https://doi.org/10.1590/jatm.v16.1352

# pyAutonomousAgent: An Academic Tool for Modeling Autonomous Agent Behaviors Using Behavior Trees

Felipe Leonardo Lôbo Medeiros[1,*]

1. Departamento de Ciência e Tecnologia Aeroespacial – Instituto de Estudos Avançados – Divisão C4ISR – São José dos Campos/SP – Brazil.

*Correspondence author: elipefllm@fab.mil.br

## ABSTRACT

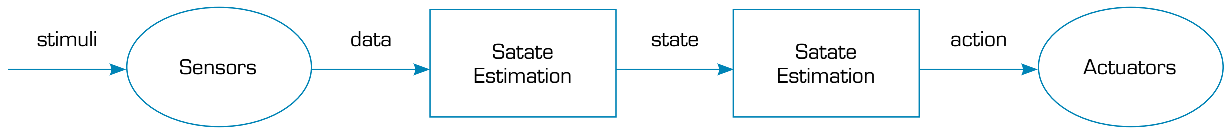
Computer simulations have been applied in several areas. Non-player characters in computer simulations are autonomous agents. An autonomous agent receives information from the simulated environment, processes this information, and estimates a situation awareness (state), makes a decision based on the estimated state, and performs an action related to the decision. The decision-making process of an autonomous agent can be efficiently modeled through behavior trees. However, teaching the construction of behavior trees for autonomous agents can be a very complex task, especially if students have little knowledge of computer programming. The objective of this work is to simplify this task. Therefore, I proposed the pyAutonomousAgent, an academic tool for modeling autonomous agent behaviors through behavior trees. The tool is an open-source set of computer routines, developed with the aim of being easy to use and learn. As a case study, the tool provides a simulation scenario with a swarm of drones. Results of the drone behavior modeling process are presented.

**Keywords:** Autonomous agents; Behavior trees; Constructive simulation; Academic tool; Drone swarm.

## INTRODUCTION

Experiments conducted via computer simulations have important advantages compared with real experiments: economy of resources, safety, and replicability. Due to this fact, computer simulations have been applied in several areas.

Non-player characters in computer simulations are autonomous agents. An autonomous agent receives stimuli from the simulated environment through its set of sensors. It processes data from sensors and estimates a state, which is a set of information from the environment, the agent itself, and other agents. The agent then makes then a decision based on this estimated state. The result of the decision-making process is an action that is sent to its appropriate set of actuators to be executed. Figure 1 presents a simplified structure of an autonomous agent.



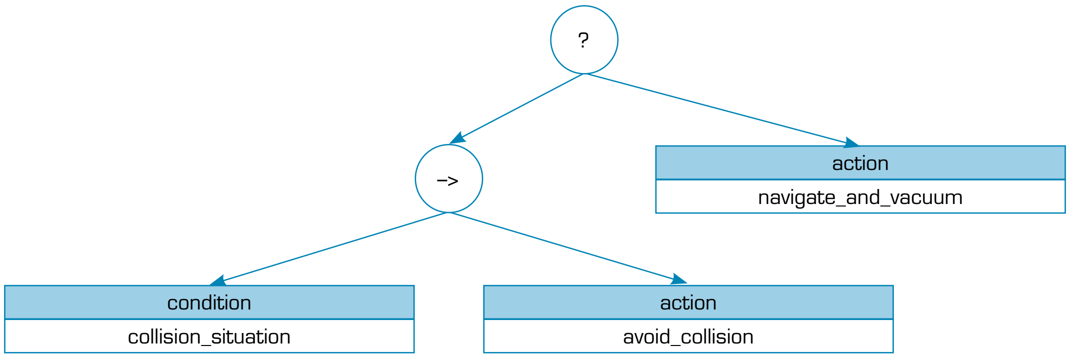
Source: Elaborated by the authors.

**Figure 1.** Simplified structure of an autonomous agent.

**Received:** May 09, 2024 | **Accepted:** Sep. 30, 2024
**Section editor:** Luiz Martins-Filho
**Peer Review History:** Single Blind Peer Review.

We can consider a vacuum cleaner robot as an example of an autonomous agent. It receives data from its set of sensors, including a sonar, processes this data, and estimates a state. The state could be a set containing the robot's location and the distance between the robot and an obstacle detected by the sonar. Through this estimated state, the robot could decide to avoid the obstacle, creating an action for turning left or right. This action would then be sent to the robot's control system actuators.

The decision-making process is the artificial intelligence of an autonomous agent. A behavior tree is an efficient method for modeling the decision-making process of autonomous agent. A behavior tree is a formal language for graphically modeling logical processes that are, in the case of an autonomous agent, behaviors. An example of a behavior tree for a vacuum cleaner robot is presented in Fig. 2. The main elements of a behavior tree are a blackboard, leaf nodes (conditions and actions), control nodes, and decorator nodes. The blackboard is a set of shared values that can be read and written by the nodes. Leaf nodes are the basic elements of a behavior, and can return "success," "failure," or "running," based on their internal procedures regarding the estimated state of the agent and blackboard values. The control nodes, as the name indicates, control the activation of child nodes based on the values returned by these nodes. Each decorator node has a single child node and alters the value returned by the child node or alters the activation of this node. The highest priority nodes are placed in order from top to bottom and from left to right. This is an interesting way to prioritize the most critical behaviors of an agent. Behavior tree nodes are activated through a tick, an execution flow that traverses the tree, following the mentioned order and being controlled by control and decorator nodes. A complete and comprehensive explanation of behavior trees is presented in Colledanchise and Ogren (2020).



Source: Elaborated by the authors.

**Figure 2.** Example of a behavior tree.

In the example of Fig. 2, I consider that the actions always return "success" when they are activated. There are two types of control nodes in this example: sequence, represented by the arrow "→"; and fallback (or selector), represented by the question mark "?". A sequence node begins by activating the first node to the left, and the sequence node activates the next child node while the previous child node returns "success." When a child node returns "failure," the sequence node stops and returns "failure." A fallback node begins by activating the first node to the left, and the fallback node activates the next child node while the previous child node returns "failure." When a child node returns "success," the fallback node stops and returns "successs." Therefore, in considering this example, the collision avoidance action is more important than the actions of navigating and vacuuming. If there is a collision situation, caused by a critical distance between the robot and an obstacle detected by its sonar, then the condition returns "success," and the robot will avoid the collision by performing a left or right turn action. If there is no collision situation, then the condition returns "failure" and the robot will navigate and vacuum.

## Related works

Several tools allow the application of behavior trees for modeling autonomous agent behaviors. Some of these tools are mentioned below. The aerospace simulation environment (*ambiente de simulação aeroespacial* – ASA), presented in Dantas *et al.* (2020), is a powerful tool that enables the creation, execution, and analysis of scenario simulations. FLAMES Simulation Framework (2024) and VR-Forces (2024) are commercially useful tools for developing custom computer simulations. However, these tools are primarily designed for professional applications and require the user to have some experience in simulation and computer programming.

The Robot Operating System (ROS) is a set of computer libraries for developing robot applications (ROS - Robot Operating System 2021). It provides users with robot simulation environments, and behaviors of simulated robots can also be modeled through behavior trees. Although ROS enables a wide range of robotic applications, it requires a lot of software configurations.

There are other tools created for specific applications. A smart home simulation tool was proposed in Ho *et al.* (2019). In this tool, the inhabitants of simulated smart homes are autonomous agents, and their decision-making processes were modeled by behavior trees. A simulation tool for in-orbit assembly of large structures was proposed in Bissonnette *et al.* (2020). Behavior trees were used to model the robotic assembly system's behaviors. The Human Navigation Simulator (HuNavSim), an open-source tool for the simulation of different human-agent navigation behaviors in scenarios with mobile robots, was proposed in Pérez-Higueras *et al.* (2023). These simulated human behaviors were also modeled using behavior trees.

