A picture containing text, outdoor, sign

Description automatically generated

Software Engineering Department

Braude College of Engineering

Capstone Project Phase B

**Path Planning Via BMC of Hyper-Properties**

**23-2-R-5**

Students:

|  |  |  |
| --- | --- | --- |
| daniel.bal0797@gmail.com | 318214673 | Daniel Balaish |
| lironlavi20@gmail.com | 206230831 | Liron Lavi |

Supervisor:

**Dr. Sarai Sheinvald**

Link to the project’s GitHub repository:

[GitHub - BMC Path Planning](https://github.com/danielbal21/BMC_Path_Planning)

**Table of Contents**

[1.Introduction 3](#_Toc164373366)

[2. Background and Related Work 7](#_Toc164373367)

[2.1 Kripke Structures 7](#_Toc164373368)

[2.2 SAT Problem 8](#_Toc164373369)

[2.3 SAT Solver 8](#_Toc164373370)

[3. Expected achievements. 8](#_Toc164373371)

[4. The Solution 9](#_Toc164373372)

[4.1 learning stage 9](#_Toc164373373)

[4.2 System Generator 10](#_Toc164373374)

[4.3 Flattening the Counteragents 12](#_Toc164373375)

[4.4 Algorithm 14](#_Toc164373376)

[4.5 Formula Creation Process 15](#_Toc164373377)

[4.6 Software Development 18](#_Toc164373378)

[5. Analysis and Benchmarking 26](#_Toc164373379)

[6. Project Reflection 32](#_Toc164373380)

[7. User Manual 34](#_Toc164373381)

[8. Maintenance Guide 37](#_Toc164373382)

[9. References 38](#_Toc164373383)

**Abstract**

*Model Checking* is a collection of algorithms used to explore a state space of a system to determine if it obeys a specification of its intended behavior. *Bounded Model Checking (BMC)* is a technique used to iteratively explore the state space of a system, exploring all possible states up to a certain bound[2], in search for a counterexample, which is a system run that violates a required specification. *hyper-properties* are used to describe the systems combined behavior which is intersection of agents in our case. This project presents an application of BMC over hyper-properties to find a path for an agent which does not intersect with dynamic counteragents on a grid. This is achieved translating the systems (agent and counter-agents systems) into Boolean Formulas and using *SAT-Solvers* for finding a path if exists.

In this project we will present the algorithm, data structures, software design and implementation while providing elaborate explanations of the processes, tools and evaluation plans.

# 1. Introduction

Pathfinding on a grid with moving agents is a problem of determining whether there exists a valid path between two points on a grid-based map.[6]

The pathfinding process is done on a 2D grid-based environment, the grid size can vary. Multiple agents are moving on the grid in an indeterministic fashion.

There is one agent (main agent) which always starts in point and its goal is to reach point , in our case is the bottom left and is the top right of the grid.

This problem is often encountered in scenarios where multiple entities, such as robots, vehicles, or characters in a video game, need to navigate a shared environment without colliding with each other.

The pathfinding algorithm needs to consider the dynamic nature of the agents' movements, anticipating their positions and avoiding them accordingly. It involves analyzing the current and predicted positions of the agents and calculating a path that avoids potential collisions or intersections. Common algorithms may need to continuously update the path as the positions of the agents change over time.

The objective is to determine if a feasible path exists, rather than finding the shortest or most optimal path. The focus is on avoiding the agents and ensuring safe navigation rather than minimizing distance or cost.

In our project the environment is a von Neumann grid; A von Neumann grid is a 2D grid structure where each cell is connected to its four neighboring cells: up, down, left, and right. It is named after the mathematician John von Neumann. The von Neumann grid offers simpler connectivity compared to the more intricate Moore grid, as it allows movement or connection in only four cardinal directions.  
however, a fifth direction which is to stay in place is also possible for the agent.

For BMC to be applied into pathfinding problems, the problem needs to be modeled in a way that can eventually be represented as a *satisfiability problem*.[1]

Given a grid the size of ( a main agent and a counter-agent, we will describe the main agent and the counter-agents behavior by using *Kripke structures* and respectively.  
Each path of can be represented by a sequence of states and transitions from the initial state to a final state (if exists).

Each state of and is an image of the current position of all agents in the system on the grid.

The transitions describe possible future images that can exist at the next iteration.

a common way to answer the question “is there a path of which does not intersect with is to explore all possible paths of and all possible paths of however this solution is highly inefficient because it involves scanning all possible paths of two systems.

Another approach is to ask the question “Is there a path of such that every path of does not intersect with it” – this approach can be modeled into a satiability problem, for example, consider the following scenario (figure 1).

The green robot can move up or right, one step in every iteration while the red robot moves vertically on an infinite course, one step at the time, The green robot seeks to reach the top right corner (figure 1).

This can be modeled by using Kripke Structures (figure 2).

Graphical user interface

Description automatically generated with medium confidence

Figure 1 – The Idea

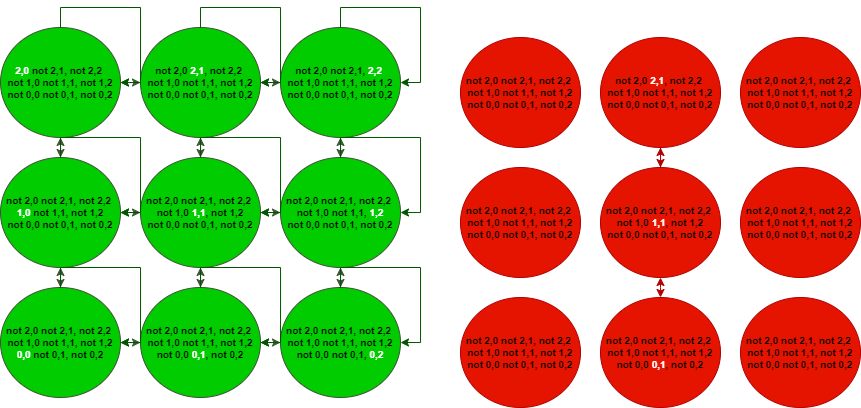
**

Figure 2 – Complete Kripke Structures

Where for each state the agent is either in the coordinates or not (green system) and for each counteragent the is not negated if the counter agent is in the position (red system), these forms of structures can be converted into Boolean formulas which then can be used with BMC in an attempt to find a safe path.

In this project we are going to implement the system’s creation process, translate the systems into Boolean formulas, feed them to a SAT-Solver and finally running experiments on various systems that differ in size and agent count in order to evaluate the algorithm efficiency and limitations.

**Boolean Formula**a Boolean formula is a finite string which is constructed from:

* Variables:
* Boolean operators:
* Parenthesis:

Each of the variables in a formula can have the value **true** or **false**  
a formula is the result **true** or **false**  under the *assignment*

For example, let us consider the formulaand the assignments  
 ,   
the value of under can be written as and is evaluated to  
 but

Boolean formulas are used to show Satisfiability, Unsatisfiability, Validity, Contradiction and Tautology:  
Let be a Boolean formula.

