# 3806ICT Assignment 3: A Model Checking based planning and goal reasoning framework for pathfinding

Bennett Taylor – s5095512, Denis Chizhov – s5275427, Clayton Johnson – s5009099

Griffith University, Gold Coast, AUS

**Abstract – Planning is integral in the development of autonomous systems, as they need to plan the tasks they carry out in order to achieve their goal. These tasks' increasing complexity does not let human designers anticipate all possible situations, making traditional control systems based on state machines are not enough anymore. This paper proposes a system that follows a "verification as planning" paradigm and uses model-checking techniques to solve planning and goal reasoning problems for autonomous systems. The system uses an industrial model checker (PAT) in close relation to ROS, to aid an autonomous system through a maze. This maze is automatically generated each time the system runs and can be specified to any length and width. The PAT modeller uses breadth and depth first search to solve these mazes in under a few seconds. PAT also provides other useful information to the system such as a tree listing solution move-set and whether the maze can be solved or not. The results of this paper can be useful for creating new game engines that auto-generate maps aid in the development of new pathfinding techniques.**

## I. INTRODUCTION

For decades now, symbolic planning has been a significant field within Artificial Intelligence, even being applied by the first generation of robots to plan their tasks [1] [2]. Over the years, this field has remained actively researched and has consistently maintained its relevance [3] [4]. One of the key factors contributing to its popularity is the development of planning languages such as STRIPS [5] and its successor PDDL [6]. These provide a common framework for integrating planning algorithms developed by the scientific community. The standardisation of planning languages has also facilitated the establishment of planning competitions [7], enabling researchers to create benchmarks for measuring and comparing planning algorithms. POPF (Forward-Chaining Partial-Order Planning) [8] and TFD (Temporal Fast Downward) [9] are some other notable examples emerging from the realm of planning algorithms. In the field of robotics, planning systems such as ROSPlan [10] has emerged as the most widely used planning system in the past decade, with over 28 contributors and over a thousand commits in the last six years. ROSPlan has powered numerous robotics projects over the years, including ones on land [11] [12], in the air [13] and underwater [14]. The success of ROSPlan can be attributed, in part, to its status as the reference package for Planning in ROS, which has become the de facto standard in robot programming.

This paper follows the development of a system that combines the power of ROS (Robot Operating System) and PAT (Process Analytical Toolkit). ROS is an open-source framework widely used in the field of robotics due to its collection of software libraries and tools that aid in building robot applications. ROS provides a wide range of functionalities and capabilities that we can leverage in our system. It offers infrastructure for inter-module communication, standardised message types for exchanging data, packages for sensor integration, motion control, mapping and localization. ROS is also able to help bridge the gap between abstract plans generated by PAT and the execution of those plans. These features enable us to implement a robust and efficient system that offers the low-level control and communication necessary to realise the plans generated by PAT.

PAT is an analytical framework used in the field of process control and optimization. It integrates various tools and techniques for real-time monitoring, analysis, and control of industrial processes. PAT enables the collection and analysis of process data, leading to improved process understanding, enhanced product quality, and increased efficiency. PAT is used because it serves as a valuable resource for both model checking and simulation purposes. It enables a user to obtain a high-level overview of a desired process, while also offering the capability to identify deadlocks, divergence, reachability, and probabilities. By integrating PAT into our system, we can leverage its analytical tools to monitor and analyse various aspects of the environment and the robot's performance. By incorporating PAT's process analytical capabilities, we can enhance the overall performance, adaptability, and efficiency of our software robot system.

## II. RELATED WORK

Several studies have explored the integration of ROS and with planning frameworks for robotics tasks. Shakey [15], developed between 1966 and 1972, was the pioneering robot to employ symbolic planning. This robot served as a testbed for STRIPS [16], one of the earliest languages for AI planning. The Rhino robot approach [17] focused on generating action plans to guide visitors through the Bonn Museum, while Dervish [18] autonomously calculated its action plan, winning the Office Delivery event at the AAAI 1994 Robot Competition. These systems however, faced computational limitations associated with their mechanisms. However, these systems faced computational limitations associated with their mechanisms. The challenge of integrating planning and action in robots operating in complex and dynamic environments prompted researchers like Arkin [19] and Brooks [20] to explore alternative techniques that deviate from symbolic planning-

based approaches. Advancements in technology have shifted the trend seen recently due to increased computing power possessed by robots. Nonetheless, limitations present within subsymbolic systems compel changes; they do not meet explainability requirements or engage appropriately in symbolic interaction with humans. To bridge this gap between plans and real-world implementation, cognitive architectures like CORTEX [21] now incorporate symbolic planners. While these architects have rendered enormous assistance towards meeting this aim; their inflexibility poses an obstacle when modifying their planning system or presenting direct utility with related laws concerning different structures. Thanks to standardised programming frameworks for robots' ease of programming with a new standard plenipotent larger community can enhance applications worldwide; notably popular among them is ROSPlan [10], serving as a reference planning system in the ROS ecosystem.