The mentioned tools are professional tools developed for the simulation of real systems and with high computational performance. They were not made for academic purposes and do not didactically present all the phases of simulating an autonomous agent. Using these tools to teach how to model autonomous agent behaviors using behavior trees can be a very complex task, especially if students have little knowledge of computer programming. Furthermore, some of these tools focus on specific simulations and do not allow the user to alter the simulation context to create simpler academic examples.

### Objective

In this work, I propose the pyAutonomousAgent, an academic tool for modeling autonomous agent behaviors through behavior trees. The tool is an open-source set of computer routines, which was developed with the aim of being easy to use and learn. These are the main advantages compared to the previously mentioned tools.

### Article structure

This article is organized as follows. In the Methodology section, I describe the main components of the pyAutonomousAgent tool. In the Results section, I present an example of a behavior tree constructed for a swarm of drones through the application of the proposed tool. Conclusions are also presented, along with some suggestions for future work. Section Source Code presents a link to a repository with the tool's code.

## METHODOLOGY

The simulation process performed by the pyAutonomousAgent tool consists of seven main phases, which are presented in the flowchart in Fig. 3. It was designed and coded in Python programming language considering the format of a Jupyter notebook. This is an easy way for the user to study the tool and make their own modifications to the code.

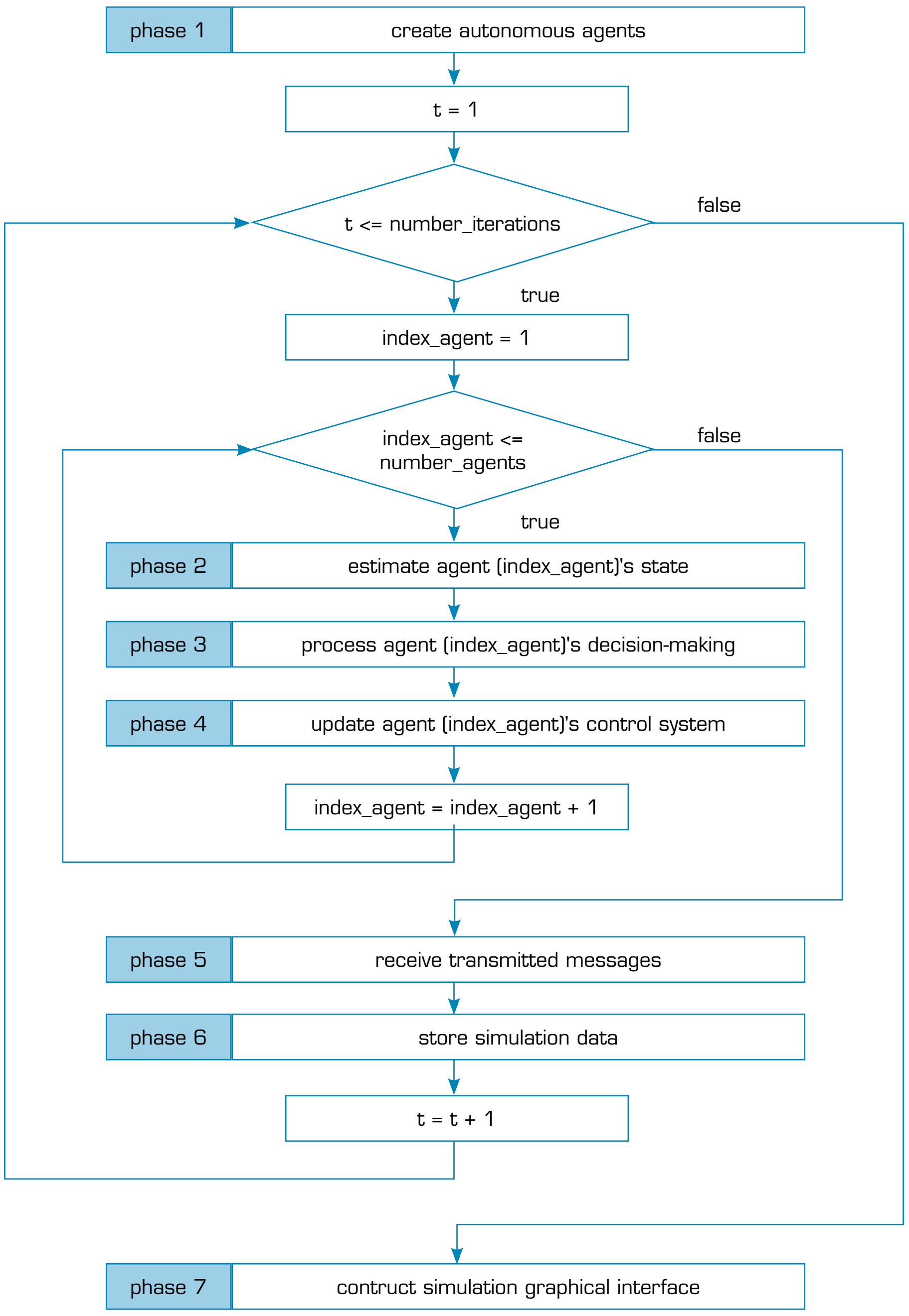
A detailed explanation of each phase is presented in the following subsections.

### Phase 1 – Create autonomous agents

In this phase, a list of autonomous agents is created with the specification of initial conditions such as position and orientation angle. Each autonomous agent is a simplified computer model of a drone, whose structure is like that presented in Fig. 1. This type of autonomous agent was used as a playful example, aiming to facilitate the learning of the basic phases of operation of an autonomous agent. However, pyAutonomousAgent can be easily adapted to model other types of autonomous agents.

The drone has an energy sensor, which returns the energy level of the drone's battery, and a radiation sensor, which returns the level of the energy emitted by a radiation source. The radiation level $r \in [0\%,100\%]$ increases as the drone approaches a radiation source.

The drone has a datalink system that allows the communication with other drones. The message protocol is composed of the following sequence of data: sender ID; list of receiver IDs; message type; message data; and sender's position. The drone can send a message to all drones or specific ones. The sender's position is used to simulate the range of the datalink.


phase 1
create autonomous agents
t = 1
t <= number_iterations
false
true
index_agent = 1
index_agent <= number_agents
false
true
phase 2
estimate agent (index_agent)'s state
phase 3
process agent (index_agent)'s decision-making
phase 4
update agent (index_agent)'s control system
index_agent = index_agent + 1
phase 5
receive transmitted messages
phase 6
store simulation data
t = t + 1
phase 7
contruct simulation graphical interface

Source: Elaborated by the authors.

**Figure 3.** Simulation process executed by the pyAutonomousAgent tool.

The drone's state is a set with the following information: position, which is composed of the Cartesian coordinates x and y without a specific unit of measurement; velocity $v \in [0,0.1]$ units/iteration; orientation angle $\alpha \in [0°,360°]$; energy level $l_e \in [0\%,100\%]$; and damage level $l_d \in [0\%,100\%]$.

The two-dimensional mathematical model of the drone's motion is defined as (Eqs. 1 to 4):

$$x(t) = x(t-1) + v(t-1)\sin(\alpha(t-1)) \tag{1}$$

$$y(t) = y(t-1) + v(t-1)\cos(\alpha(t-1)) \quad (2)$$

$$v(t) = v(t-1) + u_v(t-1) \quad (3)$$

$$a(t) = a(t-1) + u_a(t-1) \quad (4)$$

where: $u_v \in \{-0.1,0.0,0.1\}$ is the velocity control signal and $u_a \in \{-0.1,0.0,0.1\}$ is the orientation control signal.

