MAES

a ROS 2-compatible simulation tool for exploration and coverage algorithms

Andreasen, Malte Z.; Holler, Philip I.; Jensen, Magnus K.; Albano, Michele

Published in:
Artificial Life and Robotics

DOI (link to publication from Publisher):
10.1007/s10015-023-00895-7

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Andreasen, M. Z., Holler, P. I., Jensen, M. K., & Albano, M. (2023). MAES: a ROS 2-compatible simulation tool for exploration and coverage algorithms. *Artificial Life and Robotics*, *28*(4), 757-770. https://doi.org/10.1007/s10015-023-00895-7

Artificial Life and Robotics (2023) 28:757–770
https://doi.org/10.1007/s10015-023-00895-7

ORIGINAL ARTICLE

MAES: a ROS 2-compatible simulation tool for exploration and coverage algorithms

Malte Z. Andreasen[1] · Philip I. Holler[1] · Magnus K. Jensen[1] · Michele Albano[1]

Received: 26 March 2023 / Accepted: 14 July 2023 / Published online: 21 September 2023

Abstract
With the aim of allowing the efficient and realistic simulation of swarm algorithms for exploration and coverage, we present the tool Multi-Agent Exploration Simulator (MAES), which is an open-source physics-based discrete step multi-robot simulator. MAES features movement in a continuous 2D space, realistic physics based on the Unity framework, advanced visualization techniques such as heatmaps, custom wireless signal degradation, both randomly generated and custom user-provided maps, and a ROS (Robot Operating System) interface. This latter characteristic could allow to port the simulated algorithms to real-world robots. We present performance tests, conducted with rather modest hardware, showing that MAES is able to simulate up to 5 robots in ROSMode (using the ROS integration) and up to 120 robots in UnityMode (development performed directly into the C# Unity Editor). A usability test was conducted which hinted that the target audience of robotics researchers and developers is able to quickly install, setup, and use MAES for implementing simple robot logic.

Keywords Robot operating system · Continuous space · Autonomous robots · Swarm robotics · Multi-agent · Nav2

1 Introduction

Testing implementations of swarm robotics can be an expensive and difficult task. For this reason, many developers look to simulations as they offer a cheaper and easier solution for testing an algorithm or swarm behavior. Popular simulation solutions include Argos [1] and Gazebo [2]. Even simulations, however, can be difficult to set up and configure. This difficulty is usually caused by the simulation tools having a heavy emphasis on modularity and customizability, which often comes at the cost of increased complexity for the user. Additionally, some simulation software has heavy requirements in terms of CPU power, which can necessitate using high-end computers or computing clusters, and limit the size of the swarm to be simulated.

One significant domain for swarm robotics is online collective terrain exploration and coverage, where *online* refers to the terrain being initially unknown and, thus, explored while the swarm algorithm is executed [3]. Practical application of terrain exploration spans from search & rescue to surveillance, while terrain coverage algorithms can be used, for example, for floor cleaning and smart farming [4, 5]. This set of problems has arguably a strong need for simulation tools, especially for online algorithms, since the latter must be tested against a large number of plausible terrain configurations to evaluate if an algorithm behaves as required. These requirements raise the need for a simulation platform that is not only realistic, but also computationally efficient.

This paper presents the simulation tool Multi-Agent Exploration Simulator (MAES). Previous work [6] has introduced a preliminary version of MAES, which was a simple implementation of a realistic simulation framework (movement space represented as a continuous 2D plane rather than a grid of cells—see Fig. 1—and hardware capabilities of robots from different state-of-the-art algorithms mapped onto a common ground) implemented on top of the Unity physics engine [7]. However, the tool was oriented at computer scientists, and in fact it required to develop algorithms

✉ Michele Albano
mialb@cs.aau.dk

Malte Z. Andreasen
malte@mza.dk

Philip I. Holler
philipholler94@gmail.com

Magnus K. Jensen
magnjensen@gmail.com

[1] Department of Computer Science, Aalborg Universitet, Selma Lagerlöf Vej 300, 9220 Aalborg, Denmark

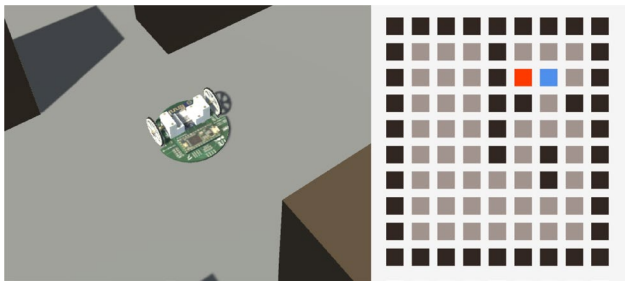


Fig. 1 Left: An agent moving in continuous space in MAES. Right: a traditional grid-based simulation, with the red and blue dots representing agents and the black dots representing walls

directly in the C# programming language in the Unity Editor, it was not compatible with the workflow of many robotics researchers and developers, and the algorithms developed were not easily ported to real robots since the simulator interface had very little overlap with existing approaches used in robotics.

This paper focuses on a new iteration (namely, 2.0) of MAES[1] that introduces a number of technical improvements as well as a new interface for controlling the robots using Robot Operating System version 2 (ROS[2]) [8], which allows robotics researchers and developers to use existing libraries and solutions when developing control algorithms in MAES, and it eases adapting the developed controllers to real world robots.

MAES allows for visual inspection for easier debugging and comparison of implemented algorithms. MAES has two modes: *UnityMode* with support for C# development directly into the C# Unity Editor, and *ROSMode* which enables the ROS integration. The main contributions of this paper include:

- With the aim of a self-contained work, a discussion of ROS 2, as well as a quick summary of existing simulation platforms (Sect. 2);
- The description of the MAES tool (Sect. 3);
- A detailed discussion on how the ROS interface was implemented into the MAES architecture (Sect. 4);
- Results of performance and usability tests, to ensure that MAES has better performance than existing systems and still is a good fit with the competencies of robotics researchers and students (Sect. 5);
- Proposals for future work to apply MAES to new use cases and scenarios (Sect. 6).

2 Background information

2.1 Simulation of exploration and coverage algorithms

Realistic simulation of swarm exploration and coverage algorithms is a relatively unexplored research area. In fact, even though many works introduce novel algorithms, we argue that most simulation settings are far from realistic. In fact, existing works consider one or more of the following: (i) the environment as grid-like structure that can be maneuvered by moving from cell to cell [9, 10] where (ii) the robots cannot suffer from colliding with each other; (iii) communication between robots has unlimited range and is not blocked by walls [11, 12]; (iv) robots know always each other's location [13], and are provided with real-time distributed Simultaneous Localization And Mapping (SLAM) [14].

A few works aimed at providing a common—and more realistic—framework to compare existing algorithms. A comparison of coverage performance for existing algorithms with reduced communication range can be found in [15]. Moreover, the authors of [16] extend the comparison by implementing the algorithms using hardware FireBird V robots on a small plane. However, contrary to MAES, both works consider movement in a grid-based environment (see Fig. 1).

2.2 ROS and robotics simulators

Multiple simulation tools exists for robotics, such as Argos [1], Gazebo [2], Player/Stage [17], and NVIDIA's Isaac [18]. All of the aforementioned simulators are general purpose, and include some kind of ROS support, for either ROS 1 or ROS 2 interface [19].

