

Applications of Autonomous BDI Agents in Robotics

By Bardia Parmoun



Supervisor: Dr. Babak Esfandiari



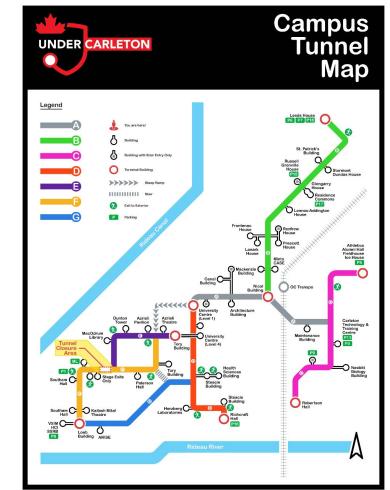
Systems and Computer Engineering



● Motivations

- Demonstrate a complex use case for Jason
- Illustrate BDI's advantage for implementing established patterns such as the subsumption architecture
- Analyze the tradeoffs of BDI for design and performance

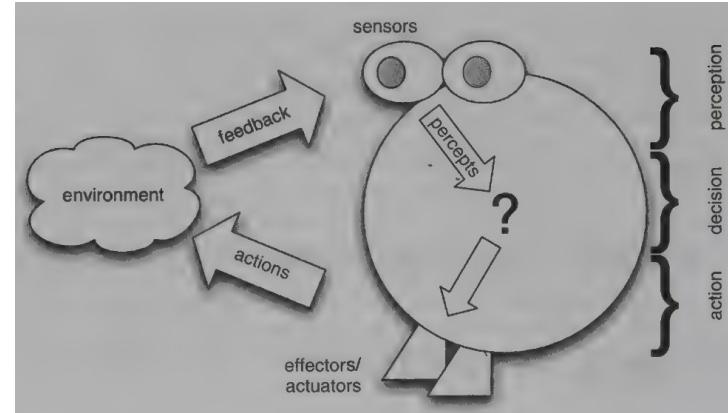
- **Problem Statement**
 - Navigating the Carleton University tunnels autonomously



What is an agent?

“An agent is a **computer system**, situated in some **environment**, that is capable of **flexible autonomous action** in order to meet its **design objectives**”

- Jennings et. al



What is BDI?



- **BELIEFS:** agent's knowledge about the world
- **DESIREs:** the goals for the agent
- **INTENTIONS:** agent's plans for achieving its goals



Beliefs

Goals

Rules

Plans

```
wallFollowAimAngle(45).  
wallFollowDistanceSetpoint(0.5).  
!wallFollow(0.8, 32.0).  
tooFarFromWall(WallDistance) :-  
    wallFollowDistanceSetpoint(SETPPOINT) &  
    WallDistance > SETPOINT.  
tooCloseToWall(WallDistance) :-  
    wallFollowDistanceSetpoint(SETPPOINT) &  
    WallDistance < SETPOINT.  
+!wallFollow(WallDistance, WallAngle):  
    not (tooFarFromWall(WallDistance)) &  
    not (tooCloseToWall(WallDistance))  
    <-  
    .print("Following the wall...");  
    !turn(WallAngle).  
+!wallFollow(WallDistance, WallAngle):  
    tooFarFromWall(WallDistance) &  
    wallFollowAimAngle(WALL_FOLLOW_AIM_ANGLE)  
    <-  
    .print("Too far from the wall...");  
    !turn(-1 * WALL_FOLLOW_AIM_ANGLE +  
          WallAngle).  
+!wallFollow(WallDistance, WallAngle):  
    tooCloseToWall(WallDistance) &  
    wallFollowAimAngle(WALL_FOLLOW_AIM_ANGLE)  
    <-  
    .print("Too close to the wall...");  
    !turn(WALL_FOLLOW_AIM_ANGLE + WallAngle).
```

- **AgentSpeak:** a theoretical formal language for BDI
 - Formal notation for the agent's beliefs, goals, and plans
- **Jason:** one of its most popular implementations!
 - The agent repeatedly perceives, reasons, and acts!



- **ROS2**: a set of libraries and frameworks used to develop robotic applications.