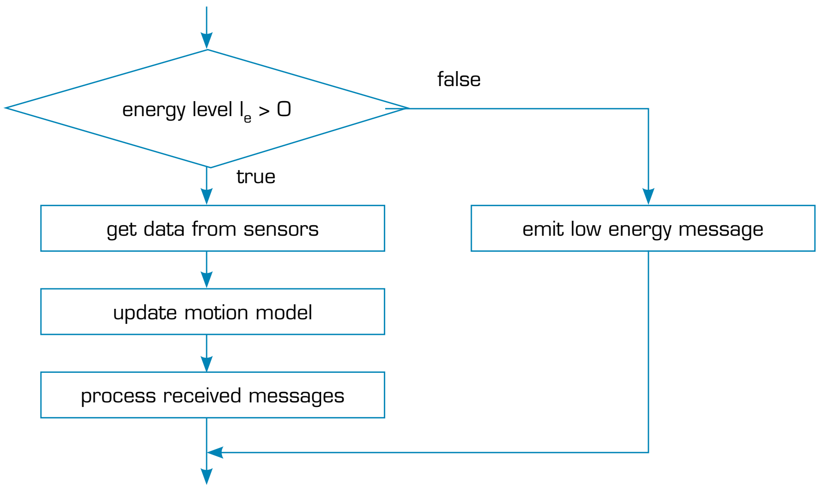
The drone's decision-making is modeled through a behavior tree. I used the Python library py_trees (2020), which is a powerful open-source tool to construct behavior trees.

The drone has two types of actuators: a velocity control actuator and an orientation control actuator. These actuators are used by the drone's control system to execute the actions created by the behavior tree.

## Phase 2 – Estimate agent's state

This phase is presented in the flowchart in Fig. 4. Data provided by the sensors is obtained and the energy and damage levels are updated. After that, the two-dimensional mathematical model of the autonomous agent's motion is updated based on information from the previous iteration and control signals sent by the control system. At this stage, the autonomous agent also receives and processes messages transmitted by other agents.

If the autonomous agent has zero energy, an alert will be then emitted at each iteration. It is important to mention that, if the agent has no energy, then it will not be able to obtain data from the sensors, because it does not have enough energy enough to activate them. Therefore, I consider that the agent refueling process must be responsible for updating the energy level.



Source: Elaborated by the authors.

**Figure 4.** Estimation of the autonomous agent's state.

## Phase 3 – Process agent's decision-making

The autonomous agent's decision-making process is modeled through a behavior tree, as mentioned previously. Thus, all agent behaviors must be represented by logical processes within the tree. The decision-making process begins with a tick on the root node. A tick is the execution flow that is controlled by the control and decorator nodes. A tick propagates through the tree until it reaches a leaf node. A leaf node performs an internal processing based on the state and blackboard information. If the leaf node is a condition node, then this node will return the node status "success" or "failure."

If the leaf node is an action node, then this node will return two pieces of information. The first is the node status, which can be "success" or "failure" if the node is synchronous, or it can be "success," "failure," or "running" if the node is asynchronous. A synchronous node is executed atomically and locks the tree until it returns its status. An asynchronous node returns "running" during its execution and, therefore, does not lock the tree.

The second piece of information is an action, which can be a blackboard action, a motion action, or a communication action. A blackboard action alters one or a set of values of the blackboard. There are two types of motion actions: requested velocity and requested orientation angle. These actions can be understood as reference values that can be reached by the autonomous agent. Therefore, the behavior tree creates these actions to control the agent's motion. A communication action is an action created to transmit a message to other autonomous agents. A detailed example of a behavior tree is described in the Results section.

### Phase 4 – Update agent's control system

The autonomous agent's control system receives the two types of motion actions from the behavior tree. The agent's control system generates velocity and orientation control signals with the objective of decreasing the difference between the reference values, that is, the requested values and the current values.

If the requested velocity $v_r$ is equal to 0.1 and the current velocity $v$ is equal to 0.0, then the velocity control signal $u_v$ must be equal to 0.1. If the requested velocity $v_r$ is equal to 0.0 and the current velocity $v$ is equal to 0.1, then the velocity control signal $u_v$ must be equal to -0.1. If the requested velocity $v_r$ is equal to the current velocity $v$, then the velocity control signal $u_v$ must be equal to 0.0.

As a similar way, if the requested orientation angle $a_r$ is greater than the current orientation angle $a$, then the orientation control signal $u_a$ must be equal to 1.0. If the requested orientation angle $a_r$ is smaller than the current orientation angle $a$, then the orientation control signal $u_a$ must be equal to -1.0. If the requested orientation angle $a_r$ is equal to the current orientation angle $a$, then the orientation control signal $u_a$ must be equal to 0.0.

### Phase 5 – Receive transmitted messages

In this phase, each autonomous agent receives the messages that are transmitted to it. This phase is important to ensure the synchronization of transmission and reception of messages among agents. After message reception, the broadcast channel is cleared.

### Phase 6 – Store simulation data

At the end of an iteration, a set of data from each autonomous agent is stored. This set consists of information about the state of each agent and its blackboard: position, velocity, orientation angle, energy level, damage level, number of ticks, and index of the current surveillance position. This dataset was created for the simulation scenario presented in the Results section. Therefore, this dataset depends on the type of simulation scenario requested for the agents, and additional information can be added to the dataset.

### Phase 7 – Construct simulation graphical interface

In this phase, a graphical interface is created based on the simulation dataset stored in the previous phase. An example of a graphical interface is presented in Fig. 5. This interface was created for the simulation scenario presented in the Results section.

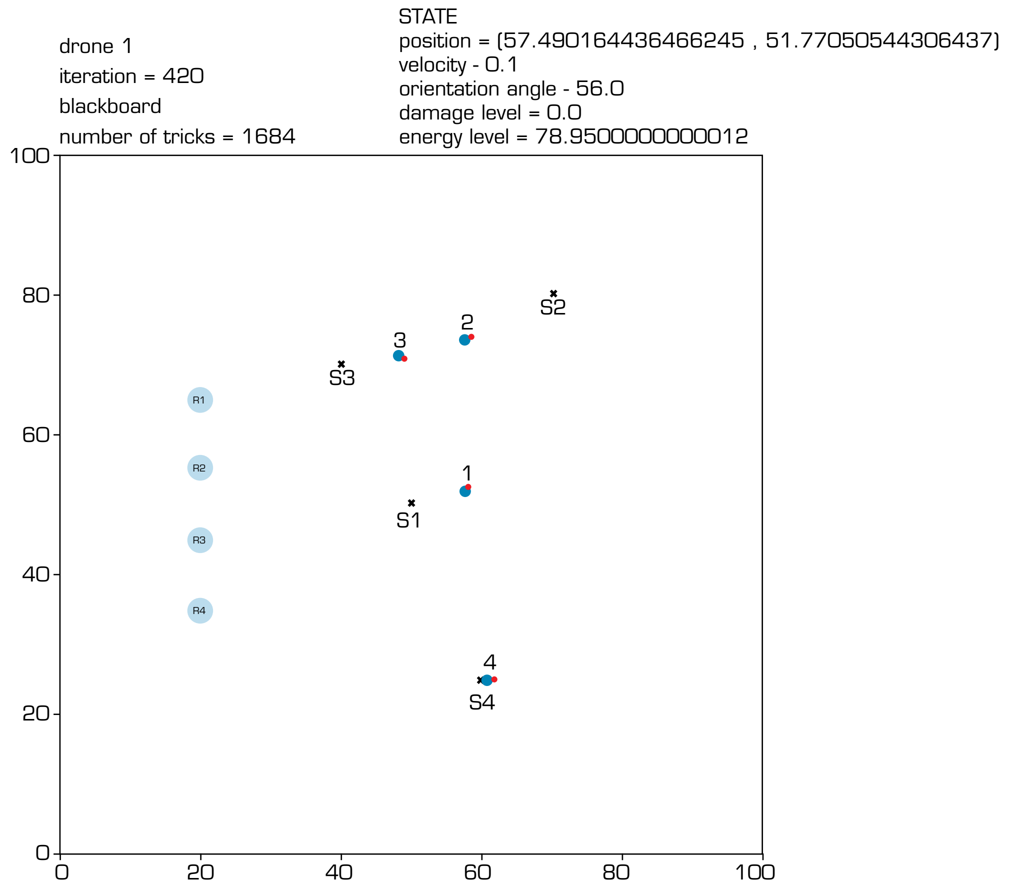
Drones are represented by a dark blue circle with a small red circle, indicating their orientation. Each drone also has its own index. The refueling point is represented by a light blue circle with the index of the respective drone. The surveillance positions are represented by the letter "x" with the index of the respective drone. This graphical interface presents the information related to the first drone in the autonomous agent list.