As of writing, the most popular simulator for use with ROS appears to be Gazebo, as it is a first party simulator, developed by the Open Robotics Foundation, which ROS also belongs to. Gazebo is highly modular and customizable. For example, the user can specify the physical structure of the robots through Unified Robot Description Format files. This type of flexibility allows it to support most use cases but comes at the cost of increased complexity, a cumbersome setup process, and high computational overhead.

The work in [20] describes a recent simulator based on Unity. The scenario targeted by the simulator is focused on the Jackal unmanned ground vehicle from ClearPath Robotics, as the robot navigates autonomously through differing environments. The simulator focuses on a single

[1] A video demonstration of MAES can be found at https://youtu.be/lgUNrTfJW5g. MAES is open source and the source code can be found at https://github.com/DEIS-Tools/MAES.

[2] Unless otherwise specified, in this paper, the term ROS refers to ROS version 2

robot, on navigation only, and it is hardly applicable to swarm algorithms.

Given their high level of modularity and general purpose, the aforementioned simulation tools present a steep learning curve and high computational load. On the other hand, our proposed MAES is intended as an alternative simulator to be used only in a limited number of use cases, namely multi-robot exploration and coverage. Thus, MAES can focus on a higher level of usability and performance, as it will be discussed in this paper.

2.3 ROS definitions

ROS is a set of libraries and tools for building robotics systems [8]. ROS is developed and maintained by Open Robotics [21]. One of the main ideas behind ROS is that developed software can be used both with real-world robots and in simulated environments.

Throughout this article, we will use several ROS-specific terms. As no formal definitions are provided in the ROS 2 documentation, the following definitions are summaries of the explanations from the ROS 2 documentation. [8] A node is a fundamental ROS element that serves a single, modular purpose in a robotics system. Nodes communicate via either Services, which implement a request-and-response communication method, or by publishing messages over topics, which allow any number of other nodes to subscribe to and access the messages' content.

Message formats and executable code are defined inside ROS workspaces, which are locations in the file system that contains a number of ROS related development files. A launch file is a file with a definition of how to execute one or more nodes with a specific configuration. A launch file can be called from the ROS 2 Command Line Interface (CLI). Actions are a communication type in ROS intended for long running tasks, and consist of three parts: a goal, feedback, and a result. A costmap is a grid-like structure of cells (an occupancy grid), representing the surrounding environment in terms of the cost of moving from the current location to the location represented by the cell. A low or zero value means a cell is free and a high value means a cell is occupied.

3 The MAES tool

This section describes the MAES simulation tool. MAES is a 2D discrete time-step continuous-space physics-based simulation, visualized in 3D. The simulation progresses in steps of 0.01 seconds, and each agent receives a state update and executes its own algorithms every *tick*, i.e., 0.1 seconds, which is, thus, the reaction time of an agent. The simulator uses the Unity Engine [7] for visualization and physics simulation. The simulated robots are mapped onto a common hardware model, which allows to add or remove hardware capabilities to provide insights into the benefit of integrating specialized hardware into the robots.

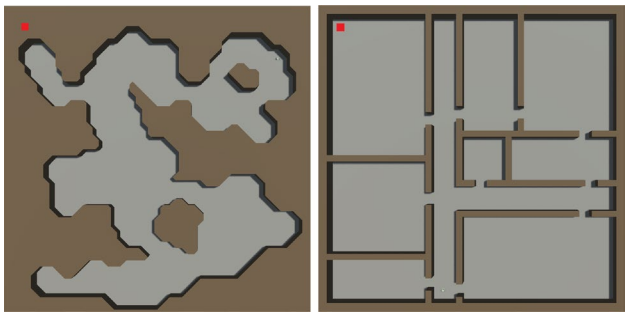


Fig. 2 Example of generated cave and building maps of 50x50 tiles. The red squares on the top left show the size of a tile

MAES features a map generator for dynamically generating cave type and building type maps Sect. 3.1 given a random seed and some parameters. MAES allows for a standardized interface for the interaction between a controller and a robot Sect. 3.2, a realistic physical model for agents' movement Sect. 3.2.1 , sensing and communication via either broadcast wireless communication or environment tagging Sect. 3.2.2, simulated SLAM Sect. 3.2.3, and advanced visualization features Sect. 3.3. MAES allows many of its features to be configured, and it takes parameters for agent constraints, for physics simulation, and for map generation. A list of all possible parameters for the simulator can be seen on the MAES public code repository.

3.1 Environment maps

MAES provides automated map generation for building and cave-map types with configurable parameters, see Fig. 2. Moreover, MAES allows to specify a custom map by providing an image in the 'Portable Gray Map' format (.pgm). PGM is also the format exported by the Nav2 Map Saver node, meaning that ROS generated SLAM maps can be imported into MAES. When provided with an image, the MAES map generator creates wall tiles for every completely black pixel and open tiles for every other pixel. Figure 3 shows an example of an image that has been converted into a map.

3.2 Agents

Within MAES, an agent's capabilities can vary depending on the parameters that describe the simulation at hand, e.g., limited vision, broadcasting range, etc. Figure 4 shows the 3D model of an agent, inspired by the MONA robots [22].

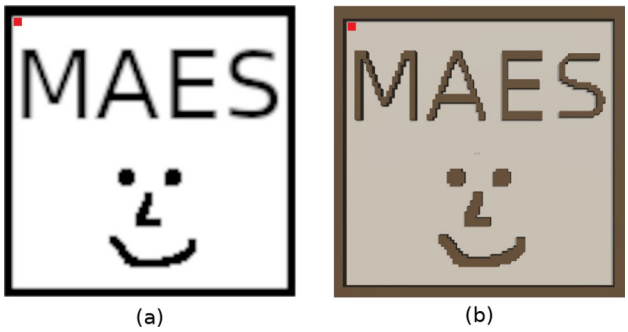


Fig. 3 Example of a custom map. MAES converts the input image of 50x50 pixels (**a**) into a map of 50x50 tiles (**b**) with walls for each black pixels. The red squares on the top left show the size of a pixel/tile

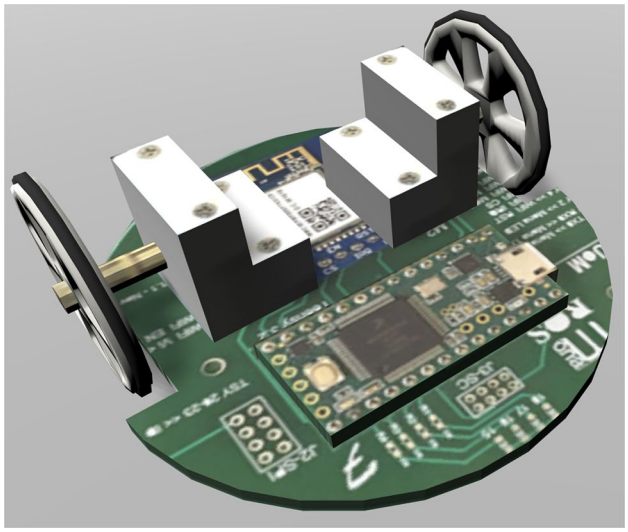


Fig. 4 The 3D model of the MONA-inspired agent used in MAES

MAES is intended to support the implementation of many different algorithms. For this reason we expose interfaces for the algorithms to control the agents and provide access to sensor information. In particular, we expose a native interface to be used in UnityMode, and a ROS interface, further explained in Sect. 4, for the ROSMode.

