Setting up a Jason Agent on Top of ROS to Control an Autonomous UAV

Felipe da C. Calegari[1], **Iago O. Silvestre**[1], **Jomi F. Hübner**[1], **Leandro B. Becker**[1], **Maiquel de Brito**[2]

[1]Programa de Pós-Graduação em Engenharia de Automação e Sistemas
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC

[2]Departamento de Engenharia de Controle, Automação e Computação (CAC)
Universidade Federal de Santa Catarina (UFSC) – Blumenau, SC

felipecunhacalegari@gmail.com,iagosilvestre2004@gmail.com,

{jomi.hubner,leandro.becker,maiquel.b}@ufsc.br

Abstract. *This paper describes the nuances related to properly setting up a Jason agent on top of the Robot Operating System (ROS) to control an autonomous Unmanned Aerial Vehicle (UAV) executing in a Gazebo simulation. To do so we refactor a previously developed search-and-rescue (SAR) UAV application with updated technology. We mainly address changing the interaction mechanism between the agent and ROS, i.e., the way that the agent interacts with its "physical body". We also address using Docker technology to setup the design environment and speed up the design process. Our results showed that it can be challenging to adapt agents since different technologies have different setups and different compatibilities, but it comes with advantages such as scalability in agent programming and constant updates to work with different ROS versions.*

1. Introduction

Currently, there is no standard way to architect the interaction between an agent and its "physical body", for instance the Unmanned Aerial Vehicle (UAV). However, as highlighted in [Dennis and Fisher 2023], the *Robot Operating System* (ROS)[1] has gained attention as an adequate technology to abstract from the agent the implementation details concerning interacting with a physical device, i.e., connecting to sensors and actuators.

On the other hand, cognitive agents with BDI (***B**elief-**D**esire-**I**ntention*) architecture [Bratman et al. 1988] are known for facilitating the design of high-level tasks such as reasoning, percepting the environment, setting goals, and performing decision-making. For such a reason, over the last years, several works addressed the design of UAV control systems using BDI architectures, such as the Jason language [Rafael H. Bordini and Wooldridge 2007]. Some examples can be seen in [Menegol et al. 2018, Silva et al. 2021, Silvestre et al. 2023].

Different technologies were created to allow the interaction between BDI agents and ROS: ROS-A [Cardoso et al. 2020], Jason-ROS [Silva et al. 2020], and E-MAS [Silvestre et al. 2025]. For instance, ROS-A considers ROS as part of the environment in which the agent acts, so that the actions upon ROS are treated as actions upon

[1]https://ros.org/

elements external to the agents (*external actions*). In Jason-ROS the agents have their execution system integrated with an *agent node*, which has topics to record perceptions to be handled and actions triggered by the agents. These topics are linked to the "physical body" via a ROS node named *bridge*, which can be configured according to the application hardware. E-MAS also allows this configuration, but it does not rely on bridging nodes. The agent execution system is connected to the hardware nodes and can read topics to get perceptions, as well as write in topics and request services to perform actions. Moreover, while E-MAS is in constant update, Jason-ROS became deprecated.

The work in [Silva et al. 2021] describes an interface to connect a Jason agent with ROS and employs it in a search-and-rescue (SAR) application. This application has been refactored to use E-MAS instead of Jason-ROS to empower Jason agents with up-to-date technologies for integration with ROS. This paper describes this refactoring, highlighting advantages and disadvantages of such update, as well as the pros and cons from both interface mechanism. Besides, we also discuss how to overcome the technological problems related to updating ROS/Gazebo and Linux-Ubuntu versions. This allows us the following:

- Empower the Jason-agents with up-to-date technologies;
- Discuss advantages and disadvantages related with such update and, consequently, the pros and cons from both interface mechanisms.

The remainder parts of this paper are organized as follows: Section 2 presents the technologies (languages, frameworks, interfaces) used throughout this work. Section 3 presents the application scenario in adaptation. Finally, Section 4 presents final considerations and future work directions.

2. Related Technologies

The section starts by presenting the Jason language. Then it presents the ROS framework. Afterward, it tackles the architecture of agents, discussing the mechanisms to interface Jason and ROS.