- **Gazebo**: a dedicated simulator environment for ROS2 with full support for most sensors.



- Continuation of “Towards Campus Mail Delivery using BDI” → Increasing complexity

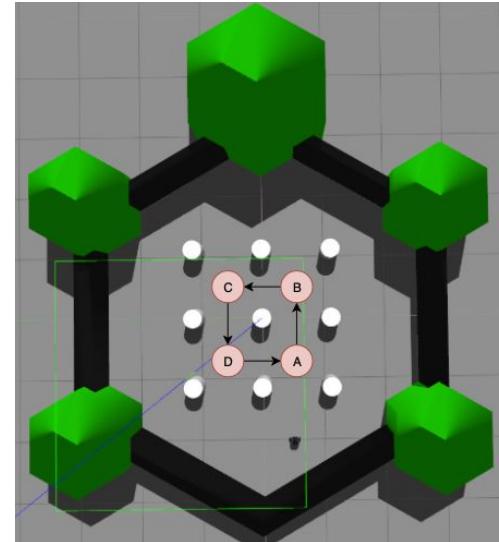
C. Onyedinma, P. Gavigan, and B. Esfandiari,
“Toward campus mail delivery using bdi,” Journal of
Sensors and Actuators, vol. 9, no. 56, pp. 127–143,
2020



(a)



(b)

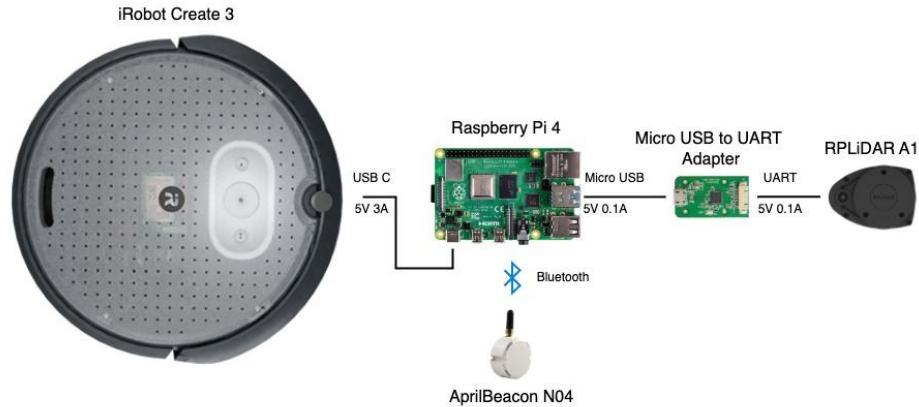


- Most existing implementations are either not realistic or fully simulated

Q. Du and R. Cardoso, “Evaluating bdi agents in ros: From basic integration to fault tolerant multi-robot systems,” in European Conference on Multi-Agent Systems (EUMAS 2025), July 2025. The 22nd European Conference on Multi-Agent Systems, EUMAS 2025



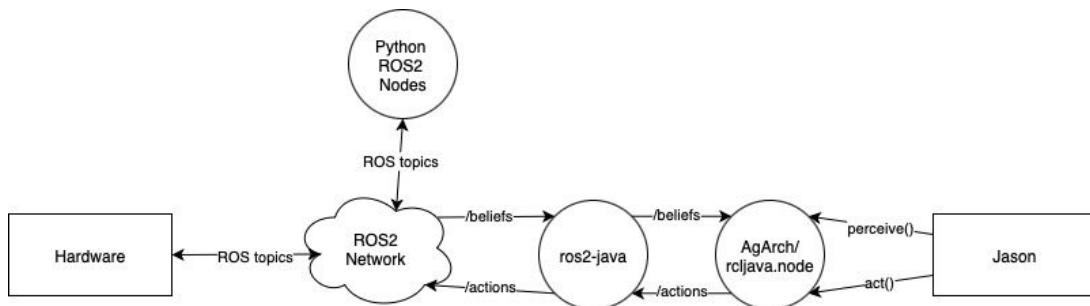
- **Hardware Design:**
 - iRobot CREATE 3
 - Raspberry Pi 4
 - RPLiDAR A1
 - AprilBeacon N04