3.2.1 Movement

An agent is able to rotate in place and move straight ahead, and the agent is not able to rotate while moving forwards. Movement is simulated through the Unity 2D Physics Engine by applying force at the position of each wheel. This simulation accounts for inertia, drag, and collisions with obstacles and other agents. The agents of the MAES simulator can reach a top speed of 3 tiles/second (or 0.3 tiles/tick). The tile size can vary depending on the scale of the map. Drag is a function of speed, which in combination with inertia results in non-constant acceleration, leads to an agent reaching half its top speed (1.5 tiles/second) after 4 ticks.

3.2.2 Sensors and communication

Agents are able to sense other agents at a given distance, specified by a simulation parameter. To accommodate a variety of scenarios with differing hardware capabilities, the capability of sensing other agents can be configured to be blocked by walls, i.e., line of sight requirement. Agents detect collisions with walls and with other agents, and can detect a nearby wall and the angle to said wall. This could, for example, be achieved in the real world using a Light Detection and Ranging (LiDAR) scanner.

Agents can communicate through broadcasting, for example by means of the `/maes_broadcast` topic when in ROSMode, and both communication range and the capability to pass walls is defined via simulation parameters. Line of sight is determined using ray tracing, and user can either provide a maximum communication range or a custom function, which is used to calculate communication success. This latter capability is available at initialization time (execution of the `SimulationScenario` function), by passing it a custom `calculateSignalTransmission` function, which takes as arguments the total distance traveled for the signal, and the distance traveled through solid walls, and must return a Boolean value indicating whether the signal can be received. The advantage of this approach is that the user can control the model used for signal degradation. If a stochastic model is desired, the user can factor random number generation into the signal success function.

Finally, agents can drop tags on the ground to deposit information in the environment and communicate indirectly with other agents, for example by means of the `/maes_deposit_tag` topic when in ROSMode, as required for example by the Local Voronoi Decomposition algorithm [23]. Tags can only be dropped at an agent's current position, but data can be written to and read from at a configurable distance.

3.2.3 Simulated SLAM

SLAM is simulated by performing a series of ray traces from the position of the agent, and measuring the distance that the rays traveled before a collision. This emulates the behavior of technologies such as LiDAR scanners.

The agent continuously constructs a "SLAM map" using the ray tracing information when it becomes available. If any object is detected within the region of a tile, then that entire tile is marked as solid. If a ray trace is sent in the direction of a tile, and no object is found, the tile is assumed to be

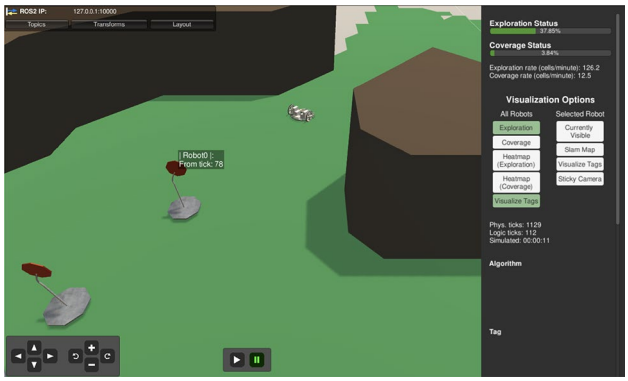


Fig. 5 A screenshot from MAES with an environment tag visualized with the hover menu containing sender and content of the tag

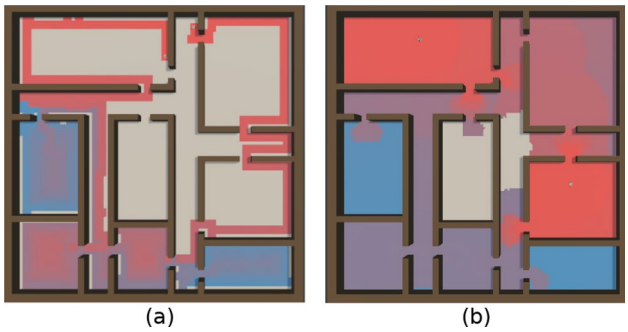


Fig. 6 **a** Shows the heatmap visualization for coverage and **b** shows the heatmap for exploration. Red areas have been explored/covered recently, blue areas earlier, beige areas have not yet been explored/covered at all

open, unless previous traces indicate that it is solid. The agent has access to an approximation of its location within the SLAM map. The simulation can be configured to automatically synchronize SLAM maps of agents that are within communication range of each other.

3.3 Visualization and debugging features

As MAES should function as a testbed for many diverse algorithms featuring a potentially large number of agents, such as most swarm algorithms, we include a wide variety of visualization and debugging tools.

The user interface of MAES (see Fig. 5, or the full sized image on the Github repository of MAES [24]) includes several panels for controlling the simulation. The user interface adapts to whether it is run in ROSMode or UnityMode. For example, the visualization of the ROS connection in the top left corner is not present when running in UnityMode. The fast forward buttons are hidden in ROSMode, due to timing issues in the nodes when fast forwarding with ROS—some Nav2 nodes have some assumptions about the robot speed, which can be violated if MAES is sped up.

A menu is included for controlling the camera view over the simulation, as well as changing the simulation speed. Additionally, agents can be individually selected, which makes the camera follow the agent as well as reveal debugging information in a side bar regarding the selected agent.

When a simulation is running, the surface of the map is highlighted in green if any agent at any point has explored it. If an agent is selected, the surface reveals in blue the tiles included in the SLAM map for said agent. The SLAM map can also include sections of the map revealed by other agents, if SLAM synchronization is enabled and the two agents have been within communication distance of each other. Environment tagging is visualized using colored boxes on the ground where an agent has tagged the environment. It is possible to hover or click on a tag to inspect the information contained in it.

To allow better analysis of algorithm behavior, MAES features two heatmap visualization modes (see Fig. 6), one for the exploration measure and one for coverage. This could, for example, be useful for testing patrolling algorithms. The heatmaps display a color at each tile of the map, indicating how recently it has been explored/covered by an agent. Tiles that have been explored/covered recently have a red tint. A tile progressively changes to a blue tint as time passes without a robot exploring/covering the tile.

4 Integration of ROS interface within MAES

This section describes how ROS is integrated into MAES. Section 4.1 discusses which version of ROS is used and why. Section 4.2, 4.3 and 4.4 describe in detail the architecture and communication of MAES with ROS. Finally, Sect. 4.5 discusses how controllers must be implemented and provide an example.

4.1 ROS1 or ROS2?

As of writing, ROS exists in two major versions, ROS1 and ROS2, where ROS2 is a complete remake created based on observations and lessons learned from ROS1. The latest release for ROS1 is ROS Noetic Ninjemys, which came out in May 2020 and has end-of-life in May 2025. No further releases for ROS1 are planned. The first version of ROS2 was released as alpha1 in August 2015 and the first official release came out in 2017. The currently newest version of ROS2 is Galactic Geochelone, which came out in May 2021 [25]. Galactic is, however, not a long-term support (LTS) version. The first LTS version of ROS2, Humble Hawksbill, came out on the 23rd of May 2022 and will have

support until May 2027. As of the time of writing, Humble Hawksbill does not yet support some of the navigation functionalities, thus the descriptions in this document refer to the non-LTS ROS2 (Galactic Geochelone) release.