2.1. Jason

AgentSpeak [Rao 1996] is one of the most influential abstract languages based on the BDI framework. *Beliefs* are information the agent has about the world, other agents, and itself. *Desires* are all possible states of affairs that the agent might like to accomplish. *Intentions* are the states of affairs that the agent has decided to work towards.

Jason [Rafael H. Bordini and Wooldridge 2007][2] is an interpreter for an enhanced version of AgentSpeak. It implements the operational semantics of this language and offers a platform for developing Multi-Agent Systems (MAS). Jason is implemented in Java (thereby it is multi-platform) and is made available as open-source software.

2.2. ROS

ROS [Quigley et al. 2009, Koubaa 2017] is an open-source set of libraries and tools for developing robotic applications. It employs a messaging system that enables the communication among various *nodes* (programs) within a system. It uses *topics*, which works as

[2] `https://jason-lang.github.io`

a message-passing mechanism. *Nodes* can publish messages to a particular *topic* and can also subscribe to receive messages from topics, facilitating the exchange of information such as sensor data, motor commands, or other state-related data. A given node can also request specific actions or responses from other nodes by means of *services*. A service client can send a request to a service server, which processes the request and returns a response. Services are commonly used for tasks that require a particular action to be carried out. Although ROS 2 is the most up-to-date version, the current work uses *ROS Noetic*.

2.3. Agent Architecture

As previously mentioned, there are different technologies to allow the interaction between Jason and ROS, such as ROS-A [Cardoso et al. 2020], Jason-ROS [Silva et al. 2020], and E-MAS [Silvestre et al. 2025]. This paper focuses the last two, as follows.

2.3.1. Jason-ROS

The Jason-ROS framework uses Jason agents to integrate with robot systems that use the ROS platform. This framework changes the agent's main architecture on how it perceives and acts on the environment in which it is situated, giving advantages such as control and optimization of the system's integration. This framework uses specific configuration files stating the agents' actions (for actuators) or perceptions (for sensors). In these files, the developer needs to set up the method used (topic or service, for example), the path of this method, message type, dependencies, and the necessary parameters for the message to be properly sent. In the agent code, there is no difference since the name of the action used by the agent is already stated in the configuration file.

The framework also uses four ROS nodes to integrate Jason with ROS. The first is called *Agent* that is the agent itself. The second is called *HW-Bridge* which is responsible to integrate the hardware and Jason agents. The third is the *Hardware Controller*, responsible for the management of hardware. The final node is called *Comm*, responsible for the communication protocols between the agents [Silva 2020].

2.3.2. E-MAS

The Embedded-Mas (E-MAS) framework can integrate ROS with BDI agents, specifically to the agent's perceptions and actions. The same way as the Jason-ROS framework, E-MAS extends the default Jason agents class *ag-class* with the *RosBdiAgent* class to be able to include ROS topics among the perception sources and link the actions to the service calls and topic writings. It also extends the default Jason architecture *ag-arch* with *RosBdiAgArch*, which is responsible for including what has been written in the topic when the agent perceives the environment.

The agent code (in an.asl file) does not have significant changes; actions, perceptions, and beliefs are coded in the same way. The integration between the ROS and the agents is set up in a .yaml file where each agent has its own file. This file makes it possible to define the Java classes that implement the interface between Jason and ROS. Furthermore, the item ***perceptionTopics*** configures the topics that produce perceptions

and the item ***actions*** configures the connection between the actions of the agent and the ROS resources that control the actuators that realize such actions.

2.4. PX4

PX4 is an open-source flight control software commonly used in UAV development. It provides support for autonomous flight, sensor integration, and control modes. In this work, PX4 runs in Software-in-the-Loop (SITL) mode, allowing simulations to be executed without physical hardware.

To connect PX4 with ROS, we use MAVROS, which acts as a bridge between the PX4 autopilot and the ROS environment. MAVROS exposes topics and services used to arm the UAV, set flight modes, publish target positions, and access telemetry data such as position and state.

