A picture containing text, outdoor, sign

Description automatically generated

Software Engineering Department

Braude College of Engineering

Capstone Project Phase A

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

**A-23-2-R-5**

Students:

Daniel Balaish  
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](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Intro)
2. **Background and related work ………….**…………………………….…………. [6](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Background)

2.1 Kripke Structures ……………………….………….…………………... [6](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Background)

2.2 SAT problem ………………………….………….……………….…...... [7](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#SatProblems)  
2.3 SAT Solver …………………………….……………………….……...… [7](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#SatSolvers)

1. **Expected achievements** ………………………….………………………..…….... [7](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#ExpectedAchievements)
2. **Research and Development**……………………….………………………………. [8](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#ExpectedAchievements)

4.1 Learning Stage ……….………………………..…………………………... [8](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#ResearchProcess)

4.2 Product ………………………………………………….......…………........ [8](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#ResearchProduct)

4.2.1 System Representation ……….......………………….………… [9](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Algorithm)

4.2.2 System Generation……………….………..…………..……….. [10](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#GUI)

4.2.3 System Import/Export…………………………………………… [11](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#GUI)

4.2.4 A Model of The System ………....……………………………… [11](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#GUI)

4.2.5 Encoding a system ……………………………………………… [12](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#GUI)

4.2.6 Algorithm ………..………………………………………………... [13](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Algorithm)

4.3 Software Design…………………………….……………………………… [14](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#Algorithm)

4.4 GUI ……….…………………………………………..………………….….. [19](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#GUI)

1. **Evaluation plan** …………….……………………………………………………..… [23](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#EvaluationPlan)
   1. Validation (Bug-free + Test plans)………………………………………… [23](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#References)
   2. Experimental plan (Grid size, Counteragents, Motion pattern)………… [24](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#References)
2. **References** …………..…….……………………..………………………………..… [25](file:///C:\Users\Liron%20lavi\Downloads\Capstone%20Project%20Phase%20A-23-2-R5%20New%20Model%20(1).docx#EvaluationPlan)

**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, 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.

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*.

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

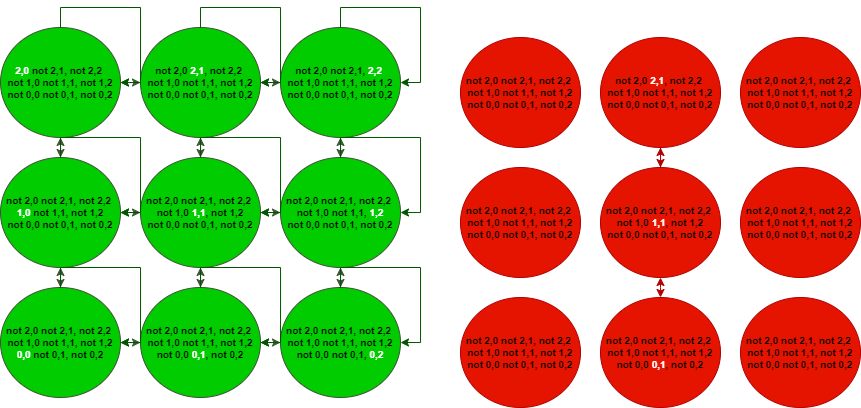
**

Figure 2

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:

1. **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.  
**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.

**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’. 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: .

1. **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.

1. **Research and Development**

**4.1 Learning Stage**In Phase A it was necessary for us to learn the concepts and basics used for achieving the goals: BMC, Kripke Structures, Encoding Kripke Structures, SAT Solver.  
after learning the basics we started looking at the algorithm and the based approach to implement it including which methods and data structures to use.  
once the foundation was ready we could start designing the software architecture, decide on the features, tools and looks of the GUI.

At the end we used that information to create a top view of the project idea.

Diagram

Description automatically generated

Repeat for larger path.

Figure 3

**4.2 Product**In this section, we provide a detailed description of the algorithm, discuss data structures, features, and general system architecture.