The downside of using ROS2 is that a large amount of existing libraries are written for ROS1 and many of them do not have a version compatible with ROS2. However, due to the end-of-life of ROS1 in 2025 and the continued support for ROS2, we chose to target ROS2 for the MAES simulator.

4.2 Architecture

MAES is intended to be used as an importable library, and it exposes an interface that allows full configuration without needing to manipulate MAES code. This interface exposes methods for simulator instantiation, injection of algorithms and scenarios, and allows for extraction of performance metrics. MAES also contains a Unity Package definition, which allows it to be used in the official Unity Package Manager tool by pasting the Github repository URL into its user interface.

Code snippet 1 shows an example of how a simulation can be set up using the MAES framework. This code can be attached to an empty Unity `GameObject` (the base-objects used in the Unity Editor) so that the simulator gets instantiated and run.

Listing 1: Example of MAES usage in a unity project

```
void Start(){
    // Get/instantiate simulation
    var simulator = Maes.Simulator.GetInstance();

    // Configure the scenario
    var caveConfig = new CaveMapConfig(123,
    ↪  widthInTiles: 75, heightInTiles: 75);
    var scenario = new SimulationScenario(123,
    ↪  mapSpawner: generator =>
    ↪  generator.GenerateCaveMap(caveConfig));
    simulator.EnqueueScenario(scenario);

    simulator.StartSimulation();
}
```

One of the goals of MAES is to allow for easier development and less setup and configuration efforts compared to using ROS with other simulators. This is achieved partly by having a single configuration file `maes_config.yaml`, where all customization is done. As an alternative, it is possible to feed the configuration parameters programmatically to the configuration classes (i.e., `SimulationScenario`, `CaveMapConfig`, `RobotConstraints`, and `BuildingMapConfig`). All the parameters are described in the documentation in the public repository. [24]

The size and complexity of the MAES architecture increase significantly when in ROSMode with respect to UnityMode, due to the large amount of ROS nodes running alongside MAES. When using MAES in ROSMode, the user must first launch the ROS components, and then subsequently launch MAES. All ROS nodes for all robots are launched from a single custom ROS Launch file called `maes_ros2_multi_robot_launch.py`. An abstract overview of the ROS nodes launched from this launch file can be seen in Fig. 7. The overview is abstract in the sense, that the internal sub nodes created by the Nav2 and Slam_ toolbox packages are excluded from the graph for the sake of readability. A full picture containing all nodes and topics can be seen on the public code repository. [24]

Unity, on which MAES is based, already supports both ROS 1 and ROS 2 communication through the `ROS TCP Connector` and `ROS TCP Endpoint` plugins. These plugins work in tandem to facilitate communication between Unity and an external ROS system. The `TCP Endpoint` is a ROS node that relays communication from Unity to the ROS system and vice versa. The `TCP Connector` is responsible for establishing a connection between Unity and the `TCP Endpoint` node. In our configuration, the `maes_ros2_multi_robot_launch.py` script initially launches the `default_server_endpoint` node from the `ROS TCP Endpoint` script. This node is not namespaced,[3], and a single `default_server_endpoint` node is launched for all robots. After launching that node, `maes_ros2_multi_robot_launch.py` starts launching namespaced nodes.

To know how many robots, i.e., namespaces, to create, `maes_ros2_multi_robot_launch.py` reads from the same configuration file as MAES, which ensures that they are synchronized. In addition to the number of robots, `maes_ros2_multi_robot_launch.py` reads parameters such as the raytrace range from the configuration file and injects it into the parameter file of each robot. This approach allows for a single parameter file to be modified and used for all robots, which enables easier development and debugging. A disadvantage of this approach is that all robots have the exact same configuration, which can reduce realism and flexibility.

After successfully reading and injecting the parameters, `maes_ros2_multi_robot_launch.py` launches for each robot a `maes_robot_controller` node, which

[3] Namepace refers to a prefix, e.g., `/robot0/` which is prepended to all topic names used by the node, as well as the node name itself, to allow to distinguish between the robots.

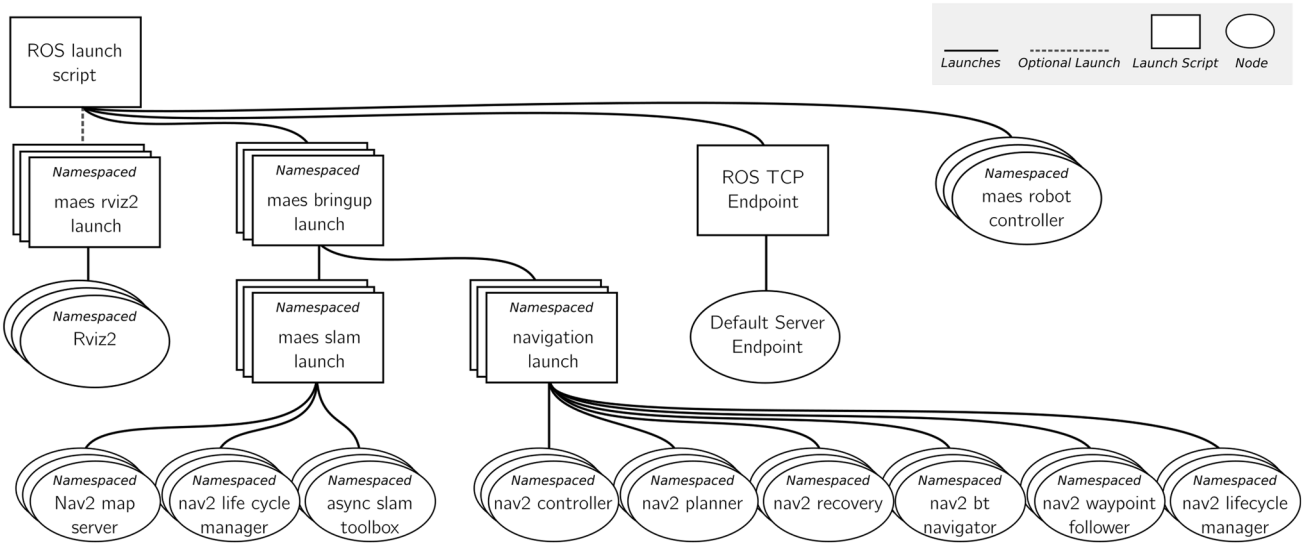


Fig. 7 Visualization of how nodes are launched from our ROS launch script

is the node containing the robot logic. Additionally, a namespaced launch file called `maes_bringup_launch` is called for each robot, which further launches the nodes slam_toolbox and Nav2 inside the given namespace, used for navigation and mapping purposes respectively.

4.3 Publishing robot state through ROS

Many nodes in the MAES ROS workspace depend on input from MAES to function. To supply the needed input MAES publishes data to the topics `/tf`, `/scan` and `/maes_state`. The `/tf` topic is used for the transforms, i.e., the position and rotation of a given robot including the relative positions of sub-components of the robot. As of writing, the transforms published by MAES do not contain any inaccuracies, which reduces realism, as a position derived through odometry would likely be imprecise due to sensor inaccuracies. However, in the future this could be adapted to use the positional inaccuracy variable that already exists for SLAM when MAES is executed in UnityMode [6].