This setup enables the Jason agents to issue commands (e.g. takeoff, land, fly to a position) and perceive the UAV state through ROS. All interactions happen within a simulated environment using Gazebo, with PX4 controlling the low-level UAV behavior.

The Docker image used in this project (further detailed in 3.1) includes PX4, MAVROS, ROS Noetic, and all required dependencies. This ensures the simulation behaves consistently across different systems and makes the environment easier to reproduce.

3. Case Study

The presented work consists of the adaptation of the search-and-rescue (SAR) application presented in [Silva et al. 2021], which aims to rescue humans drowning in the open sea. The scenario[3] was run on Gazebo Simulator and Rezende's Jason-ROS framework was used to integrate the agents system with ROS.

In this SAR application, there are two types of UAVs: (i) the *scouter*, responsible for flying above the water searching and identifying humans and then sending location information to the second type of agent; (ii) the *rescuer*, which receives information about the location of the human, goes to the specific coordinates, and then releases a buoy to save the human. Figures 1 and 2 contain flowcharts representing the behavior of the scout and rescuer agents, respectively. The scout's and rescuer's beliefs, goals, and plans, along with their descriptions are presented in Tables 1 and 2.

The *Scout* agent is set up to start with the initial beliefs such as *water offset*, a *search area*, *flight altitude*, and a *setpoint goal*. After, the agent flies with maximum horizontal speed, maximum altitude the UAV would return, and the path that would be calculated where the UAV would fly through. This calculated path generates a list of coordinated points, and if a victim is detected in the specific area, the *scout* agent would alert the *Rescuers* agents with the location. Since it's a multiagent system, and in specific with the possibility of multiple *rescuer* agents, once the location is sent from the *scout* agent, they (*rescuers*) negotiate to decide who will drop the buoy at the victim's location based on who's the closest. Table 3 contains each agent's action and perception, and the respective data types used by the agents.