**4.2.1 System Representation**

**Our Data structure**  
we will save an array of all the vertices, each vertex in the array would have no more than 4 neighbors in his Neighbors List so we can find every vertex or edge in the complexity of where is the total number of vertices in the system. Using this data structure is the most efficient option.

Diagram

Description automatically generated

Figure 5

**Adjacency List**

A common data structure to be used for low volume graphs representation is  
an AdjacencyList. In an adjacency list an edge is a connection between two vertices in a graph. The list of edges forms a list of pairs of vertices that are connected by edges in the graph.

תמונה שמכילה תרשים

התיאור נוצר באופן אוטומטי

Example: The adjacencylist in Figure 4 is

Figure 4

**Why Adjacency List?**

In our project we are working with two systems in which each vertex would be represented by a possible image of the grid with an cells (, that’s why each vertex would have no more than 4 neighbors (left, right, top, bottom) or less depending on his location in the grid. The complexity of accessing a neighbor is

**4.2.2 System Generation**

In our project we would want to run many different types of system, therefore we will need a subsystem that produces them.  
  
A system can be generated in two ways:

1. Using the software integrated system designer – a utility used for creating a system for the agent and counteragent.
2. Using the integrated auto generation - to provide a fast automated benchmarking we need to be able to generate systems automatically based on input parameters such as:

* Number of counter agents
* Max counteragent steps
* N (size of grid)

The operation will create a system file that can be used and visualized using the developed software GUI.

Figure 6 is an illustration of how a system is created. an agent is created and assigned a start point. Then each cell can be assigned an arrow with the direction the agent will go when on that cell. Once all agents are placed and configured the button generate can be pressed.

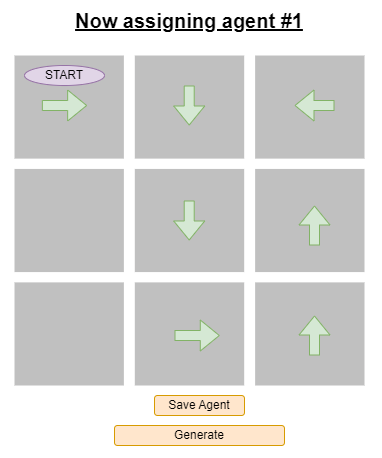


Figure 6

**4.2.3 System Import/Export**

Generated systems are produced as a *.sys* file and can be reused  
and read by the software. Generated/auto-generated systems can be exported as a .sys file into the PC’s filesystem and vice versa.  
The format of the file is the actual form of the Kripke structures representing the system.

**4.2.4 A Model of The System.**A Kripke Structure can be used to model a system; in our project a system can be defined as a collection of patterns of movement our agents can produce.

On an grid a *step* can is represented a connection between two adjacent cells. Hence, the cells of the grid define the atomic properties of the Kripke Structure making the transition relation as the step relation.Intuitively a path is a combination of steps where each *step* is one of up, down, left, right or stay.  
  
The model can be formally described as follows:

Let system be where  
-

Explanation: a state represents a picture of the grid where each agent is in a specific position.  
  
-

Explanation: in the initial state the main agent is positioned in the bottom left cell and the counteragents are distributed among the cells.  
-

Explanation: a connection exists between two states means that every agent that was in the position in the current image (state) must move to an adjacent cell (hence the ) or stay in the same place (hence that is also possible)

- the agent of the system is in the position on the grid

every path is made out of a sequence of images with valid transitions between them therefore a path in the system represents the actual grid paths of an agent along a discrete timeline (where time is measured in step) as a result finding a sequence of states that does not intersect with any of the counter-agents is equivalent to finding a safe path.

**4.2.5 Encoding a system  
\*** For comfort will be written as

As part of the algorithm implementation, the modeled system needs to be represented as a Boolean formula. every Kripke Structure can be represented as a Boolean formula in the following way:  
Let system be on a grid  
Let   
Let be variables  
Let   
Let

is a formula where if so can be defined as

Explanation: We will assign a length binary vector to each state, where the value of the vector value is the same as the index of the state. An assignment will satisfy only if it corresponds to an actual index of a state.

is a formula where if so can be defined as

Explanation: given two binary vectors as assignments (each vector represents a state), satisfies only if the indices of the vectors are the indices of related states.