Once per tick (10 times each second), data for the `/scan` topic are created and sent by performing a series of ray casts from the current position of the robot. Every message consists of distance information for 180 ray casts performed at a 2-degree interval around the robot.

The `/maes_state` topic uses a custom ROS message called `StateMsg` that contains information regarding the current simulation tick, the current status of the given robot (e.g., rotating or moving), whether the robot is colliding with something, incoming broadcast messages, environment tags nearby as well as information about other nearby robots. The information received through this topic can then be used in the logic controller for the robot, in a manner similar to what is provided to the UnityMode algorithms.

As mentioned in Sect. 4.2, we use the ROS TCP Connector package from Unity in order to publish the messages to the ROS topics [26]. The ROS TCP Connector allows for generating serializable C# classes from message definitions found in a ROS workspace. The generated classes can then be instantiated as objects in the C# code, populated with values, serialized, and then published to the corresponding topic. The messages are sent through the ROS TCP Connector to the ROS TCP Endpoint, [27] which relays all messages as if they came from a regular ROS node. A representation of this approach can be seen in Fig. 8.

4.4 Mapping Nav2 commands to MAES

Nav2 is the ROS package used for navigation [28]. Nav2 contains several nodes that depend on the `/tf` topic for information regarding current position. Additionally, Nav2 constructs costmaps from the `/scan` topic used for navigation.

Moreover, Slam_toolbox [29] uses the `/scan` topic for creating the map, and the `/tf` topic for positioning the robot within the map. Slam_toolbox, however, differs from Nav2 by having a higher resolution and the ability to export the map for later use.

For MAES, we initially wanted to be able to merge maps for both the slam_toolbox as well as the Nav2 costmap, to mirror the map merging feature that is available

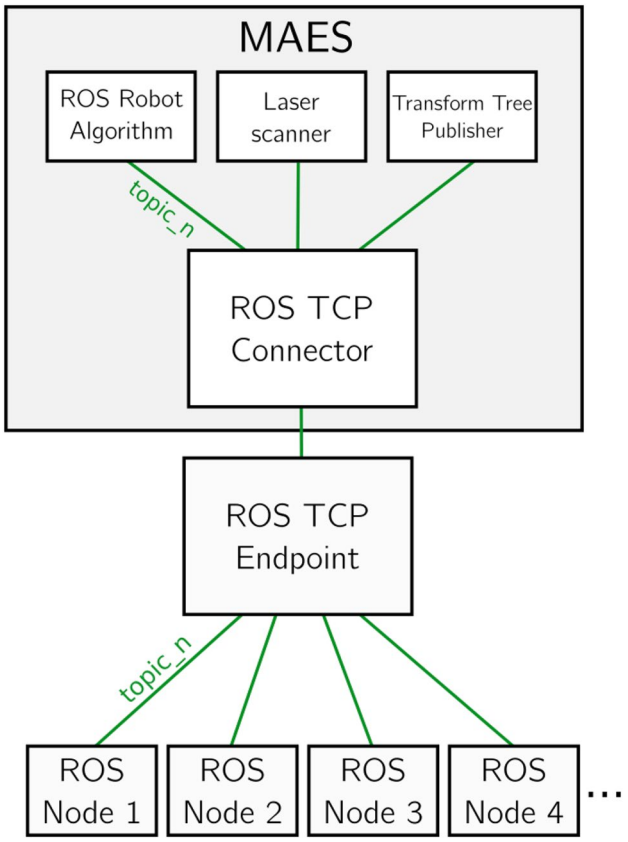


Fig. 8 A visualization of how MAES publishes to ROS nodes on ROS topics

to UnityMode algorithms and allow cooperative navigation in Nav2. This is, however, not possible using the official slam_toolbox and nav2 packages, since this capability has not yet been ported officially to ROS 2.

Whenever a robot is instructed to navigate to some point using Nav2, the Nav2 nodes publish movement instructions on the `cmd_vel` topic. These instructions consist of a desired force application for each wheel of the robot. The standard Nav2 configuration assumes that differential steering is possible, i.e., that the robot can rotate while moving.

4.5 Controlling the robot in MAES

As discussed in Sect. 4.2, when executed in ROSMode, MAES launches a number of robot specific nodes, such as `maes_robot_controller`, which allows for control functionality. When the node is initialized, all of the subscriptions and clients for the topics, services, and action servers are created in the given namespace, including callback functions. The services and action servers include functionality for navigation, broadcasting messages, and depositing environment tags. The user can use this information to implement the robots' logic. The logic controlling the behavior of the robot is written in the python programming language in the main function of the `maes_robot_controller.py` file, and can make use of the services, information received on topics, and action servers.

In MAES UnityMode, the code must be developed using C#, and the robot controller must implement the `IRobotAlgorithm` interface. This interface contains an update function called every logic tick of the simulation as well as a start method for bootstrapping. The algorithms are then compiled alongside the simulator itself. To avoid the computational overhead introduced by the ROS TCP Endpoint and TCP Connector plugins, the global settings configuration contain a Boolean value that toggles these plugins, such that they no dot consume resources when they are actually not needed.

When developing the algorithms, it must be taken into account that unlike the C# algorithm implementations, the `Ros2Algorithm` is nondeterministic because it is dependent on the timing of network messages, while the simulator is deterministic when in UnityMode.

Code Snippet 2 shows a simple frontier algorithm, which exemplifies the usage of the interfaces for logging, getting the state of the robot, and using the costmap for navigation exposed by `maes_robot_controller`.

Additionally, `maes_robot_controller` exposes methods for using the asynchronous navigation server such as `nav_to_pos`. `nav_to_pos` makes the robot navigate to an (x,y) coordinate asynchronously. Since actions servers in ROS can be canceled, we allow for checking the status of the current goal with a `goal_handle`, and we also provide utility functions such as `cancel_nav`, which simply cancels the current goal using the navigation action client.

Listing 2: Example of a Frontier Algorithm implemented in the `maes_robot_controller` in Python

```python
def main(args=None):
    rclpy.init(args=args)
    robot = RobotController()
    robot.wait_for_maes_to_start_simulation()
    next_goal: Coord2D = None
    next_goal_costmap_index: int = None

    # Returns true if the tile is not itself unknown, but has 2 neighbors, that are unknown
    def is_frontier(map_index: int, costmap: MaesCostmap):
        # -1 = unknown, 0 = certain to be open, 100 = certain to be obstacle
        # It is itself unknown
        if costmap.costmap.data[map_index] == -1: return False
        # It is itself a wall
        if costmap.costmap.data[map_index] >= 65: return False
        return costmap.has_at_least_n_unknown_neighbors(index=map_index, n=2)

    while rclpy.ok():
        rclpy.spin_once(robot) # Allow callback execution
        # If no goal found or current nav complete
        if next_goal is None or robot.is_nav_complete():
            # Find index of first tile in costmap that is a frontier
            goal_index = next((index for index, value in enumerate(robot.global_costmap.costmap.data) if
            ↪  is_frontier(index, robot.global_costmap)), None)
            # No more frontiers found, just continue
            if goal_index is None:
                robot.logger.log_info("Robot {0} has no more
                ↪  frontiers".format(robot._topic_namespace_prefix))
                continue

            next_goal = robot.global_costmap.costmap_index_to_pos(goal_index)
            next_goal_costmap_index = goal_index
            # Deposit tag when a new target/goal is found
            robot.deposit_tag("From tick {0}".format(robot.state.tick))
            robot.nav_to_pos(next_goal.x, next_goal.y)
        # If goal present but not yet reached
        elif is_frontier(next_goal_costmap_index, robot.global_costmap):
            # Logging feedback from action server
            continue
        # If goal is explored
        else:
            next_goal_costmap_index = None
            next_goal = None
            robot.cancel_nav()
```

5 Performance and usability tests

As mentioned in Sect. 1, existing simulating tools have heavy requirements in terms of computing power, even when simulating just a single robot. As robots in real life are more and more likely to act as a part of a swarm rather than as a single isolated agent, the simulating tools must be able to simulate more than one robot in the same environment.