[3]https://github.com/Rezenders/active-perception-experiments

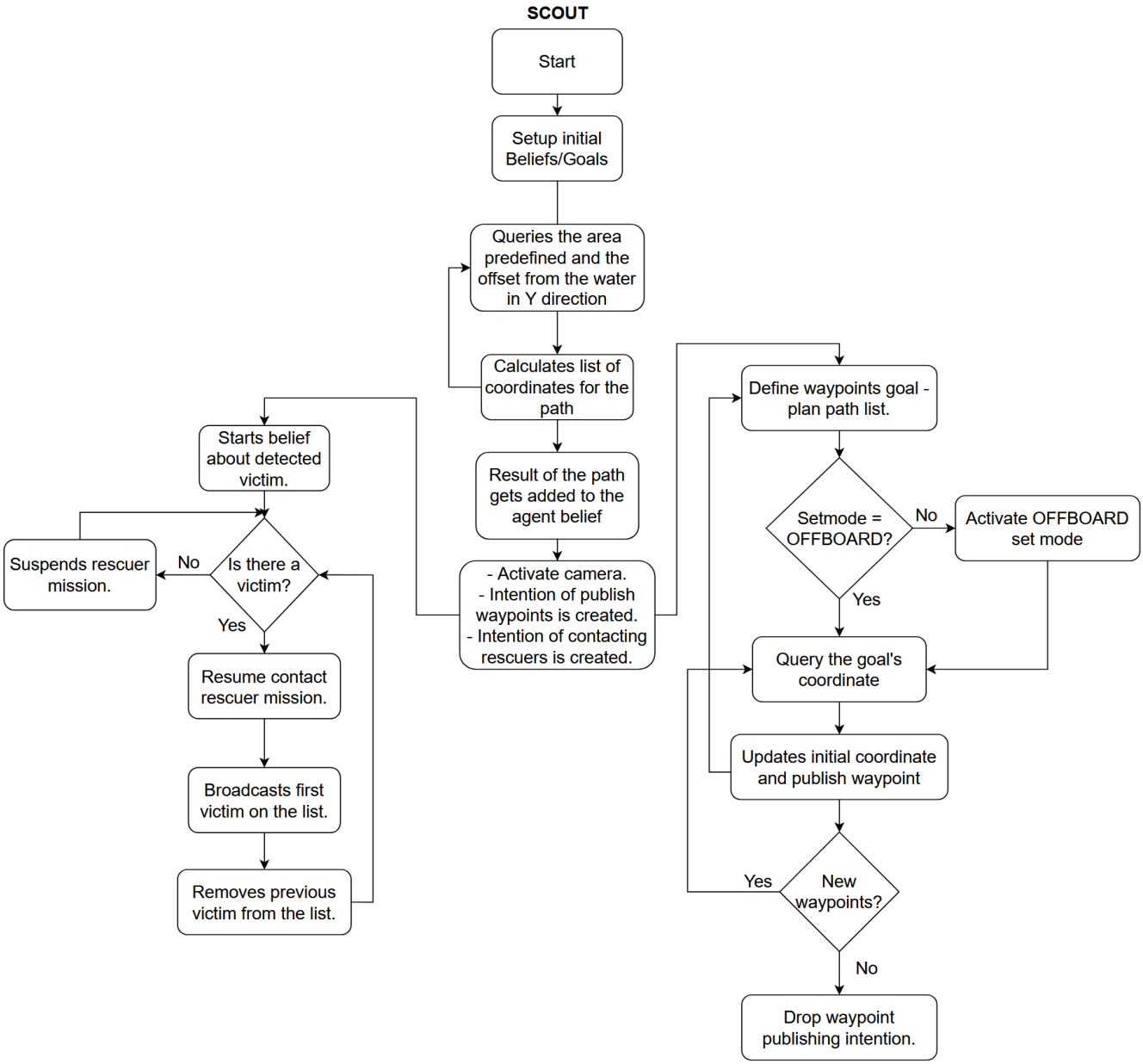

Figure 1. Scout Agent Behavior Diagram

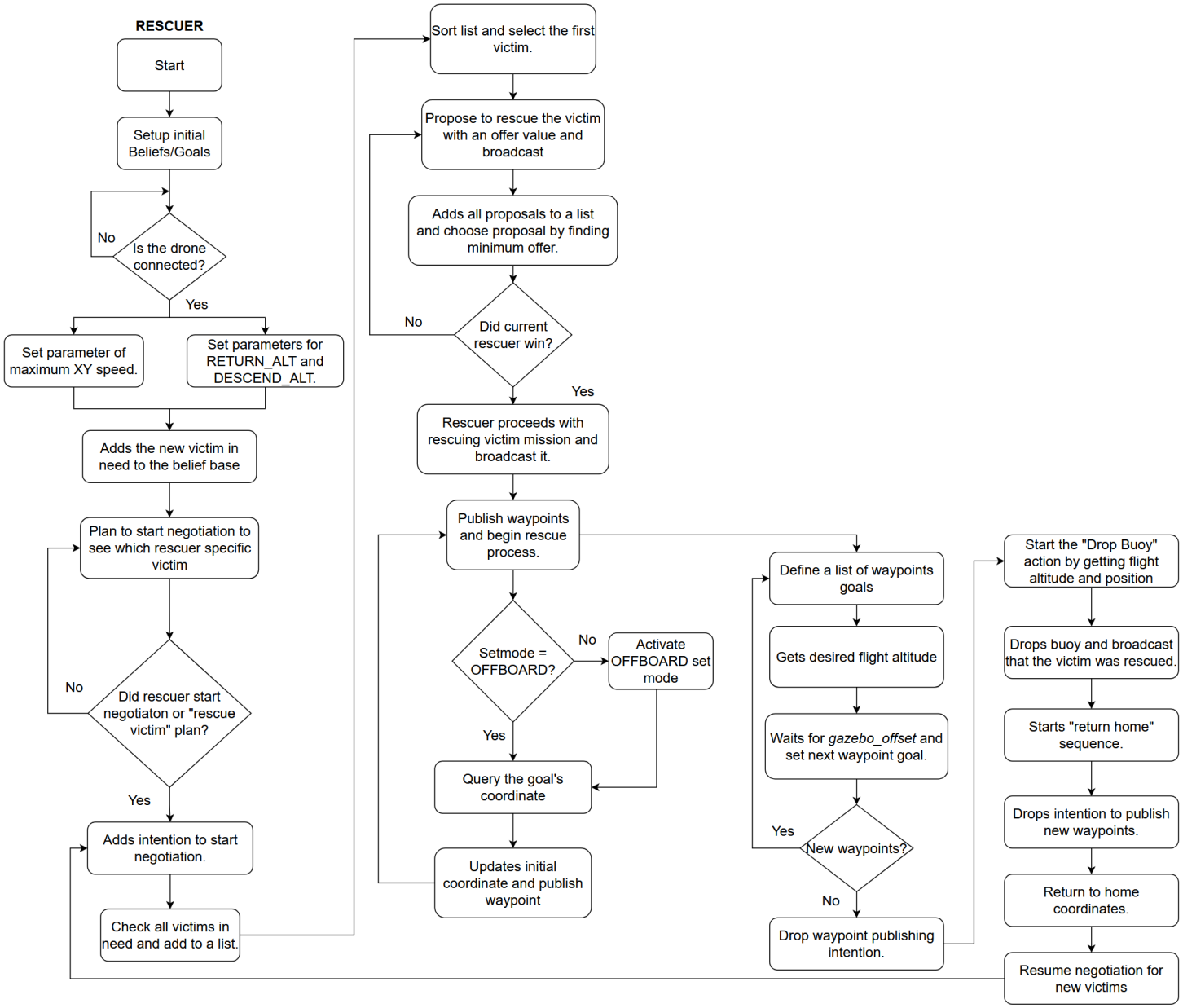

Figure 2. Rescuer Agent BehaviorDiagram

Table 1. List of Beliefs, Goal and Plans - Scout agent

Agent Property	Name	Description
Belief	water_Y_offset	An offset in Y direction for path planning
	search_area	Size of the area that should be searched
	flight_altitude	Initial altitude that the UAV would fly
	setpoint_goal	Initial coordinates goal
	plan_path_result	Add a belief that contains the result of the plan path and starts the mission
	victim	Adding to the belief base the position of a victim
	victim_position	Given the position of the victim, resume the plan to contact the rescuers
Goal	setRTLAtlitude	Altitude for "Return to Lauch"
	setMaxSpeed	Maximum speed
	planPath	Path planning
Plans	setMaxSpeed	Maximum speed
	setRTLAtlitude	Set the "Return to Launch Altitude" to a given value
	planPath	Plan a box-shaped path for the UAV
	defineGoal	Define new waypoints
	publishSetPoint	Check for UAV's mode, check for waypoint (X, Y, Z coordinates) and publish them
	contactRescuers	Inform the rescuers agents about victim position
	informVictims	Obtain victim's coordinates and broadcast
	mark_as_rescued	Mark which victim was rescued

Table 2. List of Beliefs, Goal and Plans - Rescuer agent

Agent Property	Name	Description
Belief	flight_altitude	Default flight altitude
	setpoint_goal	Initial setpoint - coordinates
	victim_position	Location of the victim
