Implementation and Verification of the Beta Algorithm for Robot Swarm Using Timed Automata and UPPAAL

Szymon Gałecki - sgal@itu.dk

Thesis Projects - Spring 2024
Speciale - KISPECI1SE

December 2024

Abstract

Swarm robotics is a field where multiple simple robots work together to perform tasks without relying on a central controller. This paper focuses on the Beta algorithm, which is designed to keep robots connected through communication with nearby robots, without knowing their exact positions. The Beta algorithm is modeled using timed automata and implemented and verified using UPPAAL. Verification focuses on examining how efficient the Beta algorithm is in preventing the robots of the swarm from disconnecting. Both asynchronous and synchronized versions of the algorithm are explored to examine the influence of the mode of concurrency on the effectiveness of the algorithm.

Contents

1 Introduction

In swarm robotics, multiple robots collaborate to solve problems by interacting with one another and their environment, mimicking the behavior of bees, ants, or birds [12]. Achieving aggregation is the first step in enabling interaction within the swarm. One of the more popular algorithms addressing such swarm behavior is the Alpha algorithm [9], which has been modeled and verified in [14], [13], [6], and [3]. The improved version of this algorithm, called Beta, was suggested as an interesting direction for future work in [6] and [3].

This paper focuses on modeling, implementing, and verifying the Beta algorithm to assess its effectiveness in maintaining swarm coherence and preventing disconnections. The Beta algorithm was designed to overcome the limitations of its predecessor by using the connection information of connected robots. A single robot implementing the Alpha algorithm would only know about its own connections. In the Beta algorithm, a robot accesses the connection information of its neighbors. To transform the pseudocode presented in [9], we leverage timed automata for modeling the algorithm and the UPPAAL tool for implementation and verification. Both asynchronous and synchronized implementations of the Beta algorithm are explored to evaluate the impact of concurrency modes on swarm behavior, as done similarly in [6].

The structure of this paper is as follows: Section 2 reviews related work on swarm robotics algorithms, focusing on the modeling and verification of the Alpha and Beta algorithms. Section 3 provides background information on timed automata and introduces UPPAAL. It also presents a complete example demonstrating how to transform code into a timed automaton, implement it in UPPAAL, and verify it using UPPAAL. In Section 4, we outline the methodology for modeling the Beta algorithm, detailing the assumptions, parameters, and system design choices made to adapt the algorithm for simulation and verification. Section 5 presents the implementation of the Beta algorithm using timed automata and discusses the differences between asynchronous and synchronized modes. Section 6 details the verification process and analyzes the results, verifying the correctness of the implementation for both modes of concurrency. It also evaluates the effectiveness of the Beta algorithm in maintaining swarm coherence under different conditions for both asynchronous and synchronized modes. Section 7 discusses the findings, including the challenges and limitations encountered during verification. Finally, Section 8 concludes the paper by summarizing the results and suggesting directions for future research.

2 Related Works

The study in [6] verified the correctness of the Alpha algorithm by using model checking on a state-space-reduced system with various modes of concurrency. To address the state explosion problem, the system was limited to 2–3 robots operating on a grid sized 5x5 to 8x8. The concurrency modes examined included synchrony, strict turn-taking, non-strict turn-taking, and fair asynchrony. Synchrony was determined to be the most accurate mode of concurrency for modeling real-world execution. The property "no specific robot will remain disconnected forever" was defined using propositional linear-time temporal logic and verified with the symbolic model checker NuSMV. This property was successfully verified for a system with two robots executing under synchronous concurrency. However, it was falsified for all systems consisting of three robots. The study also suggested the Beta algorithm as a potential topic for future research and highlighted the importance of examining algorithms under different concurrency modes, as they have a significant impact on the results of verification.

In [14], the Alpha algorithm was simplified in a manner similar to this work. The paper described the process of working with swarm algorithms within a verification framework. It focused on the Alpha algorithm, which is the direct predecessor of the Beta algorithm. Temporal logic was used to formally specify the emergent behaviors of a robotic swarm system, and the design choices made to the algorithm ensured its feasibility for verification. Two properties were defined but not verified. The first property states, "It is repeatedly the case that for each robot, we can find another robot so that they are connected." The second property states, "Eventually, it will always be the case that every robot is connected to at least k robots", where k is a predefined constant. Notably, k is also a variable used in the Beta algorithm, which extends this approach by incorporating information about the connections of neighboring robots. These properties, with modifications, are also applicable to the Beta algorithm.

