0.16\,\text{m}$. The 1:1 offset aligns sensor error to the pickup axis and remains stable in practice.

C. Problem 3: Semantic Unit Mismatch in Randomized TDs

Symptom: In the random-units mode, sensor distances sometimes appeared an order of magnitude off, leading to incorrect pickup offsets (e.g., treating 0.275 as 0.275mm).

Investigation: TDs occasionally advertised `unit: "millimeter"` while the simulation still returned meters. The conversion logic blindly trusted the TD, leading to values being divided by 1000.

Solution: Implemented unit normalization and a small heuristic in `readDistance`: if a TD claims millimeters but the value is in the meter range ($<$10), treat it as meters. This resolved the mismatch for randomized TDs and stabilized Stage 2/Stage 3 pickup offsets.

D. Problem 4: CoppeliaSim Crashes on Startup

Symptom: Running the full system (transportation + redLight + blueLight concurrently) could overload the HTTP bridge/Thing Directory and destabilize the simulator.

Investigation: Examined process logs; saw hundreds of HTTP requests/sec to Thing Directory and devices. Color sensor polling (at a 10ms interval in light scripts) hammered the HTTP server.

Solution: Reduced unnecessary polling (light scripts check presence/color at 500ms rather than aggressive intervals) and added retry/backoff behavior in closed-loop waits to tolerate transient `/HTTP` failures. The current `npm start` uses `concurrently` to run all three controllers, relying on these rate limits and shutdown coordination for stability.

E. Problem 5: Graceful Shutdown on Windows

Symptom: Concurrent processes did not terminate cleanly when the main controller exited, leaving simulation tasks running.

Solution: Added a filesystem-based shutdown flag (`.factory-shutdown`) that is created by the controller and polled by auxiliary scripts. This ensures a deterministic, cross-platform shutdown sequence without relying on POSIX signals.

IV. Adapting to Heterogeneity

A. Tackling Semantic Heterogeneity

The primary challenge involved integrating devices with fundamentally different interfaces, despite standardized TD descriptions. This was addressed at three levels:

Device abstraction layer: Rather than scattering device-specific logic throughout stages, I created adapter functions. For movement, `uarmGoToAndWait()` handles 3-axis Uarms while UR3 uses direct `goToPosition()` with 6-DOF. Gripper control similarly dispatches: `gripOpen/gripClose` for Uarms, `openGripper/closeGripper` for UR3. This kept the stage logic device-agnostic.

Unit and coordinate handling: TDs can vary in unit declarations (e.g., meters vs millimeters). I normalized units with `convertToMeters/convertFromMeters` and expanded unit synonyms (cm, mm, metre). For randomized TDs, I added a guard heuristic to check whether values appear to be meters despite a millimeter label. For coordinates, I avoided assumptions and instead calibrated fixed-world-frame poses per device: Uarm2 operates around $y \approx 1.57$ at the CB1 station. At the same time, UR3 picks at CB2 using conveyor2PickPosition with $y = 0.09$, and transitions through an intermediate waypoint with $y = -0.0876$. These values look inconsistent at first glance, but they are consistent within the shared CoppeliaSim world frame and the robots' kinematic reach.

Heterogeneous reliability requirements: Sensor properties differ semantically: `objectPresence` (boolean, event-like) vs `objectDistance` (continuous float). My control loops poll distance continuously but use presence as a pre-filter so this hybrid approach handles both semantic types efficiently.

B. Logic for Cross-Run Robustness

Initial hardcoded sequences failed during repeated runs. Three adaptive mechanisms were implemented:

Sensor-feedback control: Replaced timing-based belt stopping with sensor-driven triggers. For Stages 2 and 3, the controller first gates on `objectPresence=true` and then monitors `objectDistance` until it enters a calibrated window centered at $0.275\,\text{m}$ ($\pm 0.01\,\text{m}$). Once within the window, the belt is stopped, and the final stop distance is sampled after stabilization to compute the pickup correction.