	Benefits	Drawbacks
JROS [14]	<ul style="list-style-type: none"> Easy and simple implementation. 	<ul style="list-style-type: none"> Only works with the legacy ROS1. Does not work with Jason's Java interface.
Jason-ROS [15]	<ul style="list-style-type: none"> Supports Jason's Java interface. 	<ul style="list-style-type: none"> Only works with the legacy ROS1. Limits the ability to customize the AgArch class.
Rason [15]	<ul style="list-style-type: none"> Supports Jason's Java interface. Provides a straightforward interface for communicating with Jason. 	<ul style="list-style-type: none"> Only works with the legacy ROS1. Limits the ability to customize the AgArch class.
Agent-in-a-Box [17]	<ul style="list-style-type: none"> Very easy to set up. Provides a lot of the functionalities needed to develop a BDI robot. 	<ul style="list-style-type: none"> Only works with the legacy ROS1. Limits the ability to customize the AgArch class.
ROS-A [18]	<ul style="list-style-type: none"> Supports Jason's Java interface. Works really well with ROS 2! 	<ul style="list-style-type: none"> The setup process is a bit involved and has the overhead of requiring a ROS1 to ROS bridge. Limits the ability to customize the AgArch class.

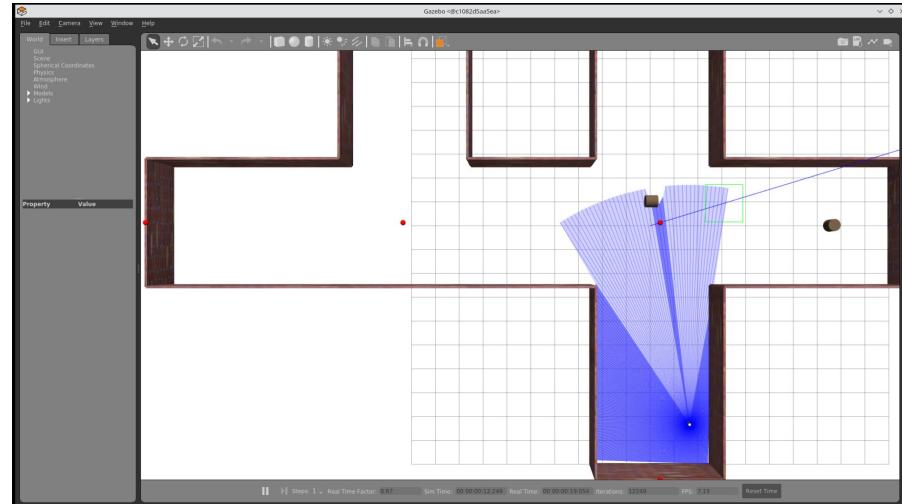
Existing methods were
not applicable!

- Software Design:
- Using **ros2-java**
- An **rcljava** node acting as **AgArch**



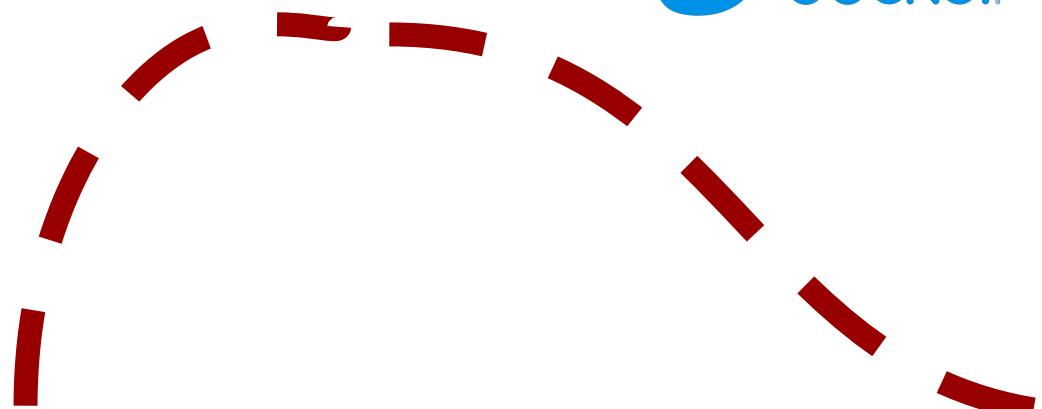
- **Simulator Design:**

- Implemented with Gazebo
- Customizable map/obstacles
- Containerized with Docker