## III. PROPOSED APPROACH

The objective is to create a software robot capable of efficient pathfinding through a variety of mazes. Developing a system that combines ROS and PAT involves integrating various modules to achieve a holistic solution. Our system design is inspired by Bride et al.'s work in Fig.1, however we will be using ROS instead of MOOS.

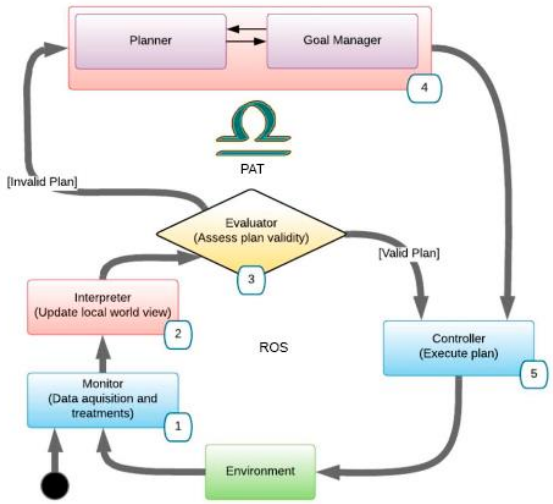


**Fig. 1.** *Workflow Model*

In this approach PAT handles high-level abstract planning and goal reasoning, while ROS focuses on low-level actuation and interaction with the environment. The system is split into four modules which are:

1. An Environment Interaction Module: This module is responsible for interacting with the environment and obtaining readings from sensors. These readings will be crucial for understanding the current state of the environment.
2. A Sensor Data Processing Module: This module processes the readings from the sensor and generates the agent's state to understand the environment.
3. A PAT Interaction Module: The integration with PAT is a critical aspect of our system. This module interacts with PAT by providing the agent's state and an abstract representation of the environment (using a CSP model). This allows PAT to perform advanced reasoning and optimization tasks on the environment.
4. A Low-level Plan Translation Module: This Module waits for PAT to provide a result, which will be a high-level plan for the environment. Once PAT provides a result, the module takes the output and translates it into a low-level plan that can be executed by ROS. The execution of this plan may lead to changes in the environment, this will trigger the first module to perform another interaction cycle until the goal is achieved.
5. A Simulator Node Module: In addition to the system modules, a simulator will be developed as a separate component. The simulator will provide a virtual environment that the robot can navigate and interact with via its sensor. It will receive the robot's movements and the effects of executing plans, as well as publish data describing the robots environment post execution of PAT result. The simulator allows real-time observation and evaluation of the system's performance in a controlled and reproducible manner without the need for physical robots or real-world environments.

## IV. IMPLEMENTATION

**Maze generation:**

This paper also utilises an implementation for automatic generation of mazes. Python was chosen as the programming language for this implementation due to its simplicity and efficiency in creating such procedures. This implementation generated mazes of various sizes, ranging from size of 5x5 up to 1000x1000. A maze map is defined as a 2D matrix of ASCII characters where 'O' represents an open cell that the agent can traverse through and 'X' represents a wall or obstacle that the agent cannot move onto (as seen in Fig. 2). The cell marked with a 'G' represents the goal position that the agent will have to reach in order to achieve the task. The cell marked with a 'S' represents the agent's starting position. The agent may move around the maze by taking 'steps', where each step may involve moving right, left, down or up by one cell.

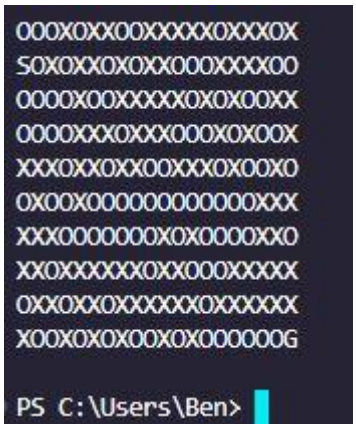


**Fig. 2.** *Locomotion mechanisms used in biological systems*

The maze implementation starts with defining the size of the maze. The user can create a custom size when running the program, however if no custom size is specified then the maze is set to 50x50 (as seen in Fig. 3).

**Fig. 3.** *Main function of Maze Generation*

```
def main():
    try:
        rows = int(sys.argv[1])
        cols = int(sys.argv[2])
        print("Generating {}x{} Maze".format(rows,cols))
        new_maze = Maze_Generator(rows, cols)
    except:
        print("Generating 50x50 maze")
        new_maze = Maze_Generator(50, 50)

    new_maze.generate_map_file("MAZE_MAP")
    new_maze.generate_pat_file("MAZE")
```