- is *valid* for every assignment is evaluated to **true** or **false.**

example:

- is *satisfiable* there exists an assignment which is evaluated to **true.**

example:

- is *unsatisfiable (contradiction)* for every assignment is evaluated to **false.**

example:

- is a tautology for every assignment is evaluated to **true**. example:

# 2. Background and Related Work

## 2.1 Kripke Structures

A *Kripke Structure* is a variation of a transition system, used in model checking to represent the behavior of a system.  
It consists of a graph where each node represents a state of the system, and the edges represent state transitions.  
Each node in the structure holds a set of properties.[4]  
**Definition 1.**  Let AP be a set of atomic propositions representing the properties. A Kripke Structure  over  is a tuple  , where is a set of *states*,  is a set of *initial states*,  is the *transition relation* of the structure and  is the labeling function.  
**Definition 2.** A *path* in a Kripke Structure  is an infinite sequence of states  such that  and  for all .  
 for each node in the sequence, there must be a transition from to .

**Definition 3.** The *Language*  of a Kripke Structure  is the set of all possible paths.

For example, let us consider the system:  
   
a path of the system is ( and are a tuple in the relation)

**Motivation**To further clarify why we should use a Kripke structure we will consider the following analogy:   
Let's define a counter agent system placed on an grid. Each step in the system is triggered by pressing a "step button." Upon each button press, all the counter agents in the system make a move.

To capture the system's behavior, we use a camera to take a picture after each button press. However, due to the indeterminism of the agents' movements, each agent can potentially move in multiple directions. This results in multiple possible pictures for each button press.

We associate each picture with the picture it originated from, forming a connection between states in the system. This collection of connected states and their associated pictures forms a Kripke structure that accurately describes the system’s behavior.  
By exploring the "gallery" of pictures connected through their originators, we can identify and create a safe path within the counter agent system. This path represents a sequence of states that guarantees the absence of collisions or unsafe situations among the agents.

## 2.2 SAT Problem

The *Boolean satisfiability problem* (SAT) is a problem of finding an assignmentto a set of Boolean variables such that a Boolean formula  will have the value ‘True’ under the assignments Given that SAT is the first problem that was proved to be [NP-complete](https://en.wikipedia.org/wiki/NP-complete), completing such a task is not straightforward, since there isn’t any known polynomial algorithm, that for a given SAT formula can determine whether its satisfiable or not. [5]

## 2.3 SAT Solver

*SAT Solvers* are programs which aim to solve the SAT problem. For a given Boolean formula , a SAT Solver outputs whether  is satisfiable, and if so, return the set of Boolean variables that satisfies . Since SAT is NP-C the solvers improve runtime by using sophisticated heuristics that enable them to solve very large Boolean formulas. Most SAT solvers include time outs, so they will stop attempting to solve the problem if they haven’t found a solution in a reasonable time and will output ‘unknown’[5]. For example, consider the following two Boolean formulas:  ,   
Giving that  is satisfiable, feeding it to a SAT solver would be resulted in an output such as:   
But, feeding  to a SAT solver would be resulted in the output: .

# 3. Expected achievements.

In our project, we expect achieve a working efficient path-finding algorithm on a grid involving a primary agent going from start to end while avoiding moving counteragents, the algorithm is based on BMC using encoded Kripke Structures generated from the agent and counteragents systems.

We also expect to generate an operational GUI with a visualization layer  
and various general abilities (System generation, import, export, benchmarking, and analyzers).  
another part of our project involves benchmarking the algorithm for various input systems of different size and complexity and test the algorithm limit in providing a solution in reasonable time.

# 4. The Solution

4.1 learning stage (Processing Pipeline)

The processing pipeline consists of five stages.

1. **Input** – At the outset, the pipeline accepts input in the form of system specifications, including grid dimensions, robot configurations, and any relevant parameters necessary for subsequent stages.
2. **Grid Cells and Robots** – In this phase, the system's architecture is abstracted into a structured representation, delineating grid cells and identifying the presence or absence of robots within each cell. This step establishes the foundational framework upon which subsequent analyses will be built.
3. **Flattening** – During flattening, the evolving dynamics of the system are distilled into a temporal sequence of snapshots, each capturing the state of grid occupancy at a specific point in time. This process facilitates the temporal abstraction of the system, essential for modeling its temporal behavior.
4. **Kripke Structure** – Transitioning to a more formal representation, the pipeline constructs a Kripke structure to encapsulate the system's evolution over time. Here, nodes correspond to discrete time points, while transitions between nodes capture the temporal progression of the system. This structural transformation enables rigorous analysis of temporal properties and behavior.
5. **Boolean Formula** – Finally, the system's dynamics, constraints, and objectives are encoded into a logical Boolean formula. By expressing system properties in a formal logic framework, the pipeline facilitates the application of automated solvers to ascertain solutions that meet specified criteria. This logical formulation provides a powerful means to explore, analyze, and reason about the system's behavior and potential outcomes.



Figure 3 – Processing Pipeline

## 4.2 System Generator

The automatic generation of a system involves creating a grid environment populated with robots (counteragents) that move according to certain parameters.

*Grid Size*  - This parameter defines the size of the grid environment, specifically an grid. It determines the dimensions of the space in which the robots will navigate.

*Number of Counter Agents* - This parameter specifies the number of robots (counteragents) to be generated in the grid environment. It controls the density of robots within the space and affects the complexity of the pathfinding problem.

*Stay Chance\** - This parameter represents the probability that a robot will choose to stay in place rather than move to an adjacent grid cell during each time step. It influences the likelihood of robots remaining stationary, adding variability to their movement behavior.

*Stray Radius* - This parameter defines the maximum distance that a robot can move away from its initial position in any direction. It constrains the extent to which robots can wander from their starting locations, affecting the overall spread of robot positions within the grid.

\* The *Stay Chance* parameter is critical as it dictates whether a counteragent remains stationary in a grid cell or moves. This decision has far-reaching consequences as staying in a cell permanently blocks it and all potential paths (potentially) originating from it, significantly impacting the grid's accessibility and path-solving feasibility.

The counteragent system generator utilizes an algorithmic framework, integrating probabilistic decision-making and recursive traversal methodologies to emulate the behavior of counteragents within a grid-based environment. Initially, the algorithm randomizes the placement of the counteragent within the grid, introducing stochasticity to its initial position and enhancing the diversity of generated scenarios.

Subsequently, at each stage of traversal, the counteragent evaluates potential movements based on predefined probabilities. Notably, these probabilities, except for the decision to remain stationary, are uniformly distributed at 50%, ensuring equitable exploration of directional options. This balanced approach contributes to a more nuanced representation of counteragent behavior, facilitating the generation of diverse movement patterns.

Upon selecting a movement direction, the counteragent calculates its trajectory from the new position, accounting for factors such as the distance traveled from the origin. Should the calculated travel distance exceed a specified stray radius, the counteragent is compelled to cease movement and maintain its current location.

Furthermore, the algorithm dynamically adapts to evolving scenarios, The detection of revisited grid cells serves as a crucial indicator of path completion, enabling the identification of closed loops and the evaluation of path efficiency.