This phase also creates an animated GIF with frames of this graphical interface. This animated GIF is stored at the file "pyAutonomousAgent_simulation.gif." This file can be converted to video format to facilitate analysis of simulation results.

The graphical interface and the animated GIF are created through the Python library Celluloid 0.2.0. (2018), which is an open-source tool to construct computational animations.

## RESULTS

A simulation scenario was created to explain the procedure for constructing a behavior tree with the pyAutonomousAgent tool. This scenario consists of a swarm of four drones, and it is presented in Fig. 5. A refueling slot and a surveillance position



```
drone 1                              STATE
iteration = 420                      position = (57.490164436466245 , 51.77050544306437)
blackboard                           velocity - 0.1
number of tricks = 1684              orientation angle - 56.0
                                     damage level = 0.0
                                     energy level = 78.9500000000012
```

Source: Elaborated by the authors.

**Figure 5.** Graphical interface of a simulation performed by the pyAutonomousAgent tool.

are assigned to each drone. The drones have two cyclic behaviors that are described as follows: they have to navigate to their respective surveillance positions and orbit those positions. When a drone's energy level reaches 40%, the drone must navigate to its refueling slot and refuel. After that, the drone has to navigate back and orbit its surveillance position again. It is important to mention that collision situations are not considered in this scenario.

At each simulation iteration, the drones must share their state information by transmitting a sharing message to the others. The first drone, i.e., the drone with ID "drone1," has to transmit a gathering message to the others in iteration number 5000. Through this gathering message, the first drone sends its own surveillance position as a message data. Each drone that receives this gathering message has to replace its surveillance position with the received surveillance position.

The construction of a behavior can be divided in three procedures: development of the leaf nodes, creation of the blackboard,assembly of the behavior tree. These procedures are explained in the following subsections.

## Development of the leaf nodes

As explained in phase 3 of the simulation process performed by the pyAutonomousAgent, a leaf node performs internal processing based on the state and blackboard information. The construction of a behavior tree follows the same object-oriented programming methodology. Each leaf node is an object of a node class, i.e., a node type. Analyzing the scenario described previously, I verified the need for nine types or classes of leaf nodes: three types of condition nodes and six types of action nodes. These classes of leaf nodes were developed and are presented in Table 1.

**Table 1.** Description of the classes of leaf nodes.

| Class | Description |
|---|---|
| Condition_Destination_Reached | Returns "success" if the destination position is reached. |
| Condition_Low_Energy_Level | Return "success" if the energy level reaches 40%. |
| Condition_Gathering_Moment | Return "success" if the simulation iteration is equal to 5000 and the drone has an ID equal to "drone1." |
| Action_Navigate_To_Destination_A | Asynchronous action that controls the agent's navigation to the destination position. It always returns "running." |
| Action_Set_Destination_Equal_To_Surveillance_S | Synchronous action that assigns the surveillance position to the destination position. It always returns "success." |
| Action_Set_Destination_Equal_To_Refueling_S | Synchronous action that assigns the refueling position to the destination position. It always returns "success." |
| Action_Transmit_Gathering_Message_S | Synchronous action that transmits the gathering message to other drones, using its datalink. It always returns "success." |
| Action_Transmit_Shared_Data_S | Synchronous action that transmits the data-sharing message to other drones using its datalink. It always returns "success." |
| Action_Refuel_A | Asynchronous action that refuels the agent's energy. It returns "running" while the drone is refueling. If the drone's energy level reaches 100%, then this action will return "success." |

Source: Elaborated by the authors.

A node of the Condition_Destination_Reached class estimates the distance between the drone's position and the destination position, defined by the x_current_destination_position and y_current_destination_position coordinates. If the estimated distance is smaller than or equal to 0.5, the node returns "success," indicating that the destination position was reached. Otherwise, the node returns "failure."

A node of the Condition_Low_Energy_Level class verifies the drone's energy level. If the drone's energy level is smaller than or equal to 40%, the node returns "success" indicating that the drone has low energy. Otherwise, the node returns "failure."

A node of the Condition_Gathering_Moment class returns "success" if the simulation iteration is equal to 5000 and the drone has an ID equal to "drone1." Otherwise, the node returns "failure."

A node of the Action_Navigate_To_Destination_A asynchronous action class controls the agent's navigation to the destination position. It always returns "running." This node reads the current drone's orientation angle and estimates the required orientation angle (requested orientation angle) that the drone must describe to navigate to the destination position. The drone's velocity is not altered by this node. Therefore, the requested velocity is equal to the drone's reference velocity, which is equal to 0.1 in this example. As explained in phase 4, the requested velocity and requested orientation angle are two motion actions that are sent to the drone's control system.

A node of the Action_Set_Destination_Equal_To_Surveillance_S synchronous action class sets the surveillance position as the destination position. It always returns "success."

A node of the Action_Set_Destination_Equal_To_Refueling_S synchronous action class sets the refueling position as the destination position. It always returns "success."

A node of the Action_Transmit_Gathering_Message_S synchronous action class fills the parts of a gathering message – sender, receivers, message type, gathering data, sender's position – and transmits this gathering message to other drones, using its datalink. It always returns "success."

A node of the Action_Transmit_Shared_Data_S synchronous action class fills the parts of a sharing message – sender, receivers, message type, shared data, sender's position – and transmits this sharing message to other drones, using its datalink. It always returns "success."

A node of the Action_Refuel_A asynchronous action class refuels the agent's energy. While the drone is refueling, it sets the requested velocity equal to zero and always returns "running." If the drone's energy level reaches 100%, then it sets the requested velocity equal to the drone's reference velocity, and it returns "success." Throughout its operation, this node sets the requested orientation angle equal to the current orientation angle.

The leaf nodes created for the simulation scenario are presented in Table 2 with their respective classes. Leaf nodes can be thought of as blocks of a puzzle toy. The requested behavior of the simulation scenario is the puzzle. There are blocks (objects) of different types (classes), and some blocks can be of the same type. These blocks must be combined to build the puzzle. Therefore, the leaf nodes in Table 2 must be combined in such a way that the drone's behavior mimics the behavior requested in the simulation scenario description.

**Table 2.** Leaf nodes with their respective classes.

| Leaf node | Class |
|---|---|
| condition_destination_reached | Condition_Destination_Reached |
| condition_low_energy_level | Condition_Low_Energy_Level |
| condition_gathering_moment | Condition_Gathering_Moment |
| action_navigate_to_destination_a_1 | Action_Navigate_To_Destination_A |
| action_navigate_to_destination_a_2 | Action_Navigate_To_Destination_A |
| action_set_destination_equal_to_surveillance_s | Action_Set_Destination_Equal_To_Surveillance_S |
| action_set_destination_equal_to_refueling_s | Action_Set_Destination_Equal_To_Refueling_S |
| action_transmit_gathering_message_s | Action_Transmit_Gathering_Message_S |
| action_transmit_shared_data_s | Action_Transmit_Shared_Data_S |
| action_refuel_a | Action_Refuel_A |

Source: Elaborated by the authors.

It is important to emphasize that a behavior tree can be composed of many nodes of the same class. Analyzing Table 2, we can verify that there are two leaf nodes of the Action_Navigate_To_Destination_A class. They are differentiated using the indexes "1" and "2." The node with index "1" is used to control the drone's navigation to the surveillance position. When the drone reaches the surveillance position, it will return, trying to reach that position again. I considered this return movement as the orbital movement. The node with index "2" is used to control the drone's navigation to the refueling position.

## Creation of the blackboard

The blackboard is composed of data that can be shared by leaf nodes in the behavior tree. The blackboard for the nodes described in the previous subsection is presented in Table 3.