The paper [7] presented concepts and notation to automatically determine whether a swarm would exhibit emergent behavior regardless of the number of agents involved. This approach was demonstrated using the Beta algorithm. Although this work is closely related to the algorithm I model, implement, and verify, I was unable to utilize its findings as the concepts were too advanced for my current level of expertise in verification.

3 Background

This section introduces UPPAAL, a tool for modeling, simulating, and verifying real-time systems. We start with a definition of timed automata, followed by an explanation of its role in modeling real-time behavior. We then outline how UPPAAL implements timed automata. To illustrate these concepts, we use a solution to the mutual exclusion problem as an example. We also explain how to verify properties in UPPAAL using logical quantifiers to define system properties. The example is used to demonstrate the process of modeling and verification in UPPAAL.

3.1 UPPAAL

UPPAAL [8] is a complete tool for modeling, simulation, and verification of real-time systems. Systems can be modeled as networks of automata and timed automata. A system is composed of one or more models that consist of locations and transitions between locations. Simulation involves traversing the state space to obtain possible paths within the defined system. Simulation is used to interactively check if the system behaves as expected. Verification is realized through model-checking. In the process of verification, properties defined for the system are determined to be valid or not. If the property is found to be false, UPPAAL will produce a diagnostic trace, a path through the system that contradicts the checked property.

UPPAAL is an appropriate tool to model a robot swarm. A robot swarm is composed of multiple uniform robots. A single robot can be modeled as a timed automaton implementing an algorithm of our choice. A system consisting of multiple uniform timed automata can be used to simulate an algorithm implementation for a robot swarm. This allows us to simulate the robot swarm and perform verification. Verification through model checking can be utilized to verify the correctness of the implementation of the algorithm as well as for the emergence of the desired behavior of the swarm.

3.2 Timed automata in UPPAAL

The timed automaton defined in the work of Rajeev Alur and David Dill [2] is a basis for timed automata used in UPPAAL. Additionally, UPPAAL extends the Definition 3.1 of the automaton with invariants and variables of boolean and integer type. The invariant is a progression condition on the system. It states that the system is allowed to stay in a given location only for a specified time before being forced to transition. A transition between locations can be decorated with a guard, a logical condition on the system variables, or clocks. If the logical value of the guard is true, the transition is enabled and disabled otherwise. Transitions can be associated with the synchronization action. Synchronization in UPPAAL is based on handshaking; therefore, one transition is responsible for sending the synchronization signal, and one or more transitions will wait for it. A transition that is waiting for the synchronization signal will remain disabled until the signal is received. This mechanism allows for multiple processes to synchronize their transitions. During transition, it is possible to reset clocks and assign values to variables. These clock and variable values are then used to determine the logical value of the transition guards.

Definition 3.1 (Definition of timed automaton [2])**.** A timed automaton is a tuple (Σ, S, S_0, C, E) where:
Σ - input alphabet;
S - finite set of automaton states;
$S_0 \subseteq S$ - set of start states;
C - finite set of clocks;
$E \subseteq S \times S[\Sigma \cup \epsilon] \times 2^C \times \Phi(C)$ - set of transitions

An edge in timed automaton is a tuple $\langle s, s', \sigma, \lambda\delta \rangle$, where:
s - origin state;
s‘ - destination state;
σ - input symbol for the transition;
λ - set of clocks to be reset with this transition;
δ - condition enabling the transition;

3.3 Modeling in UPPAAL

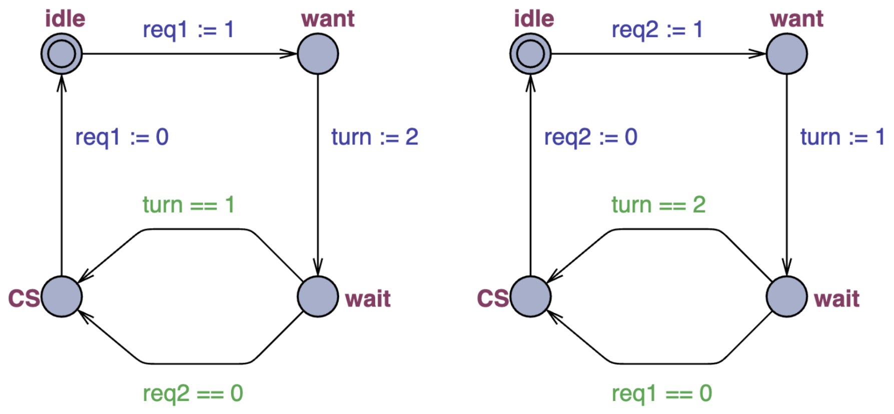
To explain the process of modeling in UPPAAL we will use one of the example models described in the UPPAAL tutorial [1]. The example model implements Gary L. Peterson's solution to the mutual exclusion problem [10]. Figure 1 presents his solution to the problem of two processes sharing access to the critical section. A critical section is a part of code that must be executed only by a single process at a time [11].