The outcome of this generation process yields a refined input format, aligning with the subsequent stage of *Grid Cells and Robots* in the processing pipeline.

## 4.3 Flattening the Counteragents

The Flattening algorithm is an essential intermediary step in transforming the grid-based representation of a counteragent system into a Kripke structure. This algorithm operates by constructing a timeline where each step corresponds to a matrix representing all potential occupancies of counteragents at that moment in time.

The Flattening algorithm serves a critical purpose in streamlining the representation of a counteragent system into a more manageable form. Without this algorithm, constructing a Kripke structure that considers each counter-agent path individually would lead to an explosion in the number of atomic propositions, significantly lengthening the boolean formula. Additionally, the complexity arising from analyzing each counteragent and creating infinite realities where each one can move freely introduces significant computational burdens. By flattening the grid from multi-counteragent configurations to single-occupancy matrices, the algorithm sacrifices individuality for simplicity, resulting in a more tractable model.[3]

At the onset of the algorithm, an initial node is established to kickstart the timeline. This initial node encapsulates the starting positions of all counteragents within the system. Each robot's position is recorded, and a record of the robot positions is maintained for subsequent steps.

The algorithm iterates through successive steps in the timeline until a closed loop is detected, signifying the completion of the process. During each iteration, the potential whereabouts of each robot in the next step are calculated based on their current positions and movement capabilities. These potential positions are examined to determine the possibility of movement in each direction.

The algorithm's convergence is essential due to a key constraint: each counteragent must demonstrate cyclic behavior, represented by closed loops in their movements. As a result, the algorithm strives to identify a cyclic timeline encompassing all potential cycles for each counteragent, marking the completion of the process. The algorithm's convergence implies that it will eventually reach a state where it finds a movement pattern for all counteragents that has been encountered previously, thereby concluding the process. This convergence ensures that the algorithm exhaustively explores all possible movement scenarios for each counteragent.

If a closed loop is detected at any point during the timeline, the algorithm concludes, and the Kripke structure is finalized. Otherwise, the process continues, with new nodes being added to the structure to represent each successive step in the timeline. Relations between nodes are established to denote the transition from one step to the next.

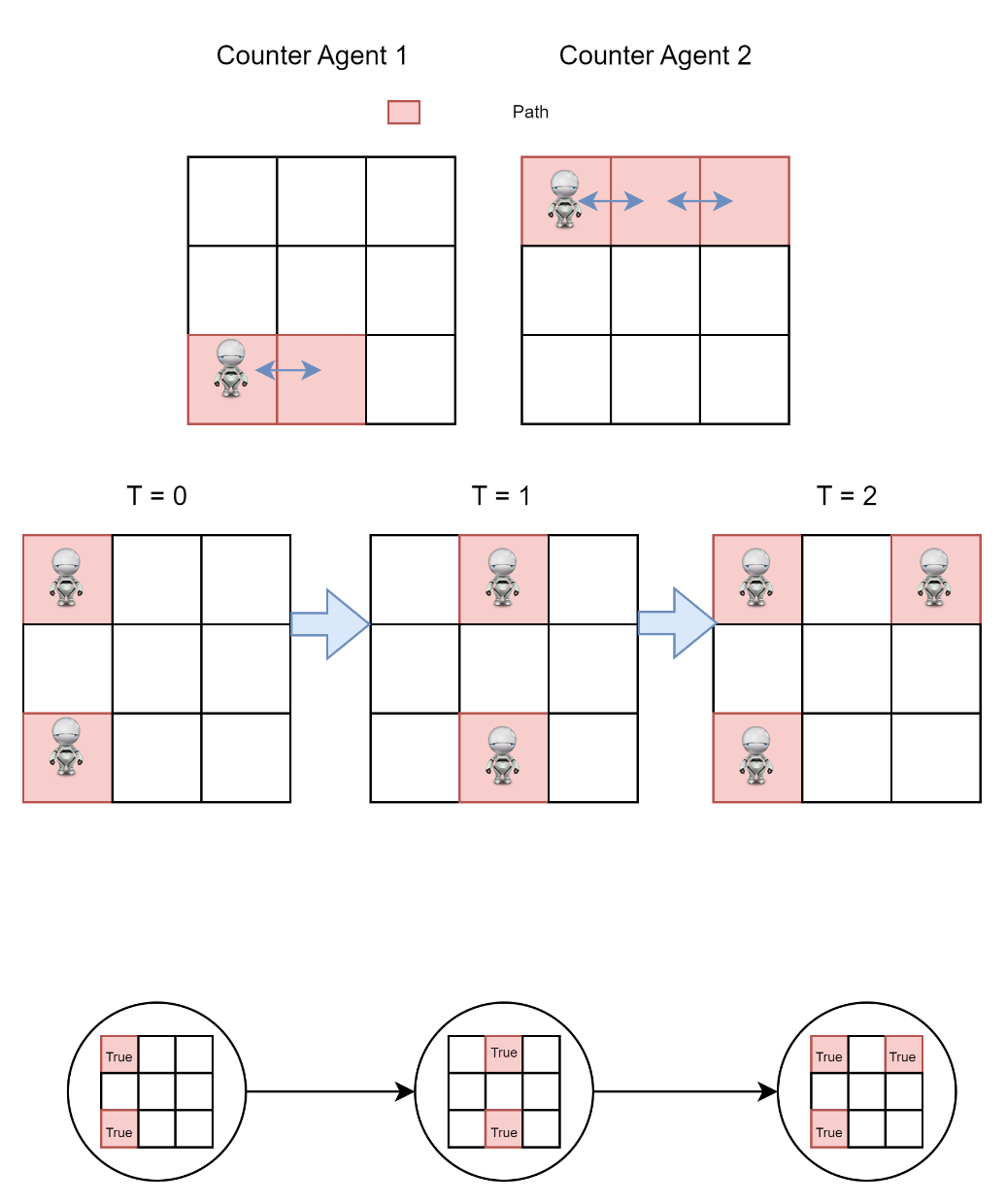


Figure 4 – Flattening Process

## 4.4 Algorithm

The algorithm aims to find a valid path for a robot moving on a grid represented by a Kripke structure, ensuring that it avoids collision with obstacles represented by another system (M2).

**Note that this explanation begins from the Kripke structure pipeline stage.**

The grid is represented as a set of nodes in a Kripke structure (M1), where each node represents a state of the grid.

Each state contains a Boolean matrix representing the *occupancy* of each cell in the grid. A True value indicates the presence of an obstacle or the robot itself in that cell, while False indicates an empty cell.

The movement of the robot is represented by transitioning between states in the Kripke structure. Each transition corresponds to moving the robot to an adjacent cell in the grid, including staying in the same cell.

**Constraints for Valid Movement:**The algorithm ensures that the robot moves in a valid manner by imposing constraints on its movement.Constraints are defined to allow the robot to move only to adjacent cells that are not occupied by obstacles or the robot itself. This ensures collision avoidance.Additional constraints are applied to ensure that the robot stays within the bounds of the grid.