**Table 3.** Blackboard.

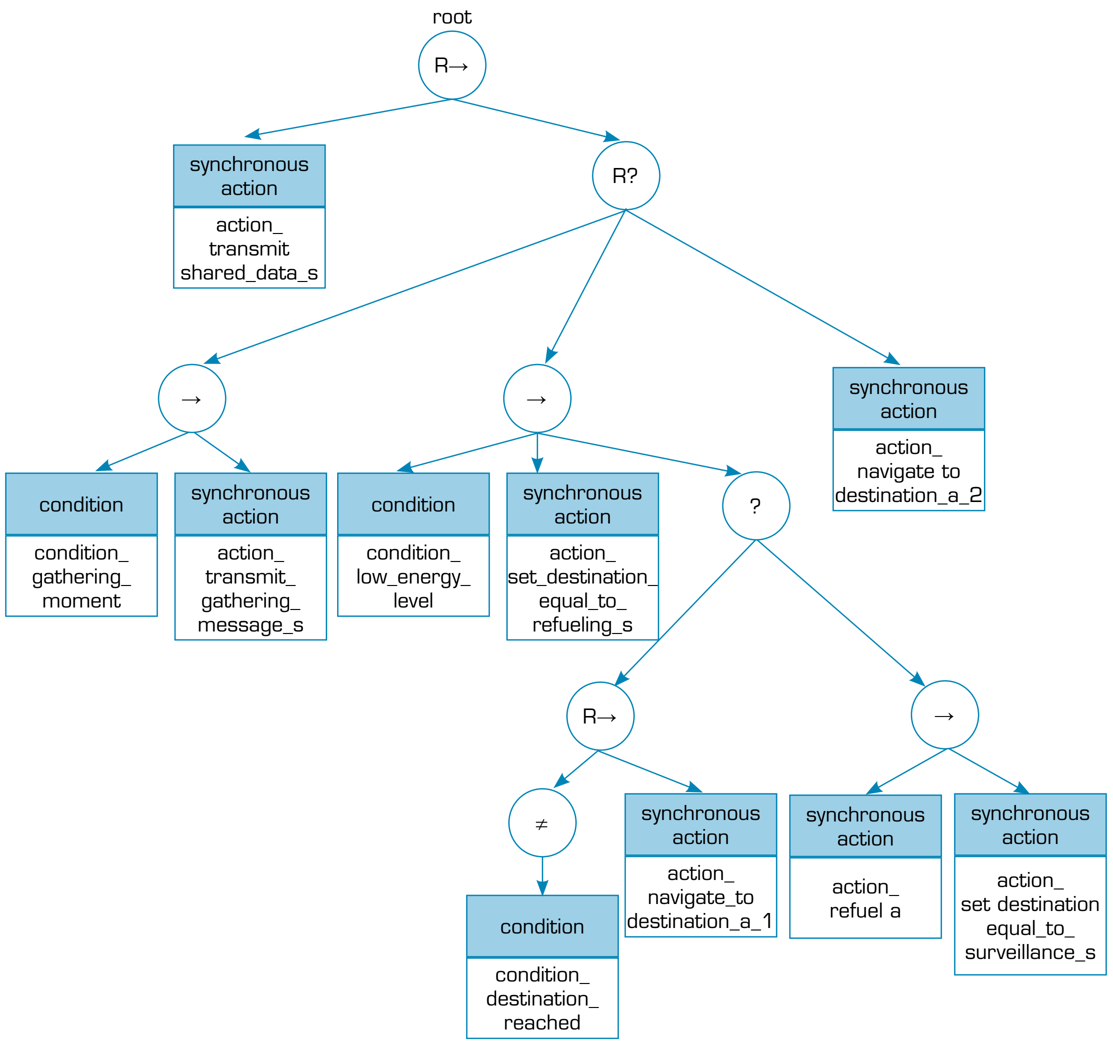
| Data | Description |
|---|---|
| number_ticks | It indicates how many times the leaf nodes were ticked. |
| x_surveillance_position | x-coordinate of the current surveillance position |
| y_surveillance_position | y-coordinate of the current surveillance position |
| x_refueling_position | x-coordinate of the refueling position |
| y_refueling_position | y-coordinate of the refueling position |
| x_current_destination_position | x-coordinate of the current destination position |
| y_current_destination_position | y-coordinate of the current destination position |

Source: Elaborated by the authors.

The number_ticks data is used to indicate the total number of ticks, that is, the number of times that leaf nodes are triggered by the behavior tree. The x_surveillance_position and y_surveillance_position data indicate the coordinates of the surveillance position specified for that drone. The surveillance position is specified for each drone at the beginning of the simulation. However, when a drone receives a gathering message from the drone with ID "drone1," this drone sets its surveillance position equal to the surveillance position of the drone with ID "drone1." The x_refueling_position and y_refueling_position data indicate the coordinates of the refueling slot specified for that drone. The refueling position is defined at the beginning of the simulation. The x_current_destination_position and y_current_destination_position data indicate the coordinates of the current destination position that the drone must navigate to. The destination position is altered by the action nodes action_set_destination_equal_to_surveillance_s and action_set_destination_equal_to_refueling_s.

## Assembly of the behavior tree

After the development of the leaf nodes and the creation of the blackboard, the behavior tree was assembled as presented in Fig. 6. There are four types of control nodes in this tree: sequence ("→"); reactive sequence ("R→"); fallback ("?"); and reactive fallback ("R?").


root
R→
synchronous action
action_ transmit shared_data_s
R?
→
→
synchronous action
action_ navigate to destination_a_2
condition
condition_ gathering_ moment
synchronous action
action_ transmit_ gathering_ message_s
condition
condition_ low_energy_ level
synchronous action
action_ set_destination_ equal_to_ refueling_s
?
R→
→
≠
condition
condition_ destination_ reached
synchronous action
action_ navigate_to destination_a_1
synchronous action
action_ refuel a
synchronous action
action_ set destination equal_to_ surveillance_s

Source: Elaborated by the authors.

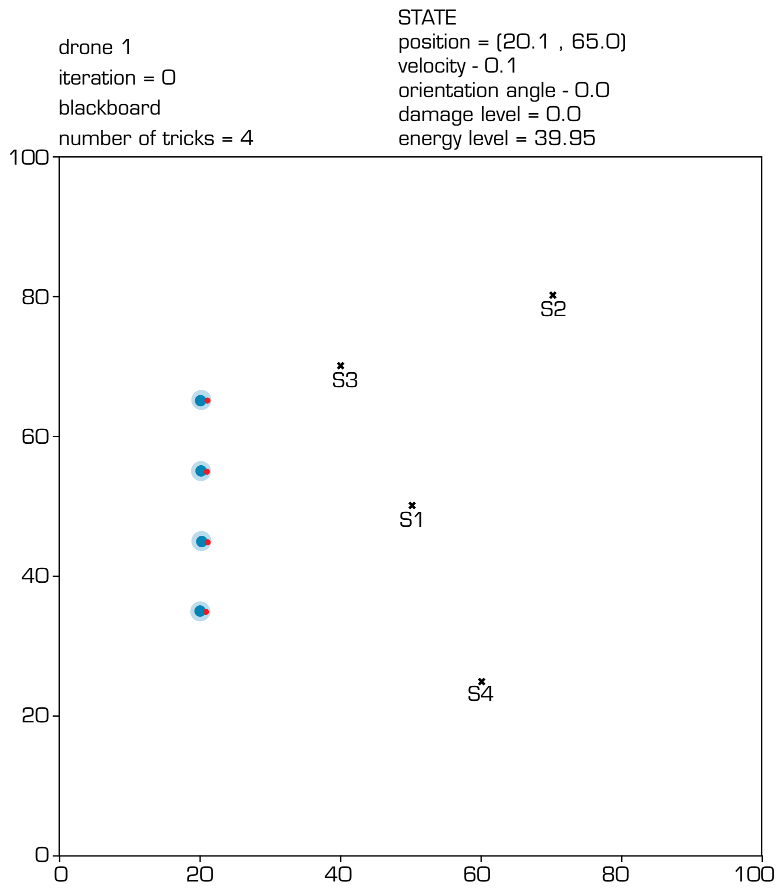
**Figure 6.** Behavior tree assembled for the simulation scenario.