A python class (as shown in Fig. 4) was created to automate and model this maze generation process. It creates an unique maze each time of arbitrary length and width that can always be solved. The maze is stored as a two-dimensional array of characters so that it can be exported into file types PAT and Python can understand.

```
class Maze_Generator:
    def __init__(self, rows, cols) -> None:

        start_time = time.time()

        self.maze_size = (rows, cols)
        self.goal = self.set_goal(self.maze_size) #Generate goal position
        self.starting_position = self.set_start(self.maze_size) #Generate starting position

        self.maze_map = [["O" for i in range(cols)] for j in range(rows)] #generate empty Maze
        self.solution = None #Path used for solution

        self.generate_solution(self.starting_position, self.goal) #Find solution path between points
        self.create_walls(0.7) #Generate walls not in way of solution path
        self.hide_solution() #Hide solution path from maze

        end_time = time.time()
        print("Time Taken: ", format(end_time- start_time, ".5f"), " seconds")

        #print(self.convert_map_to_str(self.solution)) #Uncomment above to see solution in cmd line
        #print(self.convert_map_to_str(self.maze_map)) #Uncomment above to see maze in cmd line
```

**Fig. 4.** *Python Class for Generating Mazes*

To ensure an interesting maze structure, the goal position was generated randomly in the bottom right quadrant of the maze (as shown in Fig. 5), while the starting position was randomly generated in the top left quadrant of the maze (as shown in Fig. 5). Both the starting and goal positions are generated and stored as (x,y) tuples.

```
def set_goal(self, maze_size):
    #choose random point within a quadrent to be set as the goal
    x = random.randrange(int(maze_size[0]*0.75), int(maze_size[0]))
    y = random.randrange(int(maze_size[1]*0.65), int(maze_size[1]))
    return (x, y)

def set_start(self, maze_size):
    #choose random point within a quadrent to be set as the goal
    x = random.randrange(0, int(maze_size[0]*0.25))
    y = random.randrange(0, int(maze_size[1]*0.25))
    return (x, y)
```

**Fig. 5.** *Python Class for Generating Mazes*

A solution pathway was also needed to be created to ensure the maze was solvable. From the starting position, directional choices (Left, Right, Up, Down) were made with preference weighted towards the goal. To add an element of engagement, the movement direction was randomly scaled by a factor of 1 to 5. Once the goal was reached, this path was recorded and saved (as shown in Fig. 6). The remaining sections of the maze were then assigned with a 70% chance of becoming an obstacle, adding an additional layer of complexity to the maze layout.

```
Generating 10x20 Maze
Time Taken:  0.00000  seconds
--OOOOOOOOOOOOOOOOO
S-OOOOOOOOOOOOOOOOO
----OOOOOOOOOOOOOOO
-OO-OOOOOOOOOOOOOOO
OOO-OOOOOOOOOOOOOOO
OOO-OO----------OOO
OOO-----OOOOO----OOO
OOOOOOOOOOOOO-OOOOO
OOOOOOOOOOOOO-OOOOO
OOOOOOOOOOOOO------G
```

**Fig. 6.** *Solution Path for Maze*

Finally, the initially generated solution was removed and replaced by empty cells (as seen in Fig. 7), resulting in an unsolved state for the maze.

```
def hide_solution(self): # Checks every location of map and hides solution path
    for x, row in enumerate(self.maze_map):
        for y, point in enumerate(row):
            if point == "-":
                self.maze_map[x][y] = "O"
```

**Fig. 7.** *Function for hiding solution path*

A PAT CSP file was also generated each time a maze was created (as shown in Fig 8). This was so PAT could be provided with all the information, such as the maze dimensions, maze layout and start/goal positions, required to solve the maze. The Grid variable for the maze was initiated as a 1D array, so that PAT could handle the referencing of the distinct lines. Another method was attempted by creating a blank maze model then assigning all values individually using (maze[0][1] = W) referencing. This, however, occasionally crashed PAT with grid size greater than 100 and was proven to be less efficient than the above method.

```
def generate_pat_file(self, filename): #creats a .csp file for PAT to use
    pat_file = open(filename+".csp", "w") # open/create .csp file
    pat_file.write("#define N {}; //Row Size\n".format(self.maze_size[0])) # Define row size
    pat_file.write("#define M {}; //Column Size\n".format(self.maze_size[1])) # Define col size

    pat_file.write("var maze[N][M] = [\t")
    readable_maze_map = []
    for i in self.maze_map:
        readable_maze_map.append(', '.join(i)) # add comma & space between each point/char in the maze
    pat_file.write(",\n\t\t\t\t\t\t".join(readable_maze_map))
    pat_file.write(" ];\n\n")

    #Define starting position
    pat_file.write("var x = {}; //Row Starting position\n".format(self.starting_position[0]))
    pat_file.write("var y = {}; //Column Starting position\n".format(self.starting_position[1]))

    pat_file.close()
```

**Fig. 8.** *.csp File Generation*

```
1   #define N 10; //Row Size
2   #define M 20; //Column Size
3   var maze[N][M] = [  S, O, O, O, O, O, O, O, O, O, O, O, O, O, O, O, O, O, X, O,
4                       X, X, O, O, X, X, O, X, O, O, O, X, X, O, X, X, X, X, X, X,
5                       O, X, O, O, O, X, O, X, X, O, O, O, X, O, X, X, X, X, X, X,
6                       X, X, X, X, X, X, X, O, X, O, O, X, O, O, X, O, O, O, O, O,
7                       X, X, X, O, O, X, O, X, X, O, O, X, X, O, X, O, O, O, O, O,
8                       O, X, O, X, X, X, X, X, X, O, O, O, X, O, O, O, X, O, O, X,
9                       O, X, X, O, O, X, O, O, O, O, O, X, X, O, O, O, O, O, O, O,
10                      X, X, X, X, O, X, O, X, X, O, O, O, O, O, X, X, O, G, O, O,
11                      O, X, X, O, X, O, X, X, O, O, O, O, O, O, X, X, O, O, O, O,
12                      X, O, O, O, X, X, O, X, X, O, O, O, O, O, O, O, O, X, O ];
13
14  var x = 0; //Row Starting position
15  var y = 0; //Column Starting position
16
```