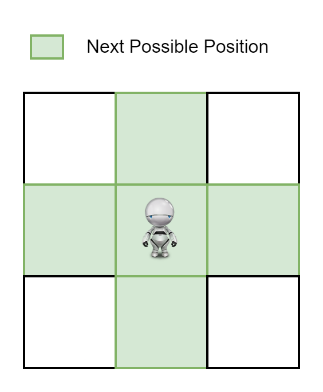


Figure 5 – Valid Movement Constraint

**Constraints for Path Finding:**

The algorithm constructs a path for the robot by defining constraints that enforce valid movement from the initial state to the final state.

Constraints are formulated to ensure that there exists a path from the initial state to the final state while satisfying the movement constraints defined earlier.

The path is constructed using Boolean variables to represent the movement of the robot at each time step.

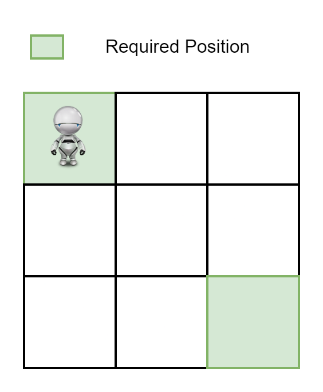


Figure 6 – Path Finding Constraint

**Safety Constraints:**To ensure safety, constraints are defined to prevent the robot from colliding with obstacles represented by the second system (M2).

These constraints ensure that at each time step, the robot does not occupy cells that are occupied by obstacles.

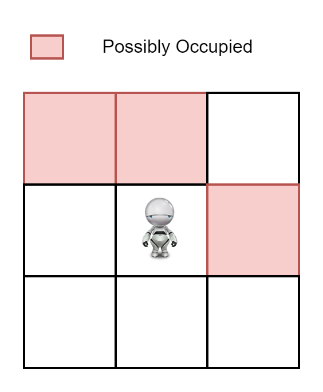


Figure 7 – Safety Constraints

**Solving the Constraints:**The formulated constraints are provided to a solver (such as Z3) to find a solution that satisfies all constraints.

If a solution is found, it represents a valid path for the robot to traverse from the initial state to the final state without colliding with obstacles.  
If no solution is found, it indicates that no valid path exists, considering the defined constraints.

4.5 Formula Creation ProcessThe solution involves formulating the problem as a set of logical constraints using Boolean variables. This process ensures that we capture the essential requirements for finding a valid path for the good robot while avoiding collisions with bad robots and staying within the grid boundaries. Each constraint corresponds to a specific condition that the solution must satisfy, and these constraints are combined using logical operators to form a comprehensive problem formulation.

The base boolean variables, denoted as represents the presence or absence of the good robot in each cell of the grid at each time step and represents the row and column respectively. These variables are the building blocks of our problem formulation and are used to encode the state of the system at each point in time.

The first formula *Initial Position* ensures that the good robot starts at the initial position which is fixed to .

The second formula *Final Position* ensures that the good robot ends at the final position which is fixed to where is the grid size and is path length.

The third formula *Valid Steps* ensures that the good robot can only move to neighboring cells or stay in its current cell at each time step, thus defining valid movements options.

\* Note that there is a grid boundary checking that is not shown in for simplicity concerns.

The fourth formula *Safety* ( ensures that the good robot cannot occupy cells with bad robots at any time step, thereby ensuring safety.

\* Where is the bad robot system occupancy indicator at

The fifth formula *Single Path* ensures that the good robot follows a single path by allowing at most one occupied cell per time step, preventing multiple simultaneous movements.

The overall boolean formula to represent the problem is the conjunction (AND) of all the individual constraints:

This formula represents the complete set of constraints necessary to ensure the existence of a valid path for the good robot from the initial state to the final state, while avoiding collisions with bad robots and adhering to the rules of movement within the grid.

Reconstructing the path involves deriving the sequence of grid states traversed by the good robot from the initial to the final state, based on the satisfying assignment acquired from solving the boolean formula.

Let represent the satisfying assignment obtained from the solver, where denotes the Boolean value assigned to the variable at time step and grid position

To reconstruct the path, we iterate through the satisfying assignment and trace the movement of the robot. Starting from the initial state we follow the sequence of transitions dictated by the satisfying assignment:

Here, denotes the row and column indices of the grid position at time step , as dictated by the satisfying assignment .

By adhering to this process, we systematically reconstruct the trajectory of the good robot, affirming its adherence to the permissible movements delineated in the satisfying assignment while navigating from the initial to the final state of the grid.

## 4.6 Software Development

The software architecture is structured into three distinct layers: Models, Services, and UI. The UI layer exclusively encompasses the user interface elements, facilitating interaction with the end user. This layer communicates with the Services layer, which is responsible for executing backend operations and processing data. The exchange of information between the Services layer and the UI layer occurs through the utilization of data types defined within the Models layer.

The analyzer is an external component, hooking into the backend services and providing automation for analyzing and testing the algorithm.