Therefore, we will define a path in the length of as

Explanation: as previously mentioned a path is a sequence of related states, therefore, satisfies only if the first state in the path is the initial state of the system which corresponds to the first vector ( followed by a sequence of related states (which is the same as stating that for each pair of states satisfies).

To encode the atomic propositions of a state we will add another bit for each symbol,  
that is because every atomic proposition is either for each state.

In our system, we will encode each state with a *position vector* which is an length binary vector which represent attendance for each of the grid cell in the state.

For example, let   
so the generated formulas are:  
 can be presented as the binary vectors and respectively.

it can be seen that satisfies under 010 or 101 and satisfies under (010,101)

So let us consider a path and check whether it is a valid path for   
according to our encoding: =

**4.2.6 Algorithm**Given two systems represented as Kripke structures  
 where represents the agent , are the counteragents and is the number of counteragents

From the encoding part we know the following constants:

– number of states in   
 – number of states in   
 where is the grid size  
 where is the number of counteragents

The basic idea is to find a path on such that all paths on agree on a case of atomic propositions.

In our case we wish to not collide with any counteragents  
therefore the atomic proposition set which we wish to agree on between the systems is

Initially we will define a relation:

note that is defined as a Boolean formula which is inferred from the AP encoding.

Explanation: T represents the agreement formula of states for different systems, and .

Let be the condition for a possible length path on with possible moves of .

We would demand an agreeing state to implicit that each possible successor state on would not collide with the successor state thus creating the definition of .

Note that are given to us as input ( and therefore will be calculated beforehand.

Let be the formula that is satisfied if the first state of agrees with the first state of

Note that the destination state is known to us (by its encoding) and therefore  
for a length path and variables encoding we will define to be:

Finally, we will define . represents a length successful path, meaning that it is valid, reaches the destination and avoids all counter agents. A satisfying assignment is equivalent to a concrete successful and can be reversed to find the actual path on the grid (Note that all functions used in the encoding can be inversed).

We will run the following formula in a SAT solver.as mentioned above finding a satisfying assignment is equivalent to a concrete successful path, therefore, it is a stop condition for the algorithm.

In the case of not finding a satisfying assignment we will repeat the process for a  
 length path until a satisfying assignment achieved or a timeout.

**4.3 Software Design**

**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. | Optional |
| 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 |

Note that the software will be developed in Python using *Z3-Solver* framework,  
the general architecture of the system is as follows:

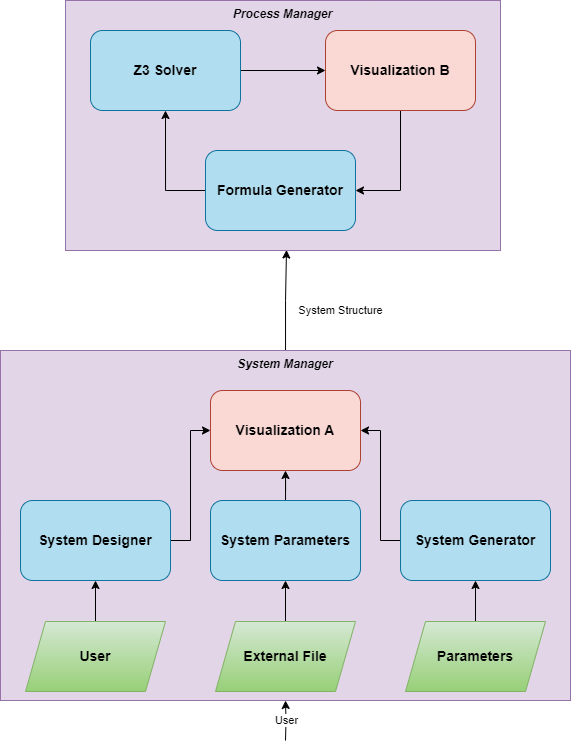


Figure 7 – System Architecture

**System Manager Module –**

1. **System Designer** – this subsystem allows the user to create their own system. only the counteragents system can be created. The user can add agents, the user describes the behavior of each agent. The system will be generated based on the user design.
2. **System Importer/Exporter** – this subsystem allows the user to save the current loaded system into the PC’s file system through a file-save dialog  
   and import a predesigned system (.sys extension).
3. **System Generator** – this subsystem receives as input parameters such as grid size, amount of counter agents, longest paths (Radius), deterministic/non-deterministic routes. With these parameters a pseudo-random system is created, eliminating the need to construct many systems manually when evaluating the algorithm many times.
4. **Visualization A** (optional) – this subsystem will visualize the system created as a Kripke structure diagram.