**Fig. 9.** *Example of .csp File produced*

**Controller:**

the action controller implemented in the Python script controls the movement of the robot based on actions read from the file. When started, the script initializes a ROS node named 'robot_action_node' and creates a publisher named 'velocity_publisher' that publishes Twist-type messages about '/cmd_vel' to control robot speed. The script also subscribes to the theme '/robot_position' to get information about the current position of the robot, and when it receives the message, calls the function 'position_callback'.

```
def position_callback(position):
    global robot_position_x, robot_position_y
    robot_position_x = int(position.x)
    robot_position_y = int(position.y)
```

**Fig. 10.** *position_callback function*

Then, the script enters a loop that runs until the ROS node is stopped. In each iteration of the loop, the script first reads the actions from the file using the 'read_actions' function, which allows the detection of possible changes in the file in real time. Then, for each action, the script requests sensor data from the service 'sensor_check' via the service client. If the query fails, the script records an error message and interrupts the current iteration.

```
def read_actions(filename):
    with open(filename, 'r') as file:
        return [line.strip() for line in file]
```

**Fig. 11.** *read_actions function*

```
    for action in actions:
        # Request sensor data
        try:
            sensor_check_client = rospy.ServiceProxy('sensor_check', SensorData)
            response = sensor_check_client()
            rospy.loginfo(f"Sensor data: left-{response.left}, right-{response.right}, up-{response.up},
down-{response.down}")
            sensor_data = {"left": response.left, "right": response.right, "up": response.up, "down":
response.down}
        except rospy.ServiceException as e:
            rospy.logerr(f"Service call failed: {e}")
            break
```

**Fig. 12.** *sesnsor_check function*

With the successful acquisition of sensor data, the script calls the function 'move_robot' with the current action and sensor data as arguments. The function checks whether an action is possible (in case, for example, there are no obstacles in the robot's path). If the motion is possible, the function sends a speed control message to move the robot. Otherwise, the function displays an obstacle message and returns False.

```
def move_robot(action, sensor_data):
    velocity_message = Twist()

    # Move the robot based on the action
    if action == "go_left":
        if sensor_data["left"] != 'X':
            print("left")
            velocity_message.linear.x = 1.0
            velocity_message.linear.y = 0.0
        else:
            print("obstacle on the left")
            return False
```

**Fig. 13.** *move_robot function (in the example only one direction is represented but for the rest the code looks the same )*

If the 'move_robot' function returns False, this implies that the movement in this direction is not possible due to an obstacle, and the current iteration is interrupted. The script calls the PAT function to recalculate the path. To adjust the frequency of actions, the script uses a Rate object with a frequency of 5 Hz, which means that each action is performed once in 5 seconds. At the end of the main block of code, the rospy.spin() function is called to wait for the ROS event and prevent the script from ending prematurely.

**Sensors:**

The sensor node is initialized with the name 'sensor_node'. After initialization, it reads static map data from 'map.txt' file as well as dynamic map data from 'dynamic_map.txt'. The map is a two-dimensional array, where each cell may contain a value indicating an obstacle or free space.

```
1 00000
2 00000
3 00000
4 00000
5 00000
```

**Fig. 14.** *initial state of dynamic map*

The node subscribes to the theme '/robot_position' to get information about the current position of the robot. When the node receives the position information, the 'position_callback' function is called. This function updates the global variables x and y, representing the current position of the robot, and obtains the information about the robot's environment, calling 'get_surroundings'. Then it stores the environment in dynamic map data and updates the data into a file.

```
def get_surroundings(map_data, x, y):
    # Get the surrounding cells of the given position in the map
    surroundings = {'left': None, 'right': None, 'up': None, 'down': None}
    if x is not None and y is not None:
        if x > 0:
            surroundings['left'] = map_data[y][x - 1]
        if x < len(map_data[0]) - 1:
            surroundings['right'] = map_data[y][x + 1]
        if y > 0:
            surroundings['up'] = map_data[y - 1][x]
        if y < len(map_data) - 1:
            surroundings['down'] = map_data[y + 1][x]

    return surroundings
```