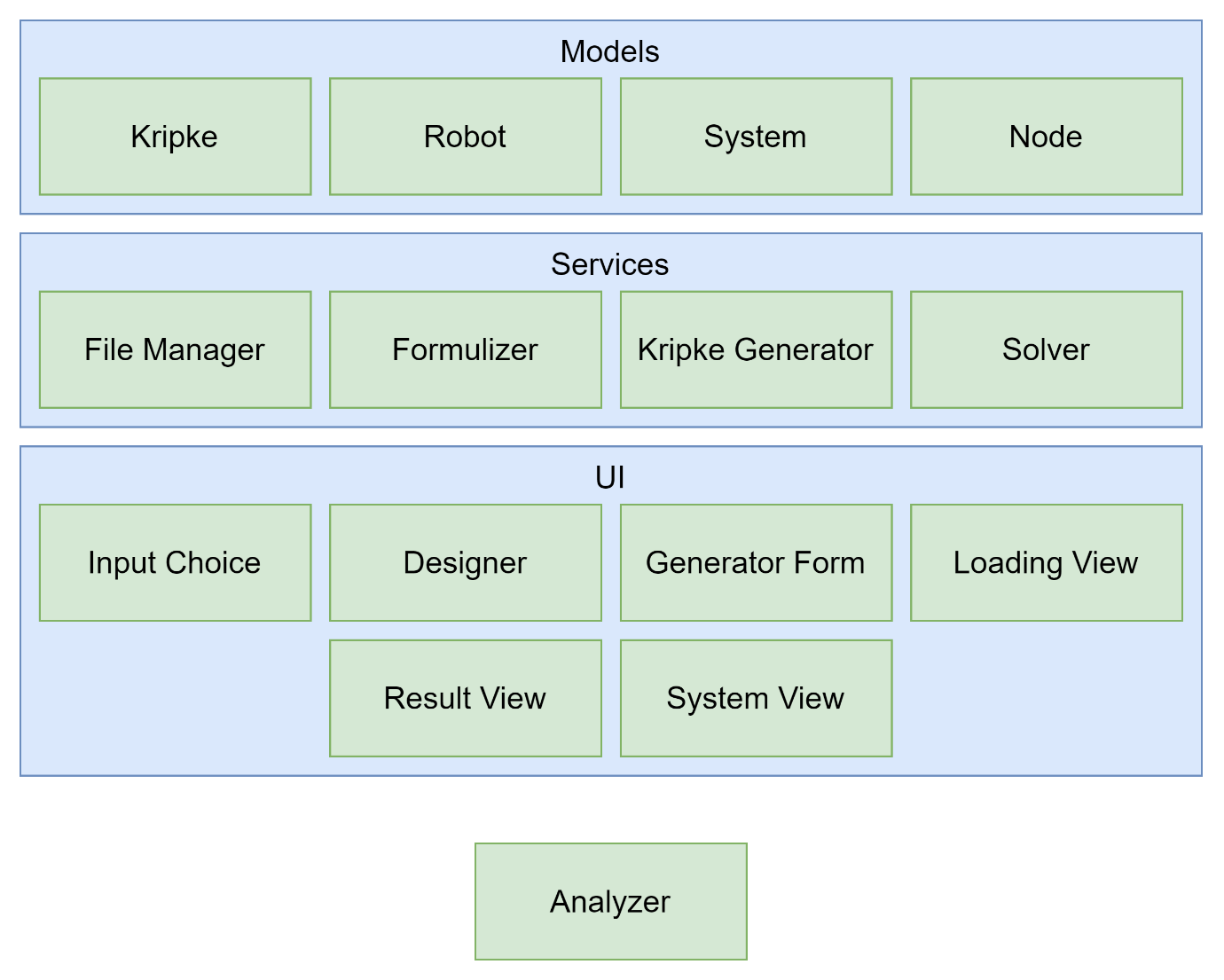


Figure 8 – Top Level Architecture

**User Interface**

input choice refers to the primary menu interface where users select their preferred method of loading the system.

A screenshot of a computer

Description automatically generated

Figure 9 – Input Choice Menu

The Designer tool allows users to craft systems interactively, enabling them to define grid dimensions, incorporate counteragents, and designate available movements (up, down, left, right, stay) for each grid cell. Additionally, users can specify the initial cell for each counteragent. As users finalize their grid designs, the Designer tool conducts checks to identify any undefined cells or open loops, ensuring the completeness and coherence of the system.

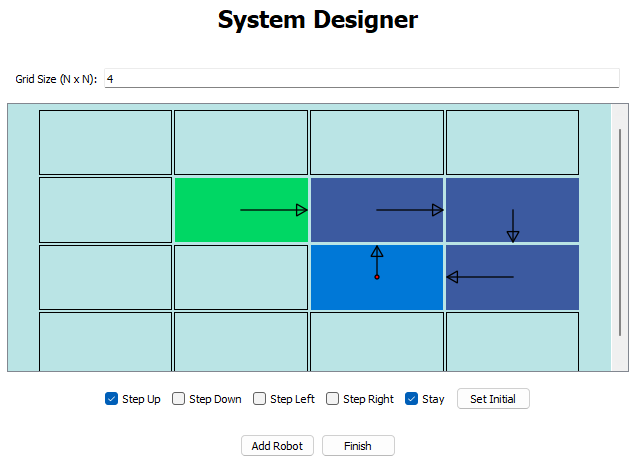


Figure 10 – System Designer

The Generator Form facilitates the automated generation of systems, encompassing all parameters outlined in the system generator section. Users input these parameters into the form, where input validation occurs to ensure the integrity of the data. If invalid data is provided, the form notifies users of any errors, promoting accurate and reliable system generation.

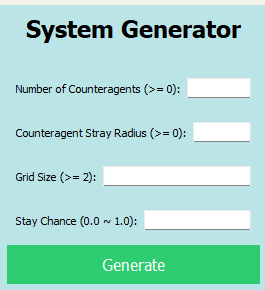


Figure 11 – Generator Form

The System View interface is presented upon successful loading of a system, prompting users to specify runtime parameters. Within this window, users have the option to display a plot showcasing the Kripke structure of the counteragents system. By selecting individual states within the Kripke structure, users can view the corresponding occupancy matrix generated during the Flattening stage. Additionally, the System View interface allows users to save the system into a file, enhancing accessibility and data management capabilities.

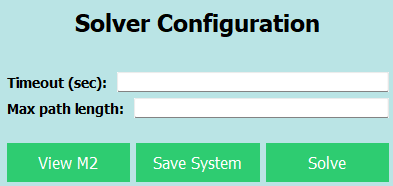


Figure 12 – System View

A screenshot of a computer

Description automatically generated

Figure 13 – Kripke Viewer

The Loading View interface is displayed during the solution phase, providing users with real-time updates on the progress of the solver. It conveys information about the number of iterations attempted out of the maximum path length specified in the solver configuration. Additionally, the Loading View interface indicates the duration elapsed since the solution attempt commenced, enabling users to monitor the solver's performance and estimate remaining processing time.

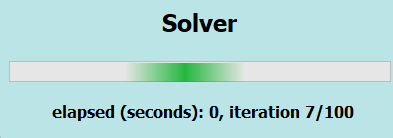


Figure 14 – Loading View

The Result View interface presents the outcome of the solution phase, providing users with essential information regarding the success or failure of the solution process. In the event of a successful solution, the Result View displays the path taken in an animation, illustrating how it navigates to avoid counter agents. Additionally, the interface includes time-point markers for each step in the animation, enabling users to track the progression of the solution. It also calculates and presents the average time per iteration and the total duration of the solution process, offering insights into the efficiency of the solver. In cases where no solution is found, a corresponding message is displayed to inform users of the unsuccessful attempt. Similarly, if the timeout threshold is reached without finding a solution, the interface notifies users, accordingly, ensuring transparency regarding the solver's limitations and constraints.

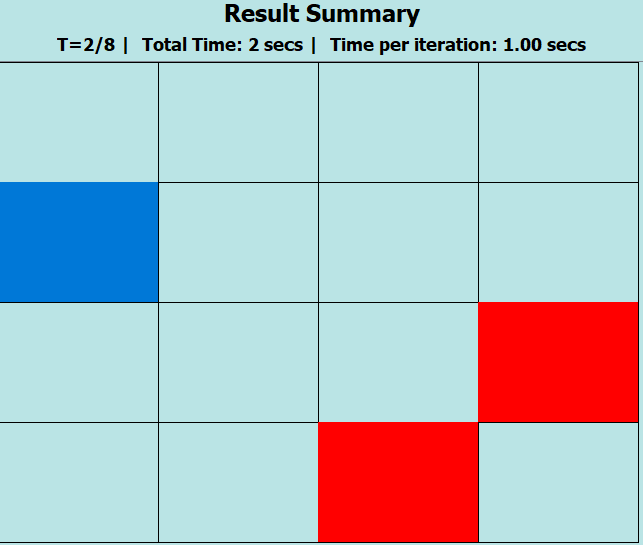


Figure 15 – Result View (Solution Found)

**Services**

*The File Manager* module handles the serialization and deserialization of Kripke objects, facilitating the storage and retrieval of system data. It is responsible for converting Kripke objects into binary data for storage purposes and restoring them back into their original form when needed.  
Loading and Saving the objects is taken/inserted to the Kripke Structure stage in the pipeline.