As explained previously, a sequence node begins by activating the first node to the left, and the sequence node activates the next child node while the previous child node returns "success." When a child node returns "failure," the sequence node stops and returns "failure." When an a synchronous child node returns "running," the sequence node activates this node until this node returns "success" or "failure." There is only one difference between a reactive sequence node and a sequence node. When a child node returns "running," the reactive sequence restarts and activates the first node to the left.

A fallback node begins by activating the first node to the left, and the fallback node activates the next child node while the previous child node returns "failure." When a child node returns "success," the fallback node stops and returns "success." When an asynchronous child node returns "running," the fallback node activates this node until it returns "success" or "failure." Similarly, there is only one difference between reactive fallback node and a fallback node. When a child node returns "running," the reactive fallback restarts and activates the first node to the left.

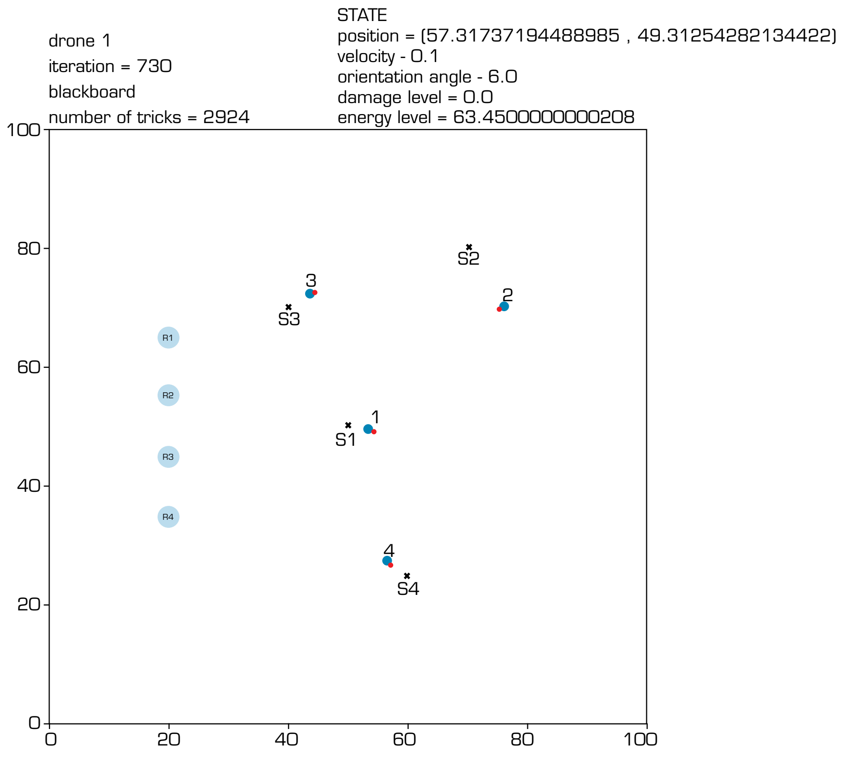
There is also a decorator node in this tree: the inverter ("≠"). This node inverts the value returned by a child node. If the child node returns "success," then the inverter node returns "failure." If the child node returns "failure," then the inverter node returns "success."

Four situations of the swarm of drones are presented in the Figs. 7–11. The first situation presents the initial moment of the simulation, where each drone is in its respective refueling slot. The second situation presents each drone orbiting its respective surveillance position. In the third situation, the second drone, that is, the drone with ID "drone2," is refueling. The fourth situation presents all drones orbiting the surveillance position of the drone "drone1." This situation occurred after the gathering message has been transmitted from the drone "drone1."



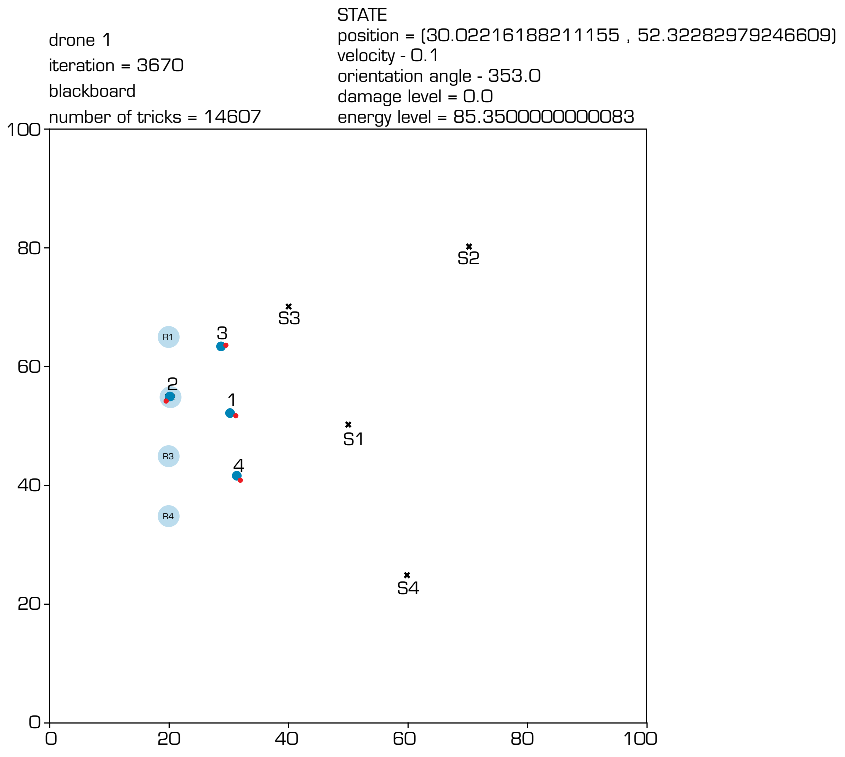
Source: Elaborated by the authors.

**Figure 7.** Initial moment of the simulation.



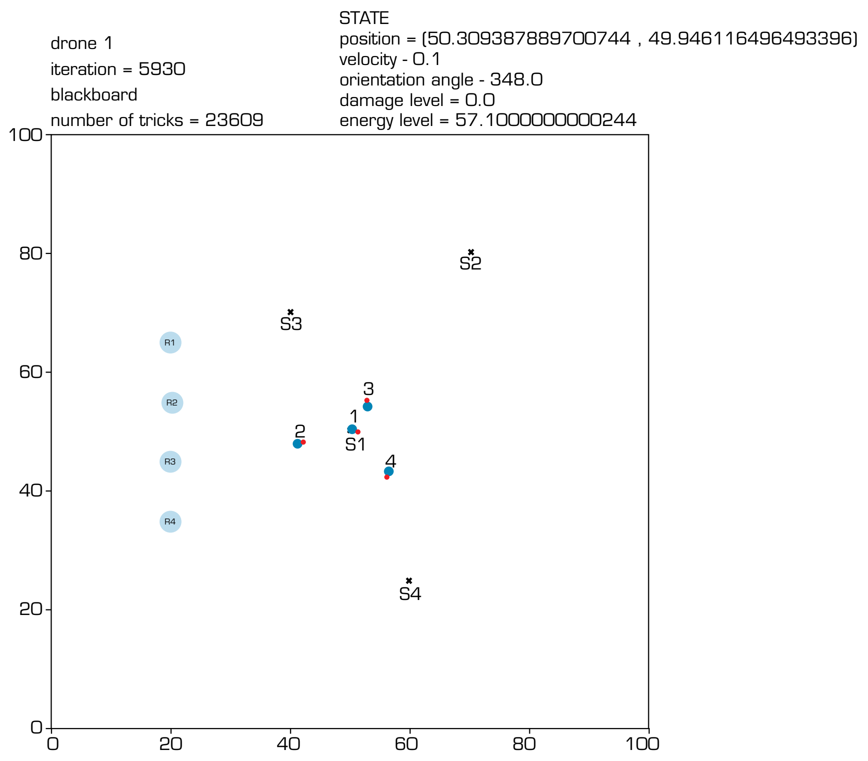
Source: Elaborated by the authors.

**Figure 8.** Drones orbiting their respective surveillance positions.



Source: Elaborated by the authors.