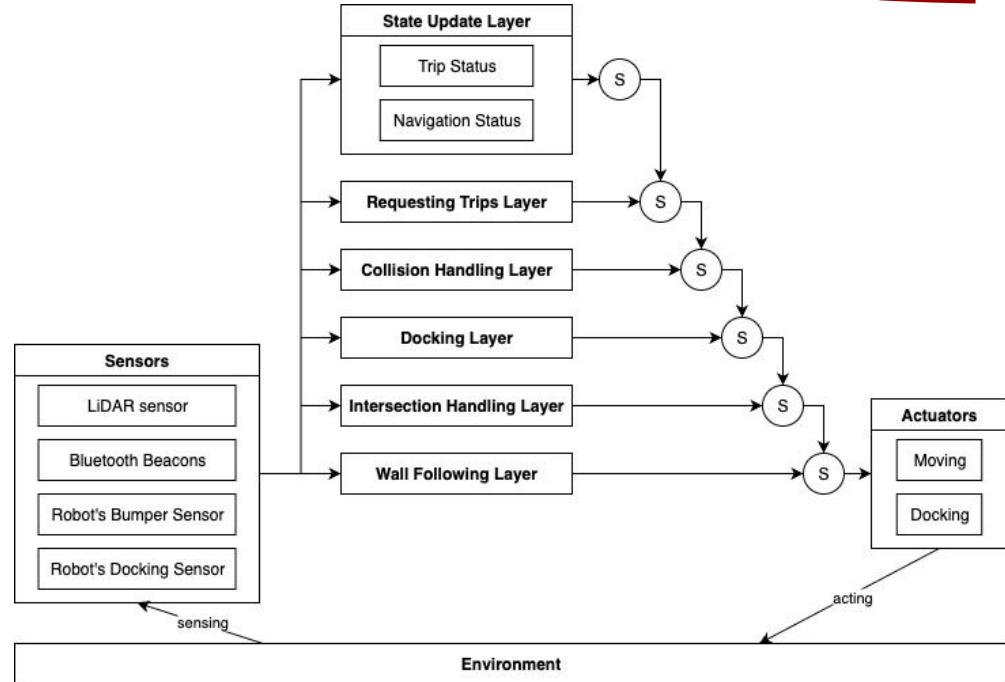


- **Components:**

- iRobot CREATE 3 model
- RPLiDAR A1 model
- Custom beacons using RF plugins
- Gazebo models for walls/obstacles



	Benefits	Drawbacks
Imperative Programming	<ul style="list-style-type: none"> • Easy to implement. • Requires minimal design and engineering efforts. 	<ul style="list-style-type: none"> • Not scalable and hard to maintain. • Lacks explainability at a large scale
Finite State Machines	<ul style="list-style-type: none"> • Very explainable. • Can easily identify and cover corner cases. • Global information can be easily stored and retrieved. 	<ul style="list-style-type: none"> • Grows fast in size as the problem gets more complex. • Behaviours cannot be defined separately due to the concurrent nature of the inputs, leading to the creation of many steps for joint behaviours.
Subsumption Architecture	<ul style="list-style-type: none"> • Clearly defines a notion of priority for concurrent behaviours. • Behaviours can be defined independently. • Can easily scale and adapt to new behaviours. 	<ul style="list-style-type: none"> • Cannot add central states for custom management of the behaviours. • The notion of subsumption and priorities might not be plausible for systems that have concurrent behaviours with similar priorities.