*The Formulizer* module orchestrates the formulation of logical expressions derived from the Kripke Structure. Its pivotal role lies in crafting formulas ranging from foundational atomic propositions to intricate specifications.  
Leveraging the robust capabilities of the Z3 framework, the Formulizer seamlessly translates structural elements of the Kripke Structure into formal logical constructs. This meticulous process ensures the synthesis of expressive formulas that accurately capture system dynamics and properties.

*The Kripke Generator* is a pivotal software module tasked with the generation of Kripke structures. As its primary input, it requires a system model encapsulating robot entities. Initially, the module undertakes the process of system flattening, as described earlier, wherein the system's hierarchical representation is streamlined into a unified structure. Subsequently, leveraging the flattened model, the Kripke Generator proceeds to construct a Kripke structure. Additionally, this module assumes responsibility for the automated generation of systems based on specified input parameters, as detailed previously. The auto-generated system undergoes analogous flattening and subsequent conversion into a Kripke structure.

*The Solver* module assumes the critical role of executing the Bounded Model Checking (BMC) process. Its core function entails solving the formula generated by the Formulizer module iteratively within a specified bound until either a solution is found, no solution is possible, or a timeout threshold is surpassed. Furthermore, the Solver module encompasses the task of extracting the solution path from the formula's satisfying assignment, thereby providing insight into the trajectory followed to satisfy the specified constraints.

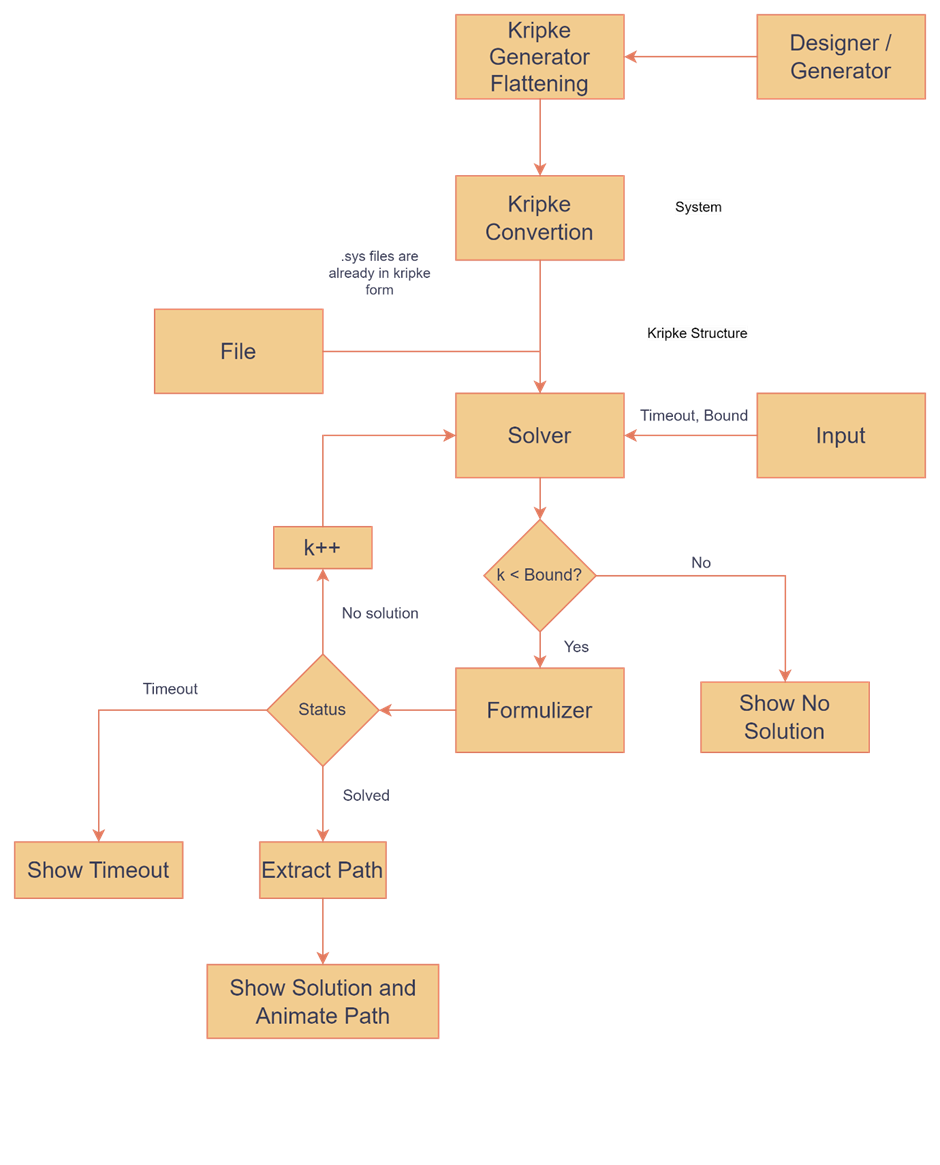


Figure 16 – System Process Flow

**Software Requirements and Specifications**

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Comply** |
| System Creation | The software will be able to create systems in terms of grid and agents. | Yes |
| System Generation | The software will be able to create systems automatically with certain constraints (grid size, counteragents count, etc..) | Yes |
| System Export/Import | The software can be saved as a file to the PC and can be loaded from a file. | Yes |
| Visualization | The software will visualize the loaded system as a Kripke structure.  The software will visualize the path finding process and algorithm advancement. | Yes |
| Algorithm | The software will encode the system as Boolean variables and use these variables to create the algorithm formulas | Yes |
| Solvers | The software will attempt to satisfy the Boolean formula and the resulting assignment will be decoded to the path taken. | Yes |

**Class Diagram**

Our project's class diagram is straightforward, serving to visually articulate the flow of data within our pipeline. As previously highlighted, this pipeline orchestrates a gradual metamorphosis of the data, transitioning it from a human-readable format to one that is computationally tractable.