	victim_in_need	Victim in certain location needs to be rescued
	victim_in_rescue	Rescue of the victim is the current process
	gazebo_pos	Position in Gazebo simulator
	gazebo_offset	Calculated offset between Gazebo and UAV's coordinates
Goal	getGazeboOffset	Calculate simulation offset
	setRTLAtlitude	Altitude for "Return to Launch"
Plans	setMaxSpeed	Maximum speed
	setRTLAtlitude	Altitude for "Return to Launch"
	victim_in_need	Register the victim at their coordinates and starts negotiation
	start_negotiation	Start negotiation for rescue
	negotiate	Find and propose rescue for victims
	propose	Propose the rescue by rescuer's name
	broadcastProposal	Broadcast the proposal
	choose_proposal	Choose which rescuer is the best
	check_winner	Check if the agent is elected for rescue
	rescueVictim	Execute rescue mission
	publishSetPoint	Publish waypoints to rescue the victim
	defineGoalLocal	Get list of waypoints, adjust with Gazebo's offsets and updates the list of waypoints
	drop_buoy	Drop the buoy at the victim's location
	resume_negotiation	Resume negotiation after rescue
	returnHome	Return to home coordinates and land
	getGazeboPos	Get position based on Gazebo simulation
	getGazeboOffset	Calculate offset from simulation
	mark_as_rescued	Mark victim as rescued

In table 3, the actions and perceptions with * were responsible for some slight modifications of the source code of the agents. This happened because there are some slight differences in how these two interfaces construct and interpret ROS messages.