**Fig. 15.** *get_surroundings function*

The sensor node also provides a service called 'sensor_check', which can be called to receive sensor data from the robot. When this service is called, the 'handle_sensor_check' function is performed, which simulates the process of reading sensor data by checking values around in a radius of one conventional unit and the current position of the robot in the static map data.

```
def handle_sensor_check(req):
    global dynamic_map_data, x, y

    # Here we simulate receiving data from the sensors
    surroundings = get_surroundings(dynamic_map_data, x, y)

    if surroundings['left'] is None:
        surroundings['left'] = 'X'
    if surroundings['right'] is None:
        surroundings['right'] = 'X'
    if surroundings['up'] is None:
        surroundings['up'] = 'X'
    if surroundings['down'] is None:
        surroundings['down'] = 'X'

    return surroundings['left'], surroundings['right'], surroundings['up'], surroundings['down']
```

**Fig. 16.** *handle_sensor_check function*

```
def update_map(map_data, x, y, surroundings):
    # Update the map data with the new surroundings
    if surroundings['left'] is not None and x > 0:
        map_data[y][x - 1] = surroundings['left']
    if surroundings['right'] is not None and x < len(map_data[0]) - 1:
        map_data[y][x + 1] = surroundings['right']
    if surroundings['up'] is not None and y < len(map_data) - 1:
        map_data[y + 1][x] = surroundings['up']
    if surroundings['down'] is not None and y > 0:
        map_data[y - 1][x] = surroundings['down']
```

**Fig. 17.** *update dynamic_map function*

In addition, the sensor node subscribes to the topic '/check_sensors', which is used to initiate sensor testing. When the host receives a message on this topic, the call callback function 'sensor_check_callback' is called. This function checks the robot's environment and publishes the sensor status (presence or absence of an obstacle) on a separate topic '/sensor_status'.

```
def sensor_check_callback(message):
    global dynamic_map_data, x, y
    rospy.loginfo("sensor_check_callback called")

    # Get the robot's surroundings
    surroundings = get_surroundings(dynamic_map_data, x, y)
    rospy.loginfo(f"Surroundings: {surroundings}")

    # Check for obstacles in the robot's surroundings
    for direction, cell in surroundings.items():
        if cell == 'X':
            rospy.logwarn(f"Obstacle detected in the {direction} direction.")
            has_obstacle = True
            break

    # Publish the sensor status
    rospy.loginfo(f"Publishing sensor status: {has_obstacle}")
    sensor_status_publisher.publish(Bool(has_obstacle))
```

**Fig. 18.** *sensor_check_callback function*

The primary purpose of this sensor node is to monitor the robot's environment and provide this information to other nodes or services through ROS message mechanisms and services. The node continues to work and process incoming messages until it is stopped or finished due to an error.

**Simulator:**

In the context of the pathfinding system, the simulator facilitates the robot navigation by providing a virtual maze environment for its sensors to detect. It serves as a proxy for a physical environment, enabling the pathfinding system to operate as it would in the real world, without requiring a physical setup. As such, the simulator offers a realistic representation of the way the proposed system would behave in reality. The simulator is implemented as a separate ROS node that handles translation of the generated map into a detectable and traversable environment for the pathfinding system.

The simulator provides a viewer of the robot and environment, allowing observation of the pathfinding system in real-time as it solves the maze. With this, qualitative validation of the system can be performed, both during development of the pathfinding algorithm and testing of its accuracy. To do so, the simulator has a MazeViewer class that receives height, width and grid_size variables from the MazeGenerator class above.

```
class MazeViewer(object):
    def __init__(self, width, height, grid_size):
        pygame.init()
        self.width = width
        self.height = height
        self.grid_size = grid_size
        self.screen = pygame.display.set_mode((width, height))
        self.clock = pygame.time.Clock()
        self.is_running = False
```

**Fig. 19.** *MazeViewer class*

It then translates this information into data provided to the sensors upon request, as well as draws the map to the viewer window.

```
def draw_map(self):
    window = self.screen
    for row in range(len(map)):
        for col in range(len(map[0])):
            x = col * self.grid_size
            y = row * self.grid_size

            if map[row][col] == "X":
                pygame.draw.rect(window, BLACK, (x, y, self.grid_size, self.grid_size))
            elif map[row][col] == "O":
                pygame.draw.rect(window, WHITE, (x, y, self.grid_size, self.grid_size))
            elif map[row][col] == "S":
                pygame.draw.rect(window, GREEN, (x, y, self.grid_size, self.grid_size))
            elif map[row][col] == "G":
                pygame.draw.rect(window, RED, (x, y, self.grid_size, self.grid_size))
```

**Fig. 20.** *map drawing function*

As each step in solving the maze is executed, the simulator is updated with the new position of the robot and the map is redrawn accordingly.

```
def get_robot_position(self, position_msg):
    # Callback function to receive the robot's position from publisher ROS node
    # the position message is of type geometry_msgs/Point

    # Extract x and y coordinates from the message
    x = position_msg.x
    y = position_msg.y

    # Convert coordinates to grid indices
    grid_x = int(x / self.grid_size)
    grid_y = int(y / self.grid_size)

    return grid_x, grid_y
```

**Fig. 21.** *update robots position in simulator function*

Once the map is redrawn, the simulator publishes data describing the environment surrounding the robot so the next step in the pathfinding system can execute.

```
def publish_sensor_information(self, robot_position):
    # Publishes sensor information about the robot's environment to a ROS topic
    x, y = robot_position

    # The sensor reads the state of the current grid square where the robot is located
    sensor_info = self.grid_pattern[y][x]

    # Publish sensor information to the "/sensor_data" topic
    sensor_pub.publish(sensor_info)
```

**Fig. 22.** *simulator map data publishing function*

Through this implementation model, the simulator and environment are separated from the pathfinding system, providing the sensors with data they would receive in a real-world environment. In other words, the pathfinding system is unaware that it is solving a maze in a simulated environment. Likewise, the simulator is separate from the maze generation and as such, can be used to translate different applications or programs into problems for the (robot) to solve.