To investigate on the feasibility of using MAES for swarm algorithms, we orchestrated a performance test, see Sect. 5.1, which gives insights on the current state of

performance, and can act as a baseline for benchmarking future performance upgrades to MAES.

The ROS MAES integration was made in cooperation with the Robotics Lab at AAU, which provided us with continuous feedback during development. Regardless, we designed and performed usability tests (see Sect. 5.2) to confirm the usability of MAES in ROSMode for the target audience, i.e., robotics researchers and students.

5.1 Performance test

The tests were conducted on a thin and light laptop with an AMD 4500U CPU (6 cores @2.3GHz) and 8 GB of DDR4 memory running Ubuntu 20.04 LTS as its operating system. A logging-script was constructed which enabled us to log the CPU and memory utilization, with data points being logged once per second. Each log entry would be labeled with a timestamp, making alignment of the data easier. While the logging-script was running, it was possible to type in events in the terminal, which would also be labeled with a timestamp, making it easier to determine which event triggered specific data point.

Tests were run in both ROSMode and UnityMode, with the goal of evaluating the performance of the simulator when supporting a given number of robots as well as different map sizes. The robots were running the simple frontier algorithm example found in the MAES ROS workspace.

During each test, a number of events were recorded. Each event is represented in the figures by a green vertical line with a label. The list of events is as follows: "ROS start" and "ROS start with RViz" for ROSMode MAES only, to specify at which point in time the ROS subsystem was started; "MAES start" and "stop" to specify when MAES was actually started and when the simulation ended; "set to fastest speed" to indicate that UnityMode MAES was set to maximum speed.

5.1.1 Map sizes

Both ROSMode and UnityMode were tested with map sizes 30x30, 40x40, and 50x50. For UnityMode MAES, the map size did not notably impact the memory usage or CPU utilization. On the other hand, ROSMode MAES appeared to be quite sensible with regards to the map size, especially when enabling the RViz plugin, as seen in Figs. 9b and 9d.

5.1.2 Number of robots

For each map size considered in Sect. 5.1.1, both ROSMode and UnityMode were tested with one, three, and five robots. The different number of robots produced no significant performance impacts in UnityMode. In ROSMode, however, resource utilization suffered an increment.

Figure 9a–c shows a run on the same 50x50 cave-map, with their respective number of robots. The results show that the CPU power required by MAES in ROSMode grows with the number of simulated robots, and this is the only resource reaching its limit during the run with five robots on a 50x50 size map with RViz enabled (i.e., the heaviest configuration identified with the tests for ROSMode MAES). If RViz is not enabled, the tests show that running with up to five robots is possible on the computer system used in the test. If RViz is needed, the test system was unable to run more than three robots without suffering from the CPU limitations.

These limits are, of course, highly dependent on the characteristics of the computer running MAES in ROSMode, and we purposely chose not to run these tests on a high-performance computer to cement the fact that MAES is a relatively lightweight simulator compared to other ROS compatible simulators. Figure 9e further backs up the claim of MAES being lightweight. Here, MAES is run in UnityMode with the same number of robots, with the same map size as the test in Fig. 9c. The resource demands were significantly lower, demonstrating the overhead of running the ROS interface.

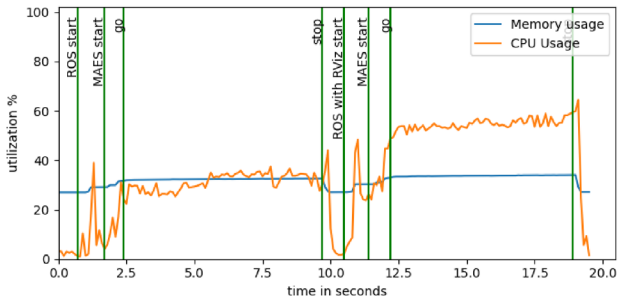
As a final set of tests, we also tried pushing the limits of MAES in UnityMode with respect to the number of robots. Figure 9f shows that MAES is able to perform a real-time simulation of 120 robots in a cave-map of size 75x75. In the experiment, we noticed that the frame rate was reduced to 4–5 frames per second when trying to simulate these many robots, but the underlying physics simulation was not affected. As Fig. 9f also shows, the computer still had a surplus of CPU resources, but the increased number of robots still had an impact on the computational resources. This result hints at MAES not being able to fully utilize multiple CPU cores, which is an optimization that is left as future work.

5.2 Usability test

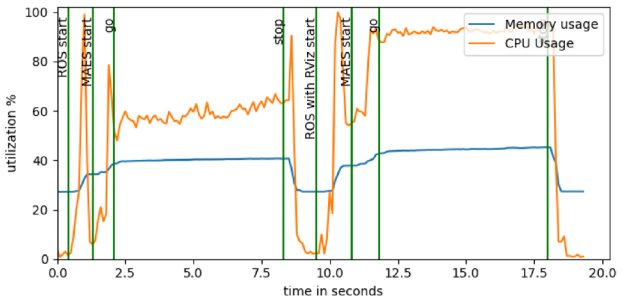
The goal of the usability test is to test whether or not the target audience can setup and use MAES in ROSMode only using the guides and documentation from the readme included in the GitHub repository. Since the primary function of MAES is to simulate multi-robot behavior, we determined the target audience to be robotics developers and researchers. Given their background, we made some assumptions about their domain knowledge. For example, we assumed that they have at least some knowledge of ROS.