Figure 1: Peterson's mutual exclusion algorithm [1], [10]

Process 1	**Process 2**
`req1=1;`	`req2=1;`
`turn=2;`	`turn=1;`
`while(turn!=1 && req2!=0);`	`while(turn!=2 && req1!=0);`
`// critical section:`	`// critical section:`
`job1();`	`job2();`
`req1=0;`	`req2=0;`

Peterson's solution of the mutual exclusion algorithm consists of two symmetrical processes. Each process requests access to the critical section and then sets a flag indicating the other process's turn to access. A process will continuously wait to access the critical section until its turn or until the other process no longer requests access. After accessing the critical section and completing the associated work, the process will indicate that it no longer requests access. This solution guarantees fairness as no process will be indefinitely denied access to the critical section.

Figure 2: Mutex automata in UPPAAL [1]



To model Peterson's solution of the mutual exclusion algorithm we will represent two processes as separate automata. The automaton on the left in Figure 2 will represent `Process 1` from Figure 1 and the right automaton will represent `Process 2`. Both automata have the same set of locations, namely, `idle`, `want`, `wait`, and `CS`. Location `idle` represents the state of the process in which it does not request access to the critical section. Location `want` represents the state of the process after requesting access to the critical section. Location `wait` indicates that the process set the turn to access the critical section to the other process. Automaton will remain in location `wait` until one of the guard conditions gets satisfied and enables the transition to location `CS`. Location `CS` is a critical section of the process. Upon transition from location `CS` to `idle` a process will no longer request access to the critical section.

3.4 Verifying properties in UPPAAL

A subset of timed computation tree logic is used to express properties of the system that are verified by UPPAAL's model checking engine [4]. The system of timed automata is unfolded into a tree with states and transitions. Paths of such a tree are traversed by the model checking engine to determine whether defined system properties are true. System properties must be specified using logical quantifiers presented in Definition 3.2. Letters `A` and `E` are used to quantify over paths while symbols `[]` and `<>` are used to quantify over states within a path.

Letter `A` is used to express property that has to hold for all paths and letter `E` is used for property that holds for at least one path. Analogically, symbol `[]` is used to express that all states within a path must satisfy the property, and symbol `<>` is used to express that there is at least one state within the path that satisfies the property.

Definition 3.2 (Logical quantifiers in UPPAAL [4])**.** The formulas should be one of the following forms
- `A[]`ϕ – Invariantly ϕ.
- `E<>` ϕ – Possibly ϕ.
- `A<>` ϕ – Always Eventually ϕ.
- `E[]` ϕ – Potentially Always ϕ.
- ϕ `-->` ψ – ϕ always leads to ψ.

where ϕ, ψ are local properties that can be checked locally on a state, i.e. boolean expressions over predicates on locations and integer variables, and clock constraints.

To present the verification process in UPPAAL we specified and successfully verified properties of Mutex implementation in Figure 3. The first property states that for all paths and all states of those paths, there is never a situation where both automata are accessing the critical section at the same time. This is a safety property that verifies whether mutual exclusion, the main objective of the algorithm, is achieved. The second and third properties are liveness properties. They state that there exists a path with a state which enables a process to access a critical section. Successful verification of all three properties means that the solution guarantees mutual exclusion and fairness. Fairness in this context means that there exists a path through the system which results in a process accessing the critical section.

Figure 3: Successfully verified properties for mutex [1]

```
A[] not (P1.CS and P2.CS)
E<> P1.CS
E<> P2.CS
```

4 Methods

This section explains how we model swarm behaviors. First, we introduce a taxonomy of swarm behaviors and categorize the Beta algorithm within it. Next, we present the Beta algorithm, including its basic assumptions and design motivations. We then explain the modeling of connections and movement, describing all imposed limitations. We also identify the smallest set of variables required to model the swarm implementing the Beta algorithm. Finally, we introduce a timed automaton for the Beta algorithm, describing its main variables, states, and transitions to show how the algorithm operates.