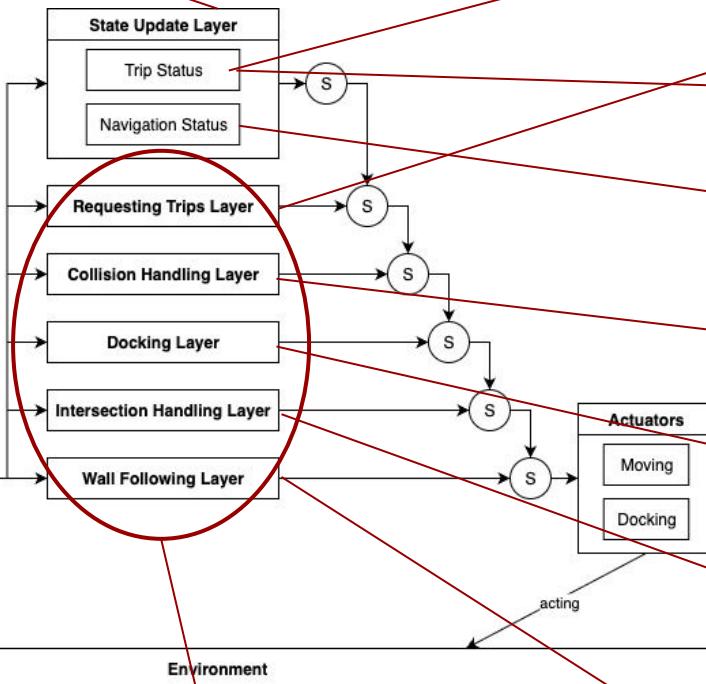
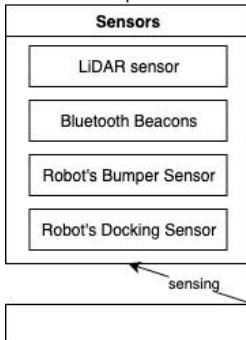


Proposed BDI design

Popular paradigms for designing robot behaviours

DESIRE

BELIEFS



ACTIONS

INTENTIONS

```

/* Initial Goals */
!requestTrip.

/* Main Behaviour */
// Trip Completion
-hasTrip: true
<-
.print("Completed a trip!");
!requestTrip.

// No Trips
+navigationInstruction(none): true
<-
.print("No more trips!");
-requestingTrip;
!requestTrip.

// New Trip
+navigationInstruction(start): true
<-
.print("Got a new trip!");
-requestingTrip;
+hasTrip.

// Navigation
+navigationInstruction(NavInstruction):
    hasTrip
    <-
    .print("Navigation: ", NavInstruction);
    +navigation(NavInstruction).

// Collision Detection
+bumperPressed: hasTrip &
    not(.intend(handleCollision))
<-
.drop_all_intentions;
.print("Bumper was pressed! Backing up");
!handleCollision.

// Dock Station Detection
+dockVisible: hasTrip &
    navigation(dock) &
    not(.intend(handleDocking)) &
    not(.intend(handleCollision))
<-
.drop_all_intentions;
.print("Reached the destination!");
!handleDocking.

// Intersection Detection
+intersection(ForwardDistance, LTurnDistance,
    UTurnDistance): hasTrip &
    navigation(NavInstruction) &
    facingWall(WallDistance, WallAngle) &
    not(navigation(dock)) &
    not(.intend(handleIntersection(_,_,-_)));
    & not(.intend(handleDocking)) &
    not(.intend(handleCollision))
<-
.drop_all_intentions;
.print("Reached an intersection!");
!handleIntersection(ForwardDistance,
    UTurnDistance, WallAngle).

// Wall Detection
+facingWall(WallDistance, WallAngle): hasTrip
    & not(.intend(wallFollow(_,_)))
    & not(.intend(handleIntersection(_,_,-_)))
    & not(.intend(handleDocking)) &
    not(.intend(handleCollision))
<-
.print("Wall following...");

wallFollow(WallDistance, WallAngle).
    
```



- **Findings:**

- Jason is capable of handling **real-time** environments.
- BDI could be used **in conjunction** with well established robotic design patterns such as the **subsumption architecture** and **state machines**.
- Jason make the robot implementation **transparent** while still being **efficient**.
- BDI robots are **explainable!**



- **Future Work:**

- Convert the project into a playground for comparing different agent implementations
→ project **Hermes**
- Connect the Jason implementation to linear and non-linear planners using Peleus



Demos!!