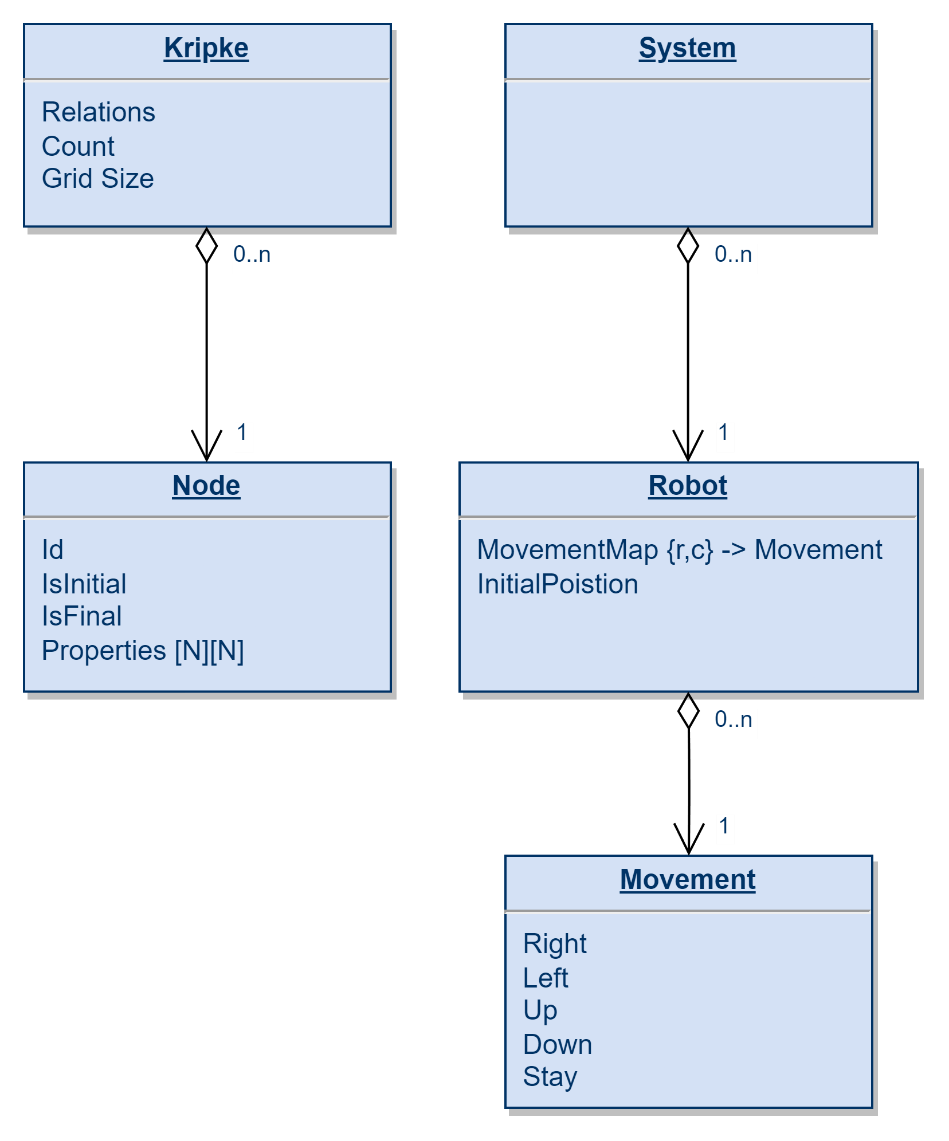


Figure 17 – Class Diagram

# 5. Analysis and Benchmarking

During our benchmarking and analysis phase, our primary objective was to identify the most influential factor in complexity. As previously mentioned, our parameters included grid size, number of counter agents, stray radius, stay chance and path length. To achieve this, we employed a methodical approach, systematically fixing parameters while adjusting a single variable at a time.

To ensure an accurate representation of computational performance, it was imperative to examine many samples for each variable.

Therefore, in these experiments every mark (blue dot) in the graph represents 100 distinct tests.

For each experiment, a hypothesis was formulated and subsequently compared with the actual results obtained.

***The hypothesis is presented before each experiment shown below.***

**Number of counter agents**

The hypothesis posited that augmenting the number of counter agents would lead to a corresponding increase in the count of potential occupants tracked during the flattening phase. However, it's important to note that the flattening phase serves as a preprocessing stage, consolidating multiple counter agents occupying the same position into a single occupancy indicator. Consequently, this process imposes a limitation on the total number of indicators within a single grid, constrained by the square of the grid cell count, .

Figure 18 – Counter Agents vs Solution Time,  
 Grid Size = 15, Radius = 10, Stay Chance = 0.1

The results indeed align with the hypothesis, demonstrating that varying the number of counter agents had only a negligible impact on solution time.

**Stray Radius**

The hypothesis suggested that a larger stray radius would result in counter agents traversing the grid to a greater extent, surpassing their initial positions, and thereby increasing their contribution to grid occupancy. However, it was emphasized that this effect primarily influences the flattening phase, where all occupants within a cell are amalgamated into a single occupancy indicator.

Figure 19 – Stray Radius vs Solution Time,  
 Grid Size = 15, Counter = 3, Stay Chance = 0.1

The results are consistent with the hypothesis, revealing only a slight variation in solution time as the radius expands.

**Stay Chance**

The hypothesis posited that increasing the stay chance of counter agents would heighten the probability of them staying in place, potentially resulting in prolonged occupancy of their positions. In addition, introducing an additional possibility of staying complicates subsequent movement predictions, as the counteragent may opt to remain stationary for an uncertain duration before transitioning to another cell. Nonetheless, it was observed that occupancy indicators remain bound by the dimensions of the grid.

Figure 20 – Stay Chance vs Solution Time,  
 Grid Size = 15, Counter = 3, Radius = 10

The results are consistent with the hypothesis, revealing only a slight variation in solution time as the chance of having the option to remain stationary increases.

**Path Length**

The hypothesis suggests that elongating the path length amplifies the number of steps an agent can undertake to discover an optimal route, thereby necessitating additional propositions in the formulas , the number of total atomic proposition (also known as base propositions) can be calculated by where is the path length and is the grid size. Consequently, this could extend the time required for the solver to find a satisfying assignment, potentially resulting in increased solution time.

Figure 21 – Path Length vs Solution Time,  
 Grid Size = 15, Counter = 3, Radius = 10, Stay Chance = 0.1

The results corroborate the hypothesis, demonstrating a consistent, nearly linear escalation as the path length increases. This indicates that the solver is compelled to exert more effort in finding an assignment that satisfies the formula under scrutiny.

**Grid Size ()**

The hypothesis posits that enlarging the grid size results in a proportional increase in both the number of rows and columns, thereby initially generating more occupancy indicators. Crucially, this expansion also escalates the count of base propositions, which can be calculated as , where denotes the path length and denotes the grid size. As the quantity of base propositions swells, the length and intricacy of also expand, exerting a substantial influence on solution time and ultimately determining the algorithm's time complexity.

Figure 22 – Grid Size vs Solution Time,  
Counter = 3, Radius = 10, Stay Chance = 0.1

The findings are consistent with the hypothesis, showcasing a substantial surge in the volume of atomic propositions, expressions, and occupancy indicators. This surge triggers an exponential escalation in solution time, validating the anticipated correlation between the increased complexity of the problem and the corresponding rise in computational burden.

**Summary**

The results shed light on the differentiation between three fundamental properties of grid-embedded systems: time, space, and density.

*Density* in this context refers to the concentration of counter agents on the grid and how they obstruct space within it. While the algorithm may not be directly influenced by density, the presence of blocked grids could indirectly lead to longer paths, subsequently elongating solution time.

*Time* in this context refers to the length of the path, as each step of the agent's movement increments this duration. With an increase in time, the solver is compelled to accommodate more constraints in its solution process, rendering the task of finding a satisfying assignment more challenging.