**Figure 9.** Second drone is refueling.

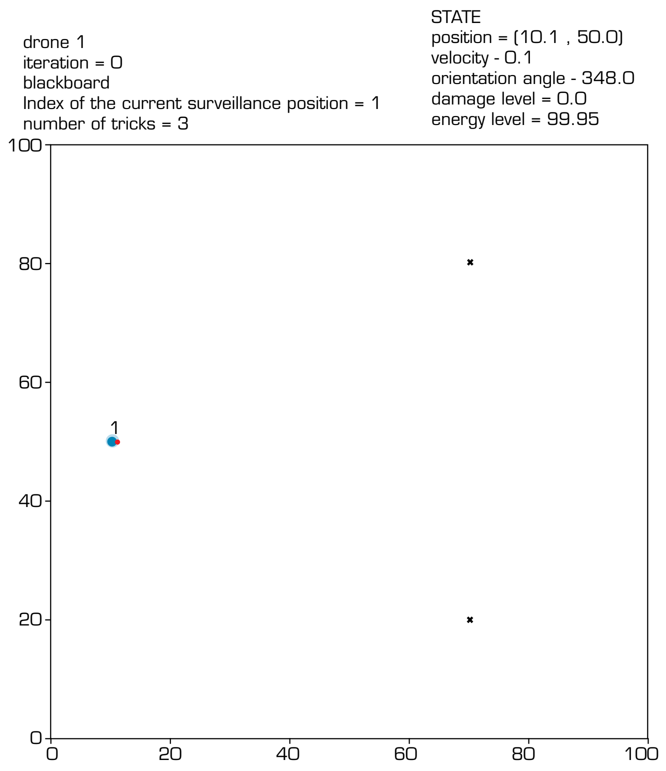


drone 1
iteration = 5930
blackboard
number of tricks = 23609

STATE
position = (50.309387889700744 , 49.946116496493396)
velocity - 0.1
orientation angle - 348.0
damage level = 0.0
energy level = 57.1000000000244

Source: Elaborated by the authors.

**Figure 10.** All drones orbiting the surveillance position of the first drone.



drone 1
iteration = 0
blackboard
Index of the current surveillance position = 1
number of tricks = 3

STATE
position = (10.1 , 50.0)
velocity - 0.1
orientation angle - 348.0
damage level = 0.0
energy level = 99.95

Source: Elaborated by the authors.

**Figure 11.** Scenario created for the first set of performance tests.

## Performance tests

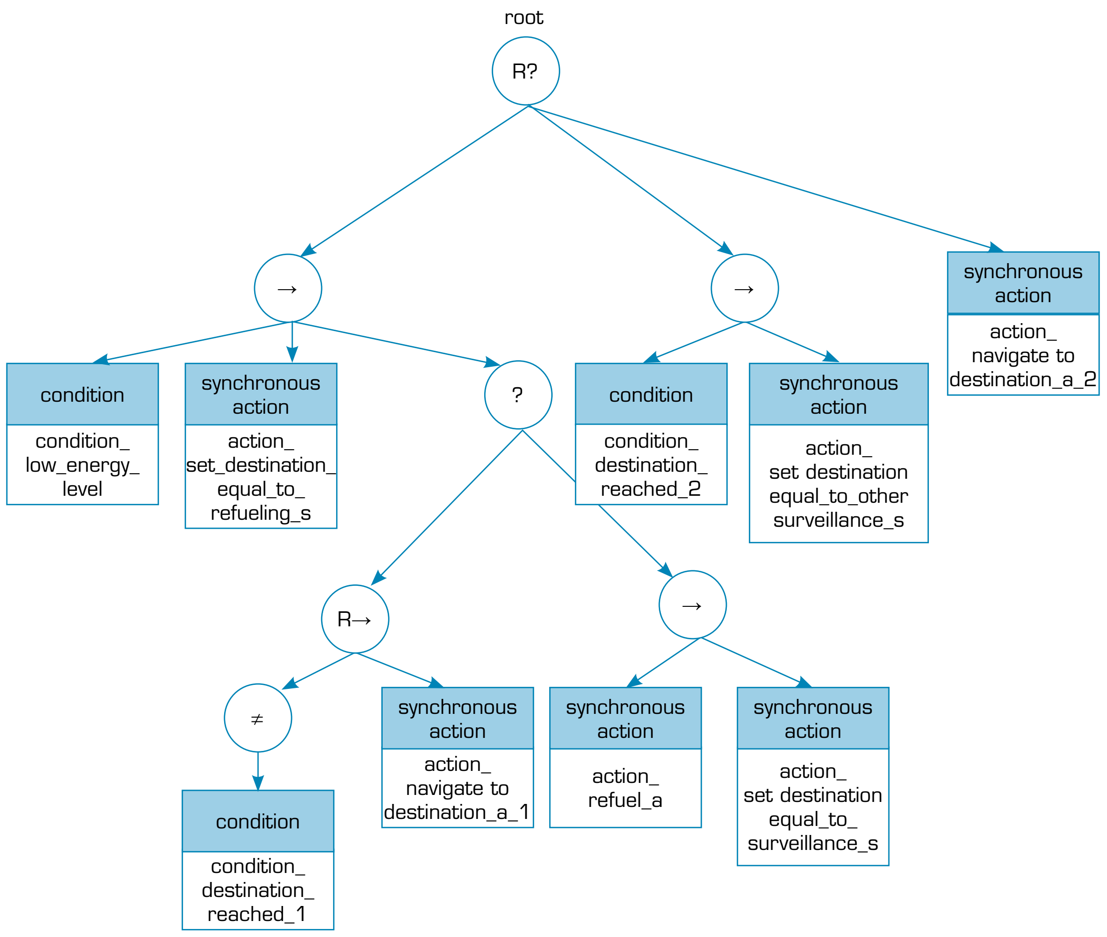
A set of experiments was conducted to analyze the tool's performance. The performance experiments were performed on a computer with the following configuration: 2 GHz processor; 16 GB of RAM; and 16 MHz of cache memory.

Although I carried out these performance tests, I would like to emphasize that the purpose of the tool is purely academic. Thus, the focus is on didactics and not on performance.

The experiments were divided into two sets: tests with the simplified model of a drone described in the Methodology section, and tests with a model of a realistic system.

The performance tests of the first set were accomplished with the scenario presented in Fig. 11.

A swarm of drones navigates from a refueling slot, represented by the blue circle, to the first surveillance position located at the top of the navigation environment. Upon reaching the first surveillance position, the drones navigate to the second surveillance position, which is located at the bottom of the environment. When the second surveillance position is reached, the drones navigate to the first one. When a drone's energy level reaches 40%, the drone must navigate to its refueling slot and refuel. The drones will cyclically alternate navigation between the two surveillance positions until a limit of 4,000 iterations. The behavior tree constructed for this simulation scenario is presented in Fig. 12. The graphical interface (phase 7) was not considered in these tests, only the simulation.


root
R?
→
→
synchronous action
action_ navigate to destination_a_2
condition
condition_ low_energy_ level
synchronous action
action_ set_destination_ equal_to_ refueling_s
?
condition
condition_ destination_ reached_2
synchronous action
action_ set destination equal_to_other surveillance_s
R→
→
≠
synchronous action
action_ navigate to destination_a_1
synchronous action
action_ refuel_a
synchronous action
action_ set destination equal_to_ surveillance_s
condition
condition_ destination_ reached_1

Source: Elaborated by the authors.

**Figure 12.** Behavior tree assembled for the performance tests.

I made a comparison of computational time, considering scenarios containing from 1 to 100 drones, varying with a step of 10 drones. After that, I made the same comparison for scenarios containing 200 to 1,000 drones, varying with a step of 100 drones. The results are presented in Table 4. The tool processed a single drone with an average computational time of 4.45e-5 s per iteration. During testing, the RAM used by the tool varied from 88.5 MB (1 drone) to 1,495.8 MB (1,000 drones).