**Process Manager Module –**

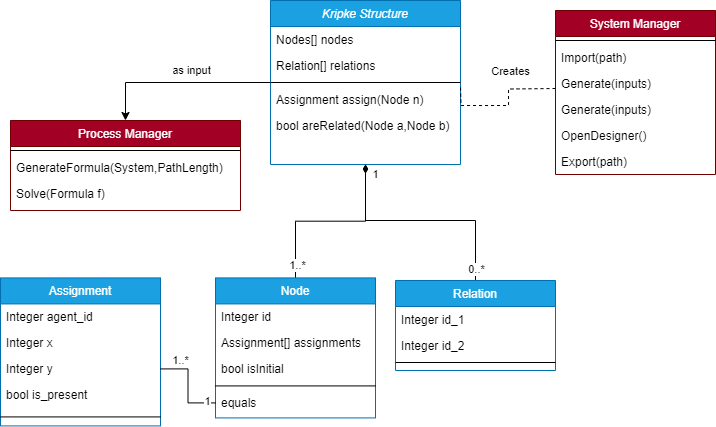
1. **Formula Generator –** in the first iteration, the Formula Generator creates a path of length and prepares the formulas mentioned in the algorithm section. Then it feeds these formulas to the Z3-Solver. (Upon receiving the result from the Z3-Solver it generates a length path.)

The formula generator will not be invoked in the case of or (both are stop conditions for the algorithm).

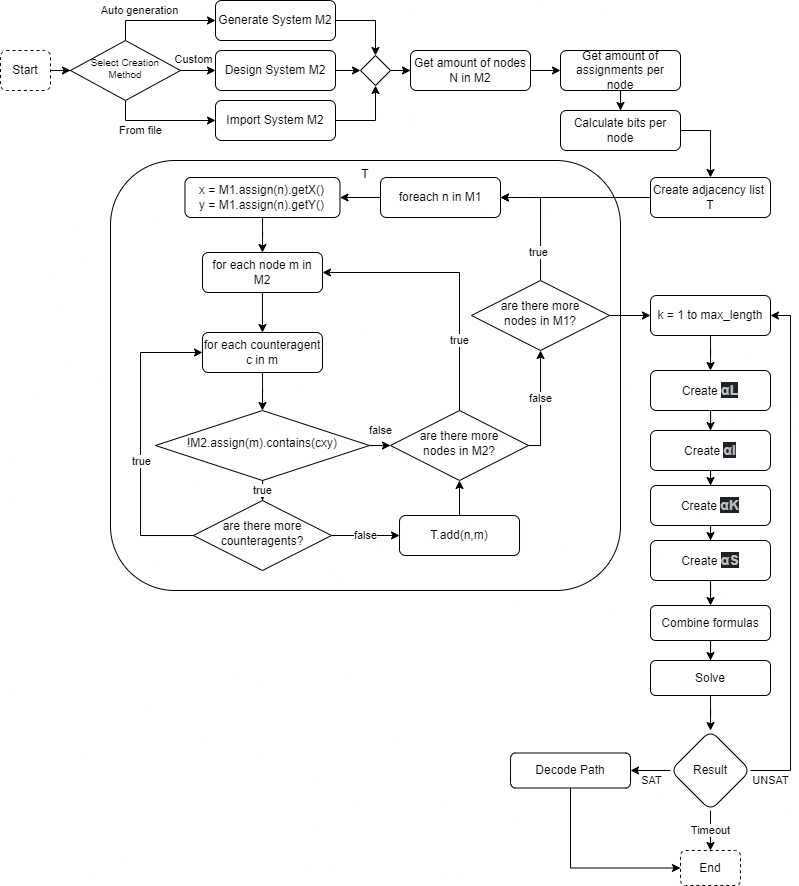
1. **Z3 Solver** – runs the created formula and attempts to find an assignment that satisfies the Boolean formula, note that Z3 is used as 3rd party software in this project.