As previously mentioned, the E-MAS framework has been used in this work to integrate the Jason agents with ROS. The focus of this section is explaining the differences in setting up both environments, the main difficulties observed for adapting it, and the pros/cons.

Table 3. List of Actions, Perceptions and their Data Types

	Name	Method	Data Type
Actions	set_mode	service	SetMode
	arm_motors	service	CommandBool
	takeoff	service	CommandTOL
	setpoint_global	topic	GeoPoseStamped
	setpoint_local*	topic	geometry_msgs/PoseStamped
	land	service	CommandTOL
	plan_path*	topic	boustrophedon_msgs/PlanMowingPathActionGoal
	set_fcu_param	service	mavros_msgs/ParamSet
	drop_buoy	topic	geometry_msgs/Pose
	camera_switch	service	std_srvs/SetBool
Perceptions	state*	topic	State
	local_pos	topic	geometry_msgs/PoseStamped
	planPathResult*	topic	boustrophedon_msgs/PlanMowingPathActionResult
	modelStates	topic	gazebo_msgs/ModelStates
	victim	topic	geometry_msgs/PoseStamped

3.1. Environment setup

The environment itself was adapted to work properly with a Docker container. Docker is a containerization platform that enables the creation of portable and consistent environments for software development and deployment. By bundling an application with all its dependencies, Docker ensures that it runs reliably across different systems. In robotics and simulation, where toolchains often involve complex configurations—such as ROS, Gazebo, and PX4—Docker significantly simplifies setup, improves reproducibility, and reduces system-specific issues during development.

For the search-rescue-px4 project [4], a custom Docker image was created to support autonomous UAV simulations using PX4 SITL, ROS Noetic, Gazebo 11, and MAVROS. The image is based on the osrf/ros:noetic-desktop-full base and includes the full PX4 firmware, MAVROS dependencies, Jason interpreter and additional system tools useful for simulation. The Dockerfile also sets up the ROS workspace and sources the appropriate environment variables, allowing for immediate use of ROS and PX4 tools within the container.

This containerized environment improves consistency across development and testing setups and allows new contributors to replicate the simulation environment with small effort. It also serves as a basis for testing new agents, ensuring that simulations involving perception, planning, and actuation run predictably across different systems.

First, the environment itself was adapted to work properly with a Docker container, preserving the original Gazebo scenario. On the agent side, in the original work, it was used Jason-ROS interface which would include 3 configuration files that are required: one for the *actions*, one for the *scout perceptions* and another for the *rescuer perceptions*. From those files, it is possible to reuse the same topics and services for the perceptions and actions of the agent when adapting to E-MAS. As an example, we present part of the code of the *actions* configuration file, as follows:

```
[arm_motors]
method = service
name = mavros/cmd/arming
```

[4] https://github.com/iago-silvestre/search-rescue-px4

```
msg_type = CommandBool
dependencies = mavros_msgs.srv
params_name = value
params_type = bool

[setpoint_local]
method = topic
name = mavros/setpoint_position/local
msg_type = geometry_msgs/PoseStamped
params_name = pose.position.x, pose.position.y, pose.position.z
params_type = float, float, float
```