**PAT:**

A .csp file was made for PAT to solve the task at hand.

```
#include "MAZE.csp";

//maze model:
#define S 0; // Starting Position
#define O 1; // Open pathway
#define G 2; // Goal
#define X -1;// Wall/Obstacle

MazeRunner =[x-1>=0]MoveUp
            [] [x+1<N]MoveDown
            [] [y-1>=0]MoveLeft
            [] [y+1<M]MoveRight;

MoveUp = [maze[x-1][y]!=X]go_up{x=x-1} -> MazeRunner;
MoveDown = [maze[x+1][y]!=X]go_down{x=x+1} -> MazeRunner;
MoveLeft = [maze[x][y-1]!=X]go_left{y=y-1} -> MazeRunner;
MoveRight = [maze[x][y+1]!=X]go_right{y=y+1} -> MazeRunner;


#define goal maze[x][y] == G;
#assert MazeRunner() reaches goal;
```

**Fig. 23.** *PAT Maze Solver*

## V. EXPERIMENTS

In this section, we present the results of implementing the proposed approach for creating a software robot capable of efficient pathfinding through a variety of mazes. The results demonstrated that the modules in the proposed approach functioned correctly, and the integrated system successfully achieved efficient pathfinding in the mazes. The modular system design allowed for rapid and comprehensive testing and optimization, the results of which are shown below.

```
********Verification Result********
The Assertion (MazeRunner() reaches goal) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_right -> go_down ->
go_down -> go_down -> go_down -> go_down -> go_down -> go_right -> go_right -> go_right -> go_down -> go_right>

********Verification Setting********
Admissible Behavior: All
Search Engine: Shortest Witness Trace using Breadth First Search
System Abstraction: False

********Verification Statistics********
Visited States:87
Total Transitions:209
Time Used:0.0048712s
Estimated Memory Used:9123.544KB
```

**Fig. 24.** *PAT Output*

The results from PAT show us the direction for the shortest path solution using Breadth First Search. The integration with the PAT framework in the PAT Interaction Module facilitated high-level abstract planning and goal reasoning, finding the optimal solution through the maze and instructing the robot accordingly.

```
[INFO] [1687087793.391340]: Node started
[INFO] [1687087793.396857]: Sensor data: left-X, right-X, up-X, down-O
down
[INFO] [1687087793.400601]: Position successfully updated to (2, 1)
[INFO] [1687087794.400388]: Sensor data: left-X, right-X, up-O, down-S
down
[INFO] [1687087794.404434]: Position successfully updated to (2, 2)
[INFO] [1687087795.447157]: Sensor data: left-O, right-X, up-X, down-O
down
[INFO] [1687087795.451365]: Position successfully updated to (2, 3)
[INFO] [1687087796.396251]: Sensor data: left-O, right-X, up-O, down-O
left
[INFO] [1687087796.399872]: Position successfully updated to (1, 3)
[INFO] [1687087797.396613]: Sensor data: left-X, right-X, up-O, down-O
down
[INFO] [1687087797.399410]: Position successfully updated to (1, 4)
[INFO] [1687087798.400817]: Sensor data: left-O, right-O, up-O, down-X
right
[INFO] [1687087798.404415]: Position successfully updated to (2, 4)
[INFO] [1687087799.411350]: Sensor data: left-O, right-G, up-X, down-X
right
[INFO] [1687087799.415021]: Position successfully updated to (3, 4)
```

**Fig. 25.** *screenshot of sensor and controller proceeding*

The screenshot above shows the operation of the sensor and controller. The sensor reads the robot's surroundings then transmits the signal to the controller, which updates the robot's position.

```
[INFO] [1687087793.275276]: Published position: (2, 0)
[INFO] [1687087793.399513]: Updated position received: (2, 1)
[INFO] [1687087794.284321]: Published position: (2, 1)
[INFO] [1687087794.403209]: Updated position received: (2, 2)
[INFO] [1687087795.275317]: Published position: (2, 2)
[INFO] [1687087795.450190]: Updated position received: (2, 3)
[INFO] [1687087796.277443]: Published position: (2, 3)
[INFO] [1687087796.398719]: Updated position received: (1, 3)
[INFO] [1687087797.303089]: Published position: (1, 3)
[INFO] [1687087797.398535]: Updated position received: (1, 4)
[INFO] [1687087798.282938]: Published position: (1, 4)
[INFO] [1687087798.403301]: Updated position received: (2, 4)
[INFO] [1687087799.275395]: Published position: (2, 4)
[INFO] [1687087799.413867]: Updated position received: (3, 4)
```

**Fig. 26.** *screenshot of the postition_publisher proceeding*

The screenshot above shows the operation of position_publisher, which updates and saves the position of the robot.