5.2.1 Setup of the usability test

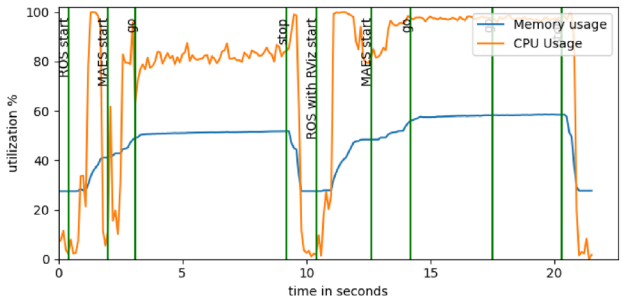
The test consisted of a short interview, followed by three phases of tasks, and then finally a small closing interview



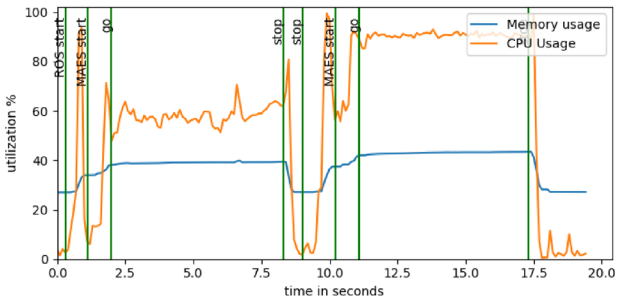
(a) Performance metrics of one robot in a 50x50 cave-type map in ROSMode



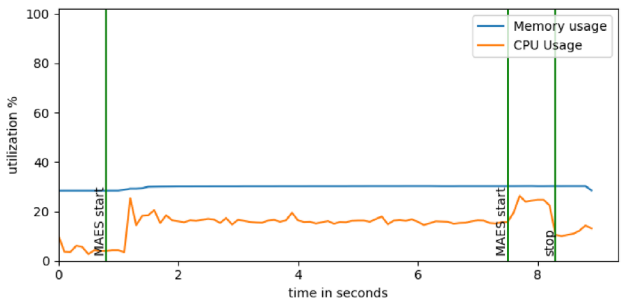
(b) Performance metrics of three robots in a 50x50 cave-type map in ROSMode



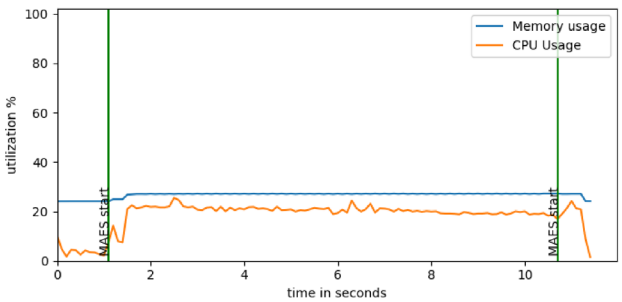
(c) Performance metrics of five robots in a 50x50 cave-type map in ROSMode



(d) Performance metrics of three robots in a 30x30 cave-type map in ROSMode



(e) Performance metrics of five robots in a 50x50 cave-type map in UnityMode



(f) Performance metrics of 120 robots in a 75x75 cave-type map in UnityMode

Fig. 9 Performance experimental results

with room for open discussions. We had an interviewer guidelines to ensure a similar experience for all subjects.

The initial interview, driven by a questionnaire, is meant to get an understanding of the subject's current knowledge of ROS, MAES, Gazebo, and Python. This latter aspect was particularly important, since some of the tasks require some knowledge of Python.

The tasks are divided into the *Setup Phase*, the *Configure Phase*, and the *Use Phase*, and in particular there were 3 tasks each for the first and second phases, and 4 tasks for the Use Phase. We considered that a subject could struggle with MAES setup and still have a good experience using it, once it is set up and configured. For this reason, we included a limit of 20 min for each phase, after which the interviewer helps the subject to finish the current task and moves on to the next phase.

The Setup Phase included tasks for setting up a Docker subsystem and running MAES in its default configuration. The Configuration Phase revolved around changing the system parameters in MAES. The Use Phase included tasks that alter the maes_robot_controller python script to control the behavior of the robots, as well as tasks for using the ROS services exposed by MAES, such as depositing environment tags.

The closing interview is meant as an open discussion. Both the initial and the closing interviews were driven by questionnaires, more information can be seen in [30]

5.2.2 Results of usability test

The usability test was completed by five subjects, which we will refer to as S1 through S5. The subjects are anonymized except for their title and experience using ROS, Python, and Gazebo.

The Setup Phase involved building and starting ROS with RViz running in a Docker container (Task 1), starting MAES from the precompiled MAES package (Task 2) and stopping ROS and closing MAES (Task 3). The Configure Phase included changing the number of robots to 1 using the MAES config file (Task 1), changing map type from building to cave using the MAES config file (Task 2), and launching first ROS and then MAES and confirm new number of robots and map type from the user interface (Task 3). Finally, the Use Phase comprised changing programmatically the behavior of a robot to make it move to coordinate (0, 0) (Task 1), make the robot cancel its movement (Task 2), make the robot deposit an environment tag (Task 3) and use the user interface to access the `maes_state` topic of the robot.

Figure 10 shows the time spent on each task for all subjects. The subjects spent time in the range of 22–37 min completing all the tasks. Phase 1 Task 1 (P1T1) includes building the Docker image and the ROS workspace, and for this reason, a lot of the time spent was waiting for the computer to complete the build. This is probably also the reason why, the subjects were so close in terms of time spent completing P1T1. All tasks in Phase 1 were completed by all subjects. The target audience thus appears to be able to set up MAES in ROSMode.

In Phase 2, four out of five subjects finished all tasks. S3, however, accidentally changed the map type to custom map and not cave-map, and thus did not finish P2T2. Regarding P2T3, S1 tried to build the ROS workspace from the wrong directory once, but managed to build it correctly later. Configuration of MAES in ROSMode, i.e., Phase 2, appears to be possible for the target audience.

In Phase 3, i.e., the Use Phase, the subjects had some difficulties. P3T1 is to implement some very simple logic into the controller of the robot. S1 finished this task without difficulties. S2, S3 and S4 all tried to put it into the main function of the robot controller Python script. This made sense to the subjects, since many ROS nodes are controlled this way. For the version of MAES used for testing, however, the logic of the robot had to be written in the logic_loop_ callback function, which is continuously called whenever the controller receives a state update from MAES, which happens every 0.1 s with default settings. (Note: This has since been changed as a result of this usability test.) Subject S2, S3, S4 eventually did either read the readme describing this behavior or found the instructions, but it took significantly longer to complete P3T1 in this way.

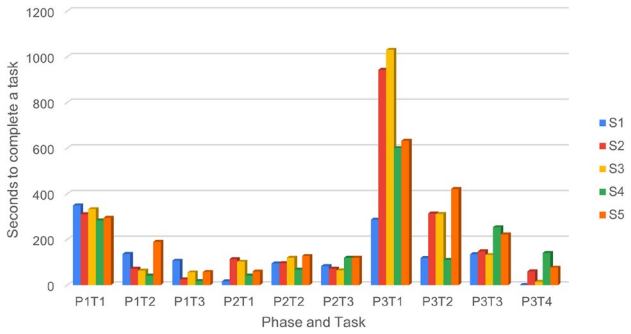


Fig. 10 Comparison of time used for the 3 tasks of the Setup Phase and Configure Phase, and for the 4 tasks of the Use Phase, for all subjects. X axis is phase and task and y axis is time spent in seconds

P3T2 was finished by S1, S3, S4, and S5 without major issues. S2, however, did not complete P3T2. P3T3 and P3T4 were finished without major issues by all subjects.

6 Conclusion

The simulation tool MAES can be used for simulating, and developing, the logic behind swarm robots in a 2 dimensional environment. MAES includes UnityMode for C# development as well as ROSMode, which allows for developing the logic of the robot directly as a ROSNode, and thus bridging the gap to real-world robot programming. Performance tests showed that MAES can be run in real time on modest hardware with up to 5 robots in ROSMode, with up to 120 robots in UnityMode. A usability test was conducted, that shows that MAES can be setup, configured, and used by target audience of robotics researchers and developers within 60 min.