**Class Diagram**

The system classes will be used to represent Kripke structures. As stated, every Kripke structure has a 4 tuple of: nodes, system relation, assignments and initial states, which all can be defined in the class model.

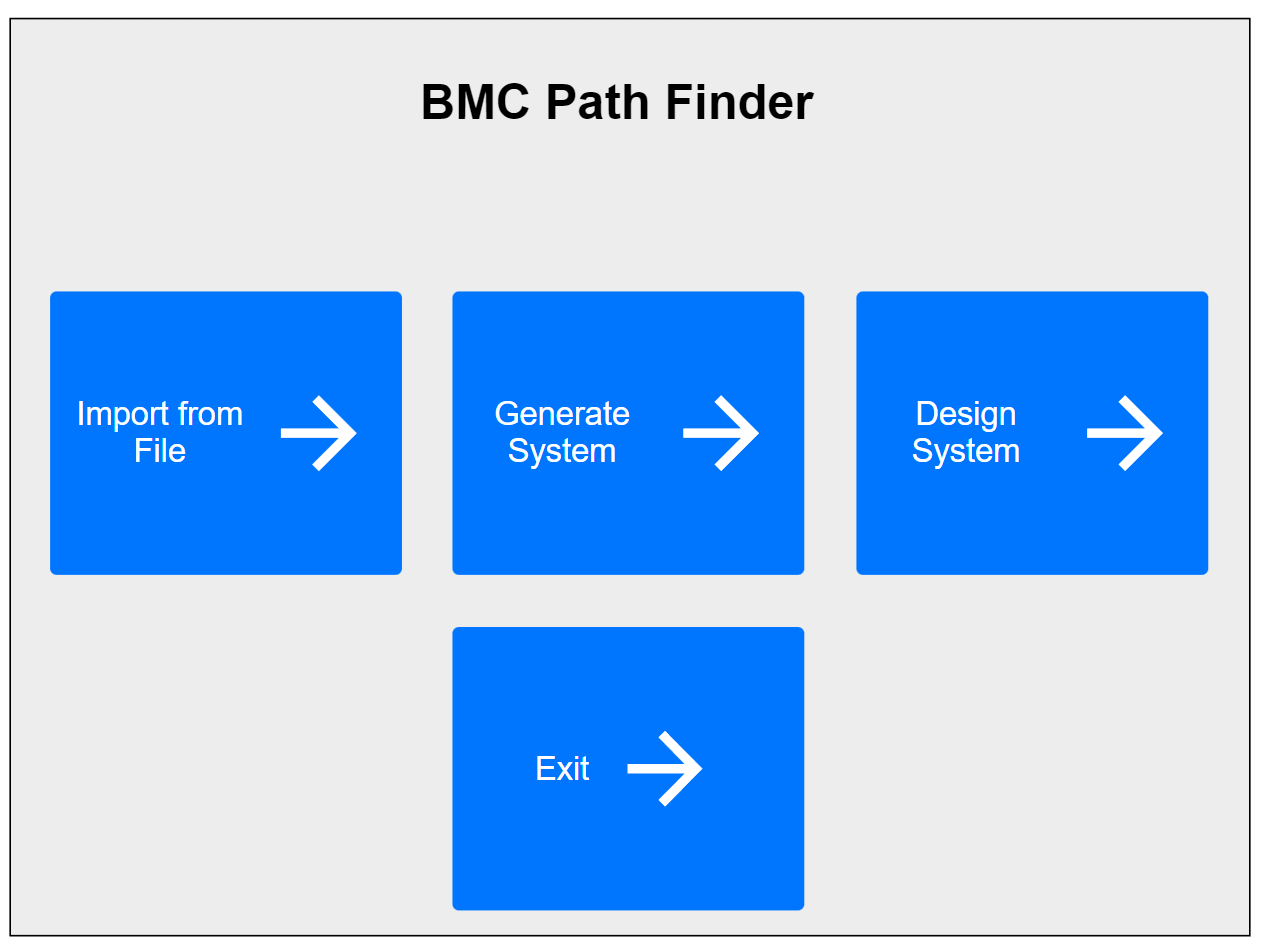
****

**Process Flow**



**4.4 Graphic User Interface (GUI)**

Initial Screen:



System Generator:

A screenshot of a computer

Description automatically generated

System Designer:

A screenshot of a computer

Description automatically generated

Simulation Configuration:

A screenshot of a computer

Description automatically generated with medium confidence

Running Screen:

**A screenshot of a computer

Description automatically generated with medium confidence**

Found Screen:

**A screenshot of a game

Description automatically generated with medium confidence**

Not Path Screen:

**A screenshot of a computer

Description automatically generated with medium confidence**

Timeout Screen:

**A screenshot of a timer

Description automatically generated with low confidence**

**5.Evaluation plan**

**5.1 Validation – Testing the software functionality (Black box testing).**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Type** | **Test No.** | **Test Description** | **Test Result** |
| Algorithm | 1.1 | Run on | Path found |
| Algorithm | 1.2 | Run on with 1 counteragent with possible path | Path found |
| Algorithm | 1.3 | Run on with 2 counteragents with possible path | Path found |
| Algorithm | 1.4 | Run on with 2 counteragents with no path | No path found |
| Algorithm | 1.5 | Run on with 1 counteragent with no path | No path found |
| Algorithm | 1.6 | Run on with stationary counteragents | No path found |
| Algorithm | 1.7 | Run on a very large grid | Timeout |
| Designer | 2.1 | Click on cell when initial position not set | Step checkboxes disabled |
| Designer | 2.2 | Click on edge cell | Edge steps checkboxes are disabled |
| Designer | 2.3 | Clicked on cell | Cell’s border becomes red |
| Designer | 2.4 | Click Set Initial Position on cell | Cell marked green.  Step checkboxes become enabled |
| Designer | 2.5 | Grid size text string input | Error notification |
| Designer | 2.6 | Click add counteragent | Clears grid and statuses |
| Designer | 2.7 | Click Create | Save File Dialog opens |
| Designer | 2.8 | Click Cancel | Back to initial screen |
| Configuration | 3.1 | Enter invalid timeout | Error notification |
| Configuration | 3.2 | Enter invalid path length | Error notification |
| Configuration | 3.3 | Display a predetermined automaton | The automaton is displayed correctly |
| Configuration | 3.4 | Click run | Running window opens |
| Runner | 4.1 | Click cancel | Simulation stops and returns to initial screen |
| Software | 5.1 | Export a system to PC | File is created on the selected path with the selected name |
| Software | 5.2 | Import a system from PC | System is loaded into the software correctly |

**5.2 Evaluation - Experimental plan**

The purpose of the evaluation plan is to test the algorithm limits in terms of complexity, performance, and memory for each of the following parameters  
we will test the iteration time (average and max) and total time.

|  |  |  |  |
| --- | --- | --- | --- |
| #Counteragents | Grid Size | Max Path Length | Max Counteragent Distance |
| 1 | 3x3 | 5 | 0 |
| 3 | 10x10 | 100 | 20 |
| 3 | 12x12 | 144 | 20 |
| 3 | 15x15 | 225 | 20 |
| 3 | 20x20 | 400 | 20 |
| 5 | 10x10 | 100 | 20 |
| 7 | 10x10 | 100 | 20 |
| 9 | 10x10 | 100 | 20 |
| 11 | 10x10 | 100 | 20 |
| 3 | 10x10 | 100 | 25 |
| 3 | 10x10 | 100 | 30 |
| 3 | 10x10 | 100 | 35 |

**References**

1. Edmund Clarke, Armin Biere, Richard Raimi and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving, abstract.
2. [6] 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