To adapt the agent to the E-MAS framework it was necessary to create an *.yaml* configuration file within the agent directory. The initial setup of the file has been stated in the listing below. In the *perceptionTopics* bracket, it is necessary to set up the topics where the agent would perceive, and in the *actions* bracket, it is necessary to set up both topics and services where the agent must act.

```
perceptionTopics:
      - topicName: /rescue_world/drop_buoy
        topicType: geographic_msgs/GeoPoseStamped
      - topicName: uav0/mavros/local_position/pose
        topicType: geometry_msgs/PoseStamped

actions:
  topicWritingActions:
      - actionName: setpoint_local
        topicName: /uav0/mavros/setpoint_position/local
        topicType: geometry_msgs/PoseStamped
        params:
          - header: [seq, stamp, frame_id]
          - pose:
            - position: [x, y, z]
            - orientation: [x, y, z, w]

  serviceRequestActions:
    - actionName: arming
      serviceName: /uav0/mavros/cmd/arming
      params:
        - value
```

3.2. Main Difficulties

Even though the E-MAS framework has performed well in different projects, it is a relatively new framework, so there are still some difficulties related to its usage.

The main difficulty so far has been dealing with different message types that were not originally supported. Commands like a boolean value “true” or “false” to arm the UAV or a command to send the coordinates that the UAV should go where the parameters

are "nested" were not originally supported. Therefore, it has been implemented in newer updates of the framework.

Another challenge to deal with is how the framework handles array messages and how complex they can be. In the original project, an external package called *Boustrophedon Planner* responsible for the UAV's path planner was used and requires a custom message type that is an extension of other standard types that uses stamps, poses, and polygon, all in agregated, which required a relatively complex array as a message because of those types. To deal with that specific customized message type, it was also necessary to adapt the framework.

3.3. Pros and Cons

Although both frameworks, Jason-ROS and E-MAS, have the purpose of integrating Jason agents with robotic systems, each have their own positive and negative values.

Both frameworks use different types of configuration files to set up the topics and services where the agents would act and perceive. However, in Jason-ROS it might be necessary to set up multiple files for different cases (one specific for one agent's *actions* and another for *perceptions*, for example) and then for different agents as well. On the other hand, with E-MAS the developer only need to set up one *.yaml* configuration file per agent, which each file would contain all the actions, and perceptions that the agent needs.

When it comes to handling different message types, the Jason-ROS seems to handle it well, "out-of-the-box". On the other hand, E-MAS needs to be updated depending on how complex a message type can be.

In the agent code itself, they are both similar since extra configurations are hardly necessary.

Finally, Jason-ROS stopped being updated while the E-MAS framework still receives constant updates. When Jason-ROS was developed, ROS 2 was starting to be developed and therefore, wasn't supported. On the other hand, E-MAS works with both ROS 1 and ROS 2 versions since it is based on a Java-ROS interface compatible with both versions [5].

3.4. Results

In terms of final results of this adaptation, Table 4 shows a comparison between the setting up of agents with Jason-ROS and Embedded-Mas (E-Mas). The most visible differences are related to the configuration files, since on Jason-ROS we have 2 files for one agent (the *action_manifest* file content is shared between both scout and rescuer agents), while on E-Mas each agent will have their own file, but it could mean that if we increase the amount of rescuers agents, it could increase the configuration files as well. In terms of lines of agent code, the few changes made were due to syntax differences such as symbols for internal actions from E-Mas that are not used on Jason-ROS. Another important difference is regarding the extra files that are necessary to be created on the E-Mas end, such as .java files. They're necessary in the current version so the actions with different data types can work properly since those files symbolize an "extension" of the default Java class used on E-Mas.

[5] `https://github.com/h2r/java_rosbridge`

Table 4. Comparison of the files between Jason-ROS and Embedded-Mas

	Framework	# of configuration files	# of lines in configuration files	# lines in agent code	Extra files?
Scout	E-Mas	1	80	80	Yes (.java)
	Jason-ROS	2	72 (Actions) + 27 (Perceptions)	80	No
Rescuer	E-Mas	2	64	159	Yes (.java)
	Jason-ROS	2	72 (Actions) + 27 (Perceptions)	159	No

It is worth noting that the extra .java files present in the adaptation, add a non-insignificant amount of lines-of-code, totaling at 281. These java files are only needed momentarily as for this PX4 example, E-MAS needed help in constructing some types of ROS messages, once these are built-in the interface these files will no longer be required.