Dynamic position correction: After the belt stops, the cube can settle short/long. The implemented controller captures the final stop distance and applies a direct (1:1) corrective offset instead of a tuned scale/clamp. For Stage 2: $x_{\text{pick}} = x_{\text{nominal}} - (d_s - d_{\text{expected}})$ with $d_{\text{expected}} = 0.16\,\text{m}$. For Stage 3: $y_{\text{pick}} = y_{\text{nominal}} + (d_s - d_{\text{target}})$ with $d_{\text{target}} = 0.275\,\text{m}$. This keeps the adjustment interpretable and consistent with the physical geometry.

State-based handshakes: Stage 1 waits for `stage2ReadyForCube==true` before producing a cube, then signals handoff by setting `stage2ReadyForCube=false`. Stage 2 waits for `stage3ReadyForCube==true` before releasing onto CB2. This prevents race conditions and makes conveyor ownership deterministic.

C. Handling Randomness Issues

Three randomness sources required specific solutions:

Cube stop position variance (±30mm): Initially caused 40% pickup failures. First fix: added a 1.5s stabilization delay after stop to reduce bounce. Second fix: adaptive offset based on measured distance (above). Third fix: widened gripper approach tolerance from 20mm to 100mm for CB2 drop (less

critical phase). The combined approach reduced failures to less than 2%.

Random unit declarations in TDs: When TDs randomly declared mm/cm while devices returned meters, conversions skewed by 10×–1000×. The fix was twofold: (1) unit normalization with expanded synonyms, and (2) a sanity check that treats meter-range values as meters even if the TD labels millimeters.

Color misclassification: To reduce misclassification, the controller holds the cube at the calibrated color-sensor pose and waits (up to 2s) for the sensor `objectPresence` signal to assert before classifying. The implemented classifier is intentionally conservative and straightforward (channel threshold 100 with mutual exclusion). If no class is confidently detected ("unknown") or a color quota is already met, the cube is skipped and released at a safe pose rather than being missorted.

Asynchronous action completion: Robot actions can return immediately while motion continues. Fixed delays were unreliable; the final solution is a polling-based `waitForUarmAt()`, which samples `currentPosition` every 100ms until convergence or a 30s timeout.

V. Discussion

A. Scaling to Industrial Deployment

In a factory with more than 50 robot types, manual TD inspection for each device becomes infeasible. The following strategies are recommended:

Automated capability profiling: A discovery agent that queries Thing Directory, parses all TDs, and builds a capability matrix (action signatures, coordinate systems, gripper types). Schema validation (e.g., JSON Schema) ensures TDs conform to expected patterns before runtime.

Semantic matching: Instead of hardcoding "Uarm" vs "UR3" checks, use ontology-based reasoning. Tag actions with semantic annotations (e.g., `@type: MoveToPosition`) and dispatch based on semantics, not device names. This enables zero-code integration of new robot types conforming to the ontology.

Calibration routines: On first device connection, run auto-calibration: command small moves, read feedback, infer workspace bounds, and coordinate transforms, store calibration in a persistent registry.

B. Additional Industrial Heterogeneity

Beyond this lab, real factories face:

Kinematic constraints: Joint limits, singularities, payload capacity. TDs should expose these as machine-readable constraints; path planning must respect them.

Safety interlocks: Emergency stops, light curtains, collaborative operation zones. Require extending WoT with safety property subscriptions and event-driven halting.

Communication protocol diversity: Mix of OPC UA, Modbus, MQTT, proprietary. WoT bindings abstract this, but latency/reliability vary. Critical paths need QoS-aware protocol selection.

Time synchronization: Multi-robot coordination (this lab used coarse handshakes) requires sub-millisecond sync in real deployments. TDs should expose timing capabilities; control logic must account for clock drift.

C. Conclusion

This laboratory exercise demonstrated that WoT Thing Descriptions facilitate discovery and basic interoperability. However, achieving production-level robustness requires additional layers, including explicit wait-for-completion patterns for asynchronous actions, sensor-feedback control to address non-determinism, and adaptive correction based on measured state. The transition from a system that functions once to one that operates reliably is achieved through systematic troubleshooting, empirical tuning, and defensive programming. These insights are broadly applicable to cyber-physical system integration.