**Table 4.** Results from the first set of performance tests.

| Number of drones | Processing time (s) |
|---|---|
| 1 | 0.19 |
| 10 | 2.02 |
| 20 | 3.48 |
| 30 | 5.12 |
| 40 | 6.99 |
| 50 | 8.79 |
| 60 | 10.51 |
| 70 | 12.07 |
| 80 | 14.03 |
| 90 | 15.59 |
| 100 | 17.32 |
| 200 | 35.41 |
| 300 | 52.61 |
| 400 | 70.24 |
| 500 | 89.12 |
| 600 | 106.74 |
| 700 | 127.87 |
| 800 | 146.69 |
| 900 | 161.41 |
| 1,000 | 177.41 |

Source: Elaborated by the authors.

I also conducted a stress test, increasing the number of drones until reaching the maximum available RAM, which was 5.12 GB (32% of the total). The remaining RAM, 10.88 GB (68% of the total), was being used to run other programs on my computer. In this way, I obtained the a maximum of 3,550 drones, with a computational time of 687.37 s.

The PyFly tool was used to simulate a real unmanned aerial vehicle (UAV) inside the pyAutonomousAgent tool. PyFly is an open-source Python implementation of 6 degrees of freedom (DOF) aerodynamic models for fixed wing aircraft (Bohn *et al.* 2019). PyFly provides a computational implementation of the aerodynamic model of the Skywalker X8 UAV and its respective proportional-integral-derivative (PID) controllers. This aerodynamic model of the Skywalker X8 UAV was proposed and validated in Gryte *et al.* (2018). Swarms of this UAV were used in the second set of tests.

The scenario for the second set of tests is similar to the scenario for the first set of tests. I only made an adaptation by changing the scale of the navigation environment to 1,000 m × 1,000 m. Each simulation consisted of 4,000 iterations, and each iteration corresponded to a simulated time step of 0.01 s, that is, *dt* was equal to 0.01 s. The behavior tree presented in Fig. 12 was also used in the second set of tests.

I developed an autopilot for the UAV consisting of an altitude PID controller and a track angle PID controller. The track PID controller was created because of the sideslip effect of the aerodynamic model. The velocity was controlled by the PID controller provided by the PyFly tool. I considered a velocity of 22 m·s and an altitude of 20 m in all the experiments.

I made a comparison of computational time considering scenarios containing from 1 to 100 drones, varying with a step of 10 drones. The results are presented in Table 5.

**Table 5.** Results from the second set of performance tests.

| Number of UAVs | Processing time (seconds) |
|---|---|
| 1 | 20.89 |
| 10 | 210.97 |
| 20 | 426.16 |
| 30 | 654.26 |
| 40 | 887.67 |
| 50 | 1,097.65 |
| 60 | 1,345.95 |
| 70 | 1,534.98 |
| 80 | 1,650.53 |
| 90 | 1,797.88 |
| 100 | 1,955.72 |

Source: Elaborated by the authors.

The tool processed a single Skywalker X8 UAV with an average computational time of 5.31e-3 s per iteration. During testing, the RAM used by the tool varied from 124.1 MB (1 UAV) to 1,904.5 MB (100 UAVs).

I conducted a stress test similar to the one performed for the first set of tests, increasing the number of UAVs until reaching the maximum available RAM, which was 5.12 GB (32% of the total). I obtained a maximum of 290 UAVs, with a computational time of 8,370.96 s. All tests in the first and second parts used a maximum of 17.2% of the processor's total capacity.

Due to the complexity of the UAV aerodynamic model, we can verify that the computational time required to process the UAV model simulation is significantly higher than the computational time to process the drone model.

I also carried out a series of long-term experiments (200,000 iterations) for the purpose of testing the actions of refueling and navigation. It is important to mention that the developed autopilot does not allow for the landing and take-off procedures that are important for the refueling action. An interesting future work would be the adaptation of the autopilot to perform these procedures.

Analyzing the long-term experiments, I noticed that at the beginning of the refueling action, the UAV started to decrease its speed, which caused its altitude to decrease as well. The UAV was then refueled at a rate of 2% per iteration, i.e., 2% every 0.01 s. Therefore, we can conclude that the UAV was fully refueled after a maximum of 50 iterations in the worst case, i.e., after 0.5 s. Thus, the refueling action caused a small displacement of the UAV and small changes in speed and altitude.

## DISCUSSION

In this work, I proposed the pyAutonomousAgent, an academic tool for modeling autonomous agent behaviors through behavior trees. The tool is an open-source set of computer routines, which are designed to be as simple as possible, aiming to facilitate learning for students with little knowledge of simulation tools and computer programming.

The pyAutonomousAgent tool allows for the modeling of all operating phases of an autonomous agent. This tool provides an example of a drone swarm simulation scenario, which includes a detailed explanation of constructing a behavior tree for an autonomous agent. It can also be modified to model other types of drone swarm simulation scenarios.

Future work directions include creating a simplified three-dimensional mathematical model of motion for drones and adding of collision situations for these drones. Another interesting future work would be the adaptation of the Skywalker X8's autopilot to perform the landing and take-off procedures.

## CONFLICT OF INTEREST

Nothing to declare.

## DATA AVAILABILITY STATEMENT

The source code of this proposed tool is provided in the repository
https://github.com/felipefllm/pyAutonomousAgent

## FUNDING

Not applicable.

## ACKNOWLEDGEMENTS

To thank Lieutenant Adrisson Rogério Samersla, for helping with some questions related to object-oriented programming in the python programming language.

## REFERENCES

Bissonnette V, Bazerque C, Trinh S, Porte C, Arcin G, Rognant M, Biannic JM, Cumer C, Loquen T, Pucel X (2020) A simulation tool for in-orbit assembly of large structures. Paper presented 2020 International Symposium on Artificial Intelligence, Robotics and Automation in Space. Lunar and Planetary Institute and Universities Space Research Association; Pasadena, USA.

Bohn E, Coates EM, Moe S, Johansen TA (2018) Deep reinforcement learning attitude control of fixed-wing UAVs using proximal policy optimization. Paper presented 2019 International Conference on Unmanned Aircraft Systems. ICUAS Association; Atlanta, USA.

Celluloid 0.2.0. San Francisco Bay Area (CA): Jacques Kvam; c2018 [accessed May 9 2024]. https://pypi.org/project/celluloid/

Colledanchise M, Ogren P (2020) Behavior trees in robotics and AI: an introduction. Boca Raton: CRC Press.

Dantas J, Costa AN, Gomes VC, Kuroswiski AR, Medeiros FLL, Geraldo D (2022) ASA: a simulation environment for evaluating military operational scenarios. Paper presented 2022 20th International Conference on Scientific Computing. American Council on Science and Education; Las Vegas, USA.

FLAMES Simulation Framework. Huntsville (AL): Ternion Corporation; c2024 [accessed May 9 2024]. https://flamesframework.com/

Gryte K, Hann R, Alam M, Rohác J, Johansen TA, Fossen TI (2018) Aerodynamic modeling of the Skywalker X8 fixed-wing unmanned aerial vehicle. Paper presented 2018 International Conference on Unmanned Aircraft Systems. ICUAS Association; Dallas, USA.

Ho B, Vogts D, Wesson J (2019) A smart home simulation tool to support the recognition of activities of daily living. Paper presented 2019 South African Institute of Computer Scientists and Information Technologists. SAICSIT; Skukuza, South Africa.

Pérez-Higueras N, Otero R, Caballero F, Merino L (2023) HuNavSim: a ROS 2 Human Navigation Simulator for benchmarking human-aware robot navigation. IEEE Robot Autom Lett 8(11):7130-7137. https://doi.org/10.1109/LRA.2023.3316072

py_trees. [Internet]. Boston (MA): Daniel Stonier; c2020 [accessed May 9 2024]. https://py-trees.readthedocs.io/en/devel/#

ROS - Robot Operating System. San Jose (CA): Open Robotics; c2021 [accessed May 9 2024]. https://www.ros.org/

VR-Forces: the MAK one multi-domain computer generated forces. Cambridge (MA): MAK Technologies; c2024 [accessed May 9 2024]. https://www.mak.com/mak-one/apps/vr-forces