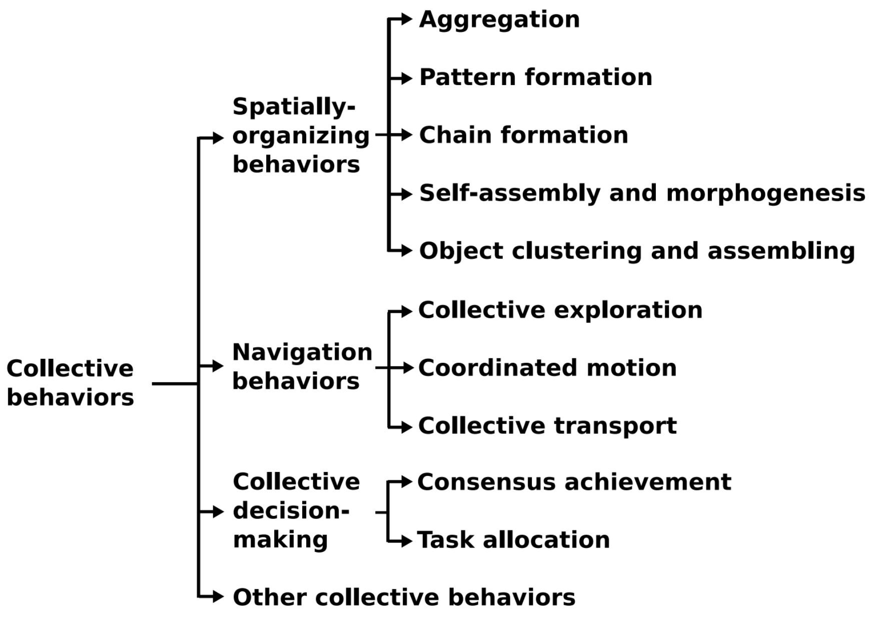
4.1 Aggregation

We will use Manuele Brambilla's [5] taxonomy of collective swarm behaviors, in Figure 4, to show the context for the modeled aggregation behavior. Behaviors categorized as spatially-organizing distribute robots and objects in space. These behaviors serve as fundamental building blocks for more advanced swarm behaviors, as they enable robots to connect, communicate, and interact with one another. Aggregation, the simplest of collective behaviors, groups all robots of a swarm in a region of the environment. If we assume that the robots' environment is unbounded we will have two design choices. The aggregation algorithm will either have to allow a robot to reconnect with the group or never let it disconnect from it. The first approach involves access to external information about the position of the swarm e.g. robot coordinates. The second approach relies on robots being initially connected.

4.2 Beta Algorithm

The Beta algorithm is a robot swarm aggregation algorithm introduced by Julien Nembrini [9]. It is his next iteration of the aggregation algorithm that relies on situated communication. Robots do not have information on their environment and the exact location of other robots. They are connected if they are within communication distance of the physical signal they are using. In the previous approach of the Alpha algorithm, robot movement was determined by the number of connections to other robots. In the Beta algorithm, a robot uses information about its connections as well as about connections of its connections. We will call robots that are directly connected neighbors. When losing a connection, robot will consider how many of its neighbors, are connected to the disconnected robot. If the number of neighbors that are connected to that robot is equal to or lower than the beta parameter, the robot will turn back in an attempt to reconnect. This prevents a situation previously observed in the Alpha algorithm where a single robot could disconnect from the swarm without triggering its reaction as the remaining robots would have a satisfactory number of connections. In the event of high swarm congestion, a robot will choose a random direction. Each robot keeps a list of currently and previously connected robots. Every robot can access its neighbors' current list of connections.

Figure 4: Taxonomy of collective swarm behaviours [5]



The underlying assumption is that connected robots can exchange information. Additionally, a robot does not need to be aware of its direction. As long as it can perform a 180-degree turn and a random turn, it can move. Although the robots are unaware of their positions and environment, the algorithm will prevent them from disconnecting.