The Simulator Node Module provided real-time observation and evaluation of the robot's progress in the maze by simulating the effect of executing the high-level plan from PAT and publishing environment data for the sensor node to read.

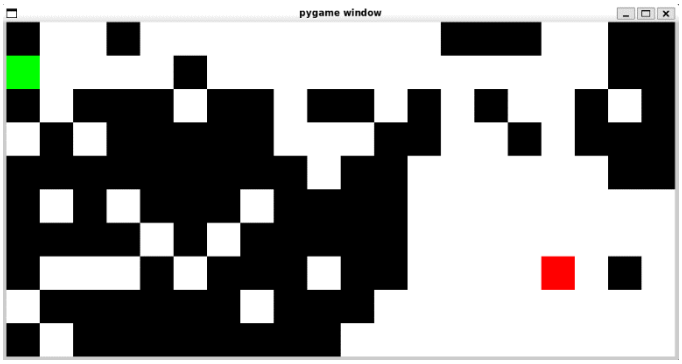


**Fig. 27.** *simulator window of the map, including start (green) and goal (red)*

The screenshot above shows the simulator window with the viewer of the map, containing the start/current position of the robot (green) and the goal (red).

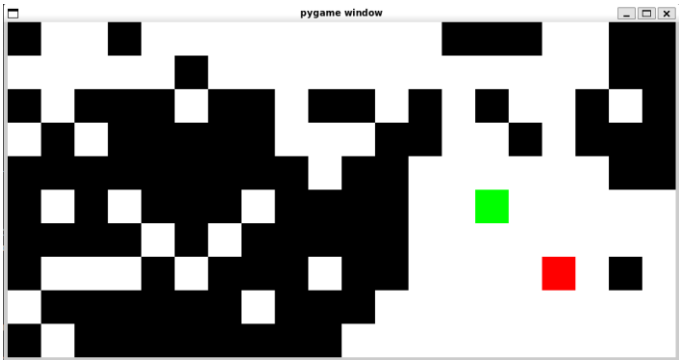


**Fig. 28.** *simulator window of the map, including robot (green) and goal (red)*

The position of the robot in the simulator is updated as it executes instructions based on the information from PAT, navigating the maze towards the goal.

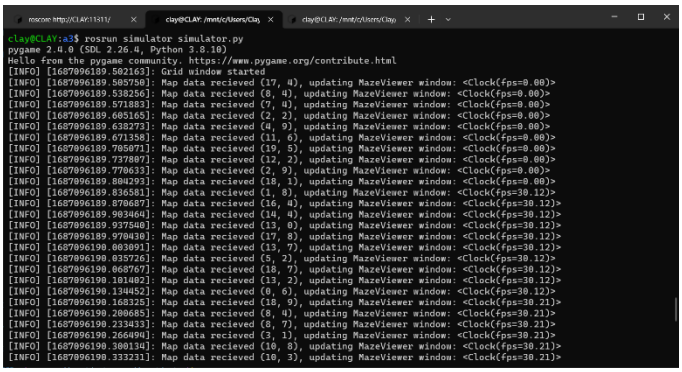


**Fig. 29.** *simulator receiving map data and updating the robot position*

As the robot moves throughout the maze, the simulator receives positional information and updates the viewer accordingly. In doing so, it also updates the environment data it publishes for the sensor.

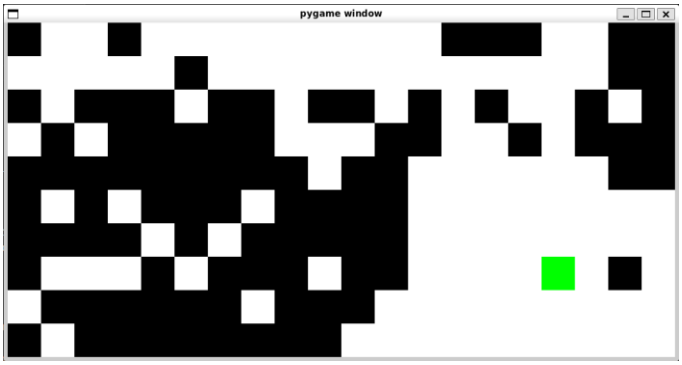


**Fig. 30.** *simulator viewer depicting the robot successfully reaching the goal point*

The robot successfully reaches the goal position of the maze and the simulation terminates.

## VI. CONCLUSION

The main issue described in this work is the system we have created for autonomous system decision-making. Our system includes not only a torus basis for hierarchical target planning but also focuses on implementation in practice with the help of the PAT model tester and ROS application community. The basis of our structure is ROS on the Linux operating system. Modeling our system has inferred that the PAT model tester is sufficient for high-level decision-making tasks. In addition, we have developed several support functions that facilitate the implementation of the PAT high-level plan for low-level activation plans. Further improvement of the project can help us with solving the problems of localization of an autonomous mobile robot and mapping. In this way, an autonomous robot can find itself in space and reach its destination in a dynamic environment.