MAES is an ongoing open-source project, released under the GPL-3 license, with a small but growing community around its GitHub code repository. Planned future work include headless runs in ROSMode and position inaccuracies when running in ROSMode. Another interesting feature for the future is that of multiple robot types, since currently MAES only supports a single robot type in each simulation run. Another more complex extension of MAES would be allowing the robots to cover and explore 3D environments, such as underground caves and sewage systems.

MAES is aimed to terrain exploration and coverage. We plan to cover other use cases involving robot movement, such as simulation of a smart factory floor. The main changes would involve the computation of how well an algorithm is behaving, but most other MAES subsystem could be kept like they are.

Unity, which is used to construct MAES, has built-in systems to support safe, high performing multi-threaded code and very efficient memory management called

Data-Oriented Technology Stack (DOTS) [31]. Upgrading MAES with Unity DOTS would likely increase MAES' performance greatly, but we predict it will be a significant feat to accomplish as it would require redesigning and rewriting a lot of MAES' internal workings. However, this would likely only notably affect algorithms running in UnityMode, as the ROS system appears to be the main bottleneck when in ROSMode.

Acknowledgements We thank Simon Bøgh and Casper Schou from the Robotics and Automation department at AAU for their support in the development of the ROS 2 interfaces for MAES. Additionally, we want to thank all of who participated in the usability test. This work was partly funded by the Villum Investigator Project "S4OS: Scalable analysis and Synthesis of Safe, Small, Secure and Optimal Strategies for Cyber-Physical Systems".

Funding Open access funding provided by Aalborg University Library.

References

1. ARGoS (2022) Argos - large-scale robot simulations. https://www.argos-sim.info. Accessed 29 Aug 2023
2. Open Robotics (2022a) Gazebo - simulate before you build. https://gazebosim.org/home. Accessed 29 Aug 2023
3. Agmon N, Hazon N, Gal KA (2008) The giving tree: constructing trees for efficient offline and online multi-robot coverage. Annals Mathemat Artif Intell 52(2):143–168
4. Dorigo M, Guy T, Vito T (2021) Swarm robotics: past, present, and future [point of view]. Proc IEEE 109(7):1152–1165
5. Schranz M, Di Caro GA, Thomas S, Wilfried E, Farshad A, Ahmet Ş, Micha S (2021) Swarm intelligence and cyber-physical systems: concepts, challenges and future trends. Swarm Evolut Comput 60:100762
6. Andreasen M, Holler P, Jensen M, Albano Mi (2022a) Comparison of online exploration and coverage algorithms in continuous space. In: Proceedings of the 14th International Conference on Agents and Artificial Intelligence - Volume 1: SDMIS,, pages 527–537. INSTICC, SciTePress, ISBN 978-989-758-547-0. https://doi.org/10.5220/0010975900003116
7. Unity Technologies (2021) Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. https://unity.com/. Accessed 29 Aug 2023
8. Open Robotics (2022b) Ros - robot operating system. https://www.ros.org. Accessed 29 Aug 2023
9. Cheraghi AR, Abdelgalil A, Graffi K (2020) Universal 2-dimensional terrain marking for autonomous robot swarms. In: 2020 5th Asia-Pacific Conference on Intelligent Robot Systems (ACIRS), pages 24–32. IEEE
10. Albani D, Manoni T, Arik A, Nardi D, Trianni V (2019) Field coverage for weed mapping: toward experiments with a uav swarm. In International Conference on Bio-inspired Information and Communication, pages 132–146. Springer
11. Kambayashi Y, Ugajin M, Sato O, Tsujimura Y, Yamachi H, Takimoto M, Yamamoto H (2009) Integrating ant colony clustering method to a multi-robot system using mobile agents. Ind Eng Manag Syst 8(3):181–193
12. Oikawa R, Takimoto M, Kambayashi Y (2015) Distributed formation control for swarm robots using mobile agents. In: 2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics, pages 111–116. IEEE
13. Kegeleirs M, Grisetti G, Birattari M (2021) Swarm slam: challenges and perspectives. Front Robot AI 8:23
14. Gonzalez E, Gerlein E (2009) Bsa-cm: A multi-robot coverage algorithm. In: 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, volume 2, pages 383–386. IEEE
15. Gautam A, Richhariya A, Shekhawat VS, Mohan S (2018) Experimental evaluation of multi-robot online terrain coverage approach. In: 2018 IEEE International Conference on Robotics and Biomimetics (ROBIO), pages 1183–1189, 10.1109/ROBIO.2018.8665196
16. Gautam A, Soni A, Singh SV, Mohan S (2021) Multi-robot online terrain coverage under communication range restrictions - an empirical study. In: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), pages 1862–1869, 10.1109/CASE49439.2021.9551390
17. Player/Stage (2022) The player project. http://playerstage.sourceforge.net
18. NVIDIA (2022) Nvidia isaac sim. https://developer.nvidia.com/isaac-sim
19. Pitonakova L, Giuliani M, Pipe A, Winfield A (2018) Feature and performance comparison of the v-rep, gazebo and argos robot simulators. In: Annual Conference Towards Autonomous Robotic Systems, pages 357–368. Springer
20. Platt J, Ricks K (2022) Comparative analysis of ros-unity3d and ros-gazebo for mobile ground robot simulation. J Intell Robotic Syst 106(4):80
21. Open Robotics (2022c) Powering the world's robots. https://www.openrobotics.org
22. Farshad A, Jose E, Benjamin B, West A, Watson S, Barry L (2019) Mona: an affordable open-source mobile robot for education and research. J Intell Robot Syst 94(3):761–775. https://doi.org/10.1007/s10846-018-0866-9
23. Fu James GM, Bandyopadhyay T, Ang MH (2009) Local voronoi decomposition for multi-agent task allocation. In: 2009 IEEE International Conference on Robotics and Automation, 1935–1940, 10.1109/ROBOT.2009.5152829
24. Andreasen MZ, JensenMagnus K, Holler PI (2023) Maes. https://github.com/DEIS-Tools/MAES. Accessed 29 Aug 2023
25. Open Robotics (2022d) Ros - robot operating system. https://docs.ros.org/en/galactic/Releases.html. Accessed 29 Aug 2023
26. Unity-Technologies (2022a) Ros tcp connector. https://github.com/Unity-Technologies/ROS-TCP-Connector
27. Unity-Technologies (2022b) Ros tcp endpoint. https://github.com/Unity-Technologies/ROS-TCP-Endpoint
28. Macenski S, Martín F, White R, Ginés CJ (2020) The marathon 2: A navigation system. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). URL https://github.com/ros-planning/navigation2
29. Steve M, Ivona J (2021) Slam toolbox: Slam for the dynamic world. J Open Source Soft 6(61):2783. https://doi.org/10.21105/joss.02783

30. Andreasen M, Holler P, Jensen M (2022b) Maes 2.0: A ros compatible simulation tool for multi robot exploration and coverage. Master's thesis, Aalborg University. available online at https://projekter.aau.dk/projekter/en/studentthesis/maes-20-a-ros-compatible-simulation-tool-for-multi-robot-exploration-and-coverage(03d7a67b-05d4-470e-882d-a5a4da1e1e75).html
31. Unity-Technologies (2022c) Unity dots. https://unity.com/dots. Accessed 29 Aug 2023