An important note is that E-MAS allows to configure different perceptions and actions for agents of the same source code. For example, rescuer1 and rescuer2, both agents described in rescuer.asl Jason file, could be configured with different perceptions and actions through respective rescuer1.yaml and rescuer2.yaml files.

Regarding changes in agent source code, these were only necessary on some occasions due to differences in how these two interfaces process the ROS message and create or update a belief based on it, which resulted in some beliefs having different arranging of the information from the ROS Topic between the two versions.

In terms of agents functionalities, the E-Mas agents performed similarly to Jason-ROS, even though E-Mas is more robust and complex.

4. Conclusions and Future Works

So far, the adaptation of the SAR application has focused on testing if different message types work properly with the E-MAS framework and also debugging the framework's functionalities. Since the framework has been receiving constant updates, it has been easier to adapt and test. If a client is not connected or if a message type is not compatible with E-MAS, those issues can be seen on a debugger that is compatible with ROS and *Rosbridge*, for example. Another great advantage is the possibility to set up an entire agent with one *.yaml* configuration file, making it easier to scale and update the agent's perceptions and actions if needed.

In terms of simulation environment, the UAV has been capable of arming, setting up different modes, and flying to a specific coordinate without issues through the actions and perceptions that were set up on the agent code. These actions and perceptions are the most, but not all, of what is needed in terms of communication with ROS.

Even though the E-MAS framework is a work in progress, it presents good advantages to integrate Jason agents with ROS compared to Jason-ROS. Some of what's missing to adapt, in terms of communication with ROS, is how the UAV would plan its path to scout an area and the human detection itself (which is done by an external script and the detection sent to a ROS topic). These implementations and integrations will be done in future work.

References

Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and Resource-bounded Practical Reasoning. *Computational Intelligence*, 4:349–355.

Cardoso, R. C., Ferrando, A., Dennis, L. A., and Fisher, M. (2020). An interface for programming verifiable autonomous agents in ros. In Bassiliades, N., Chalkiadakis, G., and de Jonge, D., editors, *Multi-Agent Systems and Agreement Technologies*, pages 191–205, Cham. Springer International Publishing.

Dennis, L. A. and Fisher, M. (2023). *Verifiable Autonomous Systems: Using Rational Agents to Provide Assurance about Decisions Made by Machines*. Cambridge University Press.

Koubaa, A. (2017). *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Springer, 1st edition.

Menegol, M. S., Hübner, J. F., and Becker, L. B. (2018). Coordinated uav search and rescue application with jacamo. In Demazeau, Y., An, B., Bajo, J., and Fernández-Caballero, A., editors, *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection*, pages 335–338, Cham. Springer International Publishing.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. Y. (2009). ROS: An Open-source Robot Operating System. In *Proc. ICRA Workshop on Open Source Software*.

Rafael H. Bordini, J. F. H. and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley Sons Inc.

Rao, A. S. (1996). AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42–55, London. Springer-Verlag.

Silva, G. R. (2020). Active perception within bdi agents reasoning cycle with applications in mobile robots. Master's thesis, Federal University of Santa Catarina.

Silva, G. R., Becker, L. B., and Hübner, J. F. (2020). Embedded architecture composed of cognitive agents and ros for programming intelligent robots. *IFAC-PapersOnLine*, 53(2):10000–10005. 21st IFAC World Congress.

Silva, G. R., Hübner, J. F., and Becker, L. B. (2021). Active perception within bdi agents reasoning cycle. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '21, page 1218–1225, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Silvestre, I., Becker, L. B., Fisher, M., Hübner, J. F., and de Brito, M. (2025). Enhanced agent-oriented programming for robot teams. *Engineering Applications of Artificial Intelligence*, 158:111390.

Silvestre, I., de Lima, B., Dias, P. H., Buss Becker, L., Hübner, J. F., and de Brito, M. (2023). Uav swarm control and coordination using jason bdi agents on top of ros. In Mathieu, P., Dignum, F., Novais, P., and De la Prieta, F., editors, *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection*, pages 225–236, Cham. Springer Nature Switzerland.