## VII. CONTRIBUTIONS

Bennett Taylor – Abstract, Introduction, Related Work, Proposed Approach, Implementation (maze generation)

Denis Chizhov – Implementation (controller, Sensors), Results, Conclusion

Clayton Johnson – Implementation (simulator), Results, Conclusion

## VIII. REFERENCES

[1] J. H. Munson, “Robot planning, execution, and monitoring in an uncertain environment,” in IJCAI, 1971.

[2] T. Nagata, M. Yamazaki, and M. Tsukamoto, “Robot planning system based on problem solvers,” in Proceedings of the 3rd International Joint Conference on Artificial Intelligence, ser. IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, p. 388–395

[3] M. P. Georgeff and A. L. Lansky, “Reactive reasoning and planning.” in AAAI, vol. 87, 1987, pp. 677–682.

[4] K. Z. Haigh and M. M. Veloso, “Interleaving planning and robot execution for asynchronous user requests,” in Autonomous agents. Springer, 1998, pp. 79–95.

[5] R. Fikes and N. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” in IJCAI, 1971.

[6] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. S. Weld, and D. Wilkins, PDDL-the planning domain definition language, AIPS-98 Planning Competition Committee, 1998.

[7] M. Vallati, L. Chrpa, M. Grzes, T. L. McCluskey, M. Roberts, S. Sanner, ´ and M. Editor, “The 2014 international planning competition: Progress and trends,” AI Magazine, vol. 36, no. 3, pp. 90–98, Sep. 2015. [Online]

[8] A. Coles, A. Coles, M. Fox, and D. Long, “Forward-chaining partial-order planning,” in ICAPS’10: Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling, 2010. [Online]. Available: https://www.researchgate.net/publication/220936392_Forward-Chaining_Partial-Order_Planning

[9] P. Eyerich, R. Mattmuller, and G. R ¨ oger, ¨ Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 49–64. [Online]. Available: https://www.researchgate.net/publication/220936158_Using_the_Context-Enhanced_Additive_Heuristic_for_Temporal_and_Numeric_Planning

[10] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, “Rosplan: Planning in the robot operating system,” in ICAPS’10: Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling, 2015. [Online]. Available:

https://www.researchgate.net/publication/283799217_Rosplan_Planning_in_the_robot_operating_system

[11] D. S. S. Miranda, L. E. de Souza, and G. S. Bastos, “A rosplanbased multi-robot navigation system,” 2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE), pp. 248–253, 2018.

[12] F. Mart´ın, J. Gines, D. Vargas, F. J. Rodr ´ ´ıguez-Lera, and V. Matellan, ´ “Planning topological navigation for complex indoor environments,” in 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2018, pp. 1–9.

[13] V. Nogueira and L. Edival de Souza, “Uav mission planning and execution via non-deterministic ai planning on ros,” in 2018, XXII Congresso Brasileiro de Automatica ´ , 2018.

[14] N. Palomeras, A. Carrera, N. Hurtos, G. C. Karras, C. P. ´ Bechlioulis, M. Cashmore, D. Magazzeni, D. Long, M. Fox, K. J. Kyriakopoulos, P. Kormushev, J. Salvi, and M. Carreras, “Toward persistent autonomous intervention in a subsea panel,” Auton. Robots, vol. 40, no. 7, p. 1279–1306, Oct. 20

[15] N. J. Nilsson, “Shakey the robot,” AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Tech. Rep. 323, Apr 1984.

[16] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” in Proceedings of the 2nd International Joint Conference on Artificial Intelligence, ser. IJCAI’71. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1971, p. 608–620.

[17] W. Burgard, A. B. Cremers, D. Fox, D. Hahnel, G. Lakemeyer, ¨ D. Schulz, W. Steiner, and S. Thrun, “The museum tour-guide robot rhino,” in Autonome Mobile Systeme 1998, 14. Fachgesprach ¨ . Berlin, Heidelberg: Springer-Verlag, 1988, p. 245–254.

[18] I. Nourbakhsh, R. Powers, and S. Birchfield, “Dervish an officenavigating robot,” AI Magazine, vol. 16, no. 2, p. 53, Jun. 1995. [Online]. Available: https://aaai.org/ojs/index.php/aimagazine/article/view/1133

[19] R. C. Arkin, “Motor schema — based mobile robot navigation,” The International Journal of Robotics Research, vol. 8, no. 4, pp. 92–112, 1989. [Online]. Available: https://doi.org/10.1177/027836498900800406

[20] R. A. Brooks, “Intelligence without reason,” in Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1, ser. IJCAI’91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, p. 569–595.

[21] P. Bustos Garc´ıa, L. Manso Arguelles, A. Bandera, J. Bandera, ¨ I. Garc´ıa-Varea, and J. Mart´ınez-Gomez, “The cortex cognitive robotics ´ architecture: Use cases,” Cognitive Systems Research, vol. 55, pp. 107 – 123, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389041717300347