*Space* in this context refers to the spatial arrangement of the agent within the grid. As the grid size expands, the minimum length path () also increases. Consequently, the initial base proposition size becomes , resulting in a base size of

**Conclusions**

The algorithm's objective is to reduce the grid pathfinding challenge into a SAT problem, achieved by translating it into boolean formulas representing space-time coordinates. This process harnesses solvers like Z3, which employ heuristic techniques and optimizations to tackle SAT problems effectively.

Given that the SAT problem is NP-complete, its response grows exponentially with input expansion. When transforming the pathfinding problem into SAT, the grid size emerges as the most influential input, followed by the path length.

# 6. Project Reflection

Initially, comprehending the topic proved to be a challenge, requiring extensive research and consultation with our supervisor. Despite encountering numerous obstacles, we persisted and eventually found effective solutions. Upon achieving a thorough understanding of the concept, we faced the challenge of translating it into machine-readable terms. It became apparent that the raw data (a grid) and the final processed data (the formula) posed difficulties in precise transformation. Subsequently, we devised a pipeline as our primary method of data processing.

This pipeline serves as a great example of the challenges we encountered. One of the initial hurdles was determining how to represent the grid and, more specifically, how to represent multiple counteragents within the same grid. Recognizing that the grid is an abstract concept with no direct representation in the code, we focused on the essential elements: the counteragents. To address this, we decomposed the problem into smaller components, creating objects of the type of Robot (counteragent). These objects encapsulated a data structure capable of mapping grid positions to potential movements. Instead of storing the entire grid, we only retained pertinent data, such as the grid cell occupied by the counteragent and its corresponding movements.

The second challenge proved to be more daunting as we encountered a major hurdle in transforming the grid structure into the Kripke structure. Initially, the system's complexity overwhelmed us as we attempted to meticulously track every movement of each counteragent, resulting in an overwhelming number of alternate realities. However, a breakthrough came when we realized that individual robot movements were not as crucial as identifying occupied grid cells. This insight allowed us to simplify the process by condensing multiple robots on the grid into a single representation with occupancy indicators. As a result, we were able to create a timeline illustrating the grid's occupancy state over time, effectively integrating the concept of time into our two-dimensional space.

The third challenge revolved around formulating the rules of the game. Despite having the timeline and understanding how the counteragents behaved, teaching our "good" robot the rules proved to be time-consuming. Initially, the robot's behavior was erratic: it would either teleport to the finish line or refuse to move altogether. At one point, it even duplicated itself across the grid. However, after extensive trial and error, we developed the formulas mentioned above, each designed to counteract the observed prohibited behaviors.

We credit ourselves for maintaining effective communication through regular meetings and brainstorming sessions, ensuring thorough planning before diving into development. Our approach emphasized sculpting our code and refining our algorithm to achieve optimal results. We utilized source control with GIT and SourceTree, adhering to industry standards by employing branching and pull requests as expected of software engineers. However, we acknowledge our shortcomings in GUI creation and design proficiency, though we made concerted efforts to produce the best possible outcomes in these areas.

# 7. User Manual

When the software is first run, four options are available to the user

1. Importing a .sys file.
2. Generating a system randomly given specific parameters.
3. Designing a system manually.
4. Exiting the software

A screenshot of a computer

Description automatically generated

If the option to design a system manually was made the following window will open

A screenshot of a computer

Description automatically generated

Users can enter the grid size in the top textbox, with any changes clearing current input. To add bad robots (counteragents), users select the initial cell and click "Set Initial." Next, they choose the cell for movements and check relevant checkboxes for each required movement. All paths must form closed loops to prevent undefined behavior, per the designer's requirement. After setting a counteragent, users click "Add Robot" before designing another or proceeding to the next stage by clicking "Finish."

Invalid robot behaviors, such as lacking an initial state or containing open loops, will prompt an error message upon addition.

The Designer can be zoomed in and out with mouse wheel, it could be panned using the mouse right click and auto adjusted using the mouse wheel button.

If the option to randomly generate a system was clicked, then the following window will show.

A screenshot of a computer

Description automatically generated

Users should fill the form as they see fit, elaborate details on each field are presented in the book Software Development section.

Once the form is filled and valid (constraint are details in each field’s label parenthesis) the “Generate” button should be clicked.

No matter which input type was selected in the initial window, eventually you will reach the Solver Configuration window.

A screenshot of a computer login

Description automatically generated

The form should be filled as the user see fit, configuring timeout for the solution time or the max steps allowed for the agent to make.

The “View M2” button can be pressed to open a figure representing the counter agent systems Kripke structure.

A screenshot of a computer

Description automatically generated

Pressing a state in the Kripke structure open a text pane showing the positive occupancy indicator for the grid at that time step, the time step itself is labeled on the states, depicting a timeline.

The figure can be manipulated, panned, and dragged using the controls at the top of the window.

The “Save System” button opens a file save dialog, giving the option of saving the system into a Kripke structure for future use.

The “Solve” buttons initiates the algorithm and solver.

A green and white progress bar

Description automatically generated

After clicking “Solve” the solver window will pop up and show the time elapsed in seconds and iteration (current path length attempted).

If a solution is found it will be presented on a grid, showing how occupancy indicators change along with the agent route.

A screenshot of a graph

Description automatically generated

The agent is the blue square while the red squares are occupied cells or potentially unsafe cells, as the animation rolls the “T” field will increase, the animation runs in a loop. Statistics of the total time taken, and the average iteration time are also shown to the user.

In the event of a timeout or no solution a window will pop up with the relevant prompt.

# 8. Maintenance Guide

The software was developed using Python 3.11, incorporating the following libraries.

|  |  |
| --- | --- |
| **Name** | **Description** |
| PyQt5 | GUI development framework |
| contourpy | Contour plotting library |
| cycler | Cycling color maps |
| fonttools | Manipulating font files |
| graphviz | Graph visualization |
| kiwisolver | Mathematical optimization |
| matplotlib | Plotting library |
| networkx | Network analysis |
| numpy | Numerical computation |
| packaging | Package management |
| pillow | Image processing |
| pip | Package installer |
| pyparsing | Parsing library |
| python-dateutil | Date and time manipulation |
| scipy | Scientific computing |
| setuptools | Package setup tools |
| six | Python 2 and 3 compatibilities |
| wheel | Built package distribution |
| z3-solver | Theorem prover |

To set up the project's source code, follow these steps:

1. Start a new project (using your preferred IDE).
2. Setup a virtual environment with a python 3.11 interpreter.
3. Run the command pip install -r requirements.txt.
4. Run the project using the Window.py file.

The project should be run on a Windows machine. The solution process is CPU-bound, meaning that a more powerful machine will result in better computation time

# 9. References

1. Edmund Clarke, Armin Biere, Richard Raimi and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving, abstract.
2. Moura, D. L., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. Springer.
3. <https://en.wikipedia.org/wiki/Model_checking>
4. Characterizing Kripke Structures in Temporal Logic, M.C Browne, E. M. Clarke, O. Grumberg, Carnegie Mellon University, Pittsburgh.
5. [10] Shtrichman, O. (2000). Tuning SAT Checkers for Bounded Model Checking. Springer
6. https://en.wikipedia.org/wiki/Pathfinding