4.3 Movement

From the pseudocode and textual description of the Beta algorithm, we know that a robot must be able to move forward, turn 180 degrees, and make a random turn. We assume that a random turn is a turn by a random number of degrees. As the robots in a swarm should be uniform, we assume they move forward by a unit step. Because we will be focusing on verification there is a need to make some arbitrary decisions that limit the state space. We aim for limitations that will not fundamentally change the algorithm itself. The first thing to be limited is the choice of the direction a robot can move. We will limit the choice to four directions, namely, up, down, left, and right. To preserve the randomness of the turn, one of four directions is chosen at random. We assume that our source of randomness produces uniformly distributed results making all directions equally likely to draw. This means that after a random turn, there is a 75% chance of changing direction and a 25% chance of maintaining a

Figure 5: Pseudo-code for Beta algorithm [9]

```
Create list of neighbours for robot, Nlist
k = number of neighbours in Nlist
i = 0

loop forever {
    i = i modulo cadence

    if (i = 0) {
        Send ID message

        Save copy of k in LastK
        Set reaction indicator Back to FALSE
        k = number of neighbours in Nlist
        Create LostList comparing Nlist and OldList

        for (each robot in LostList) {
            Find nShared, number of shared neighbours
            if (nShared <= beta) {
                Set reaction indicator Back to TRUE
            }
        }

        if (Back = TRUE) {
            turn robot through 180 degrees
        }
        else if (k > LastK) {
            make random turn
        }

        Save copy of Nlist in Oldlist
    }
    Steer the robot according to state
    Listen for calls from robots in range
    Grow Nlist with neighbours IDs and connection info

    i++
}
```

current one. The next arbitrary decision we must make to limit the state space for verification purposes is to introduce boundaries in the environment. Robots move in four directions, a unit step at a time. The environment is a grid with points that can be occupied by robots. As the chances of movement in each direction are equal, the shape of the environment should reflect that and be a square. Whether we make a robot turn around when reaching the boundary or introduce a wrap-around mechanism, we want the influence of the boundary to be reflected equally in both horizontal and vertical directions. In our case, a robot reaching the boundary of the environment will turn 180 degrees. Our approach is suitable for modeling two-dimensional environments. The wrap-around mechanism would be more suitable if the robots were to operate on the sphere. In such a case it could be expected for two robots moving away from each other to connect. This is not the expected behavior in a two-dimensional environment.

The last simplification is made to prevent further algorithm modification. The algorithm assumes that robots are unaware of other robots' positions. A robot knows whether it is connected to the other robot but is not aware of its own or its neighbor's coordinates. As the robots can occupy a limited number of points in the bounded environment, there might occur a situation where two robots occupy the same space. We accept that situation for the following reasons. Implementing the collision-avoiding mechanism would lead to significant changes in the algorithm. Our goal is verification of the algorithm so we would like to introduce as few changes as possible. Additionally, it would be hard to estimate the influence of the collision-avoiding mechanism on the algorithm's effectiveness. Lastly, this simplification is acceptable if we assume that robots can be positioned very close to each other.

4.4 Connection

The algorithm assumes that robots use a range-limited omnidirectional signal that enables two-way communication. Each robot broadcasts its ID to inform other robots of its presence. If a robot receives an ID from another robot, they are within the physical range of the signal and are therefore connected. In our case, the environment is two-dimensional, and the physical signal is modeled as a circle centered at the robot's position. All robots within that circle are considered neighbors. The range of the signal is parameterized by the radius length. The radius should be larger than a unit step to allow movement but smaller than the side of the grid to keep it range-limited relative to the dimensions of the environment.

Since we are not dealing with physical robots, we have to mimic the physical signal programmatically. To achieve this we need information about robot coordinates and the radius of the physical signal. Robots are unaware of their own and their neighbor's coordinates. However, robot coordinates are stored and utilized. To determine if two robots are connected, we determine whether the Euclidean distance between their positions is smaller than the radius of their signal. Each robot keeps information about the current and previous number of connections. They also store lists of currently and previously connected robots. By comparing these lists, a robot can track the lost connections. A list of currently connected robots can be accessed by every neighboring robot. This allows a robot to determine how many of its neighbors are still connected to each of its lost connections. If the number of such neighbors is smaller or equal to the beta parameter, a robot will turn back in an attempt to reconnect.

4.5 System variables

To describe a robot swarm implementing the Beta algorithm we must define the minimal set of variables. The first variable is the number of robots that constitute the swarm. It has the highest influence on the resulting size of the state space. It also limits the beta parameter, as each robot can be connected to at most every other robot, excluding itself. The second most important parameter is the beta parameter. The assumption of the algorithm is that a beta equal to two guarantees coherence. This means that verification should be performed for swarms of three or more robots. Although it is assumed that robots' environment is unbounded we must limit it to make verification feasible. Environment is a square defined by the length of its side. Each robot moves in the environment a unit step at a time. It is important that the length of the unit step is relatively small in regard to the size of the environment. If the unit step size is too big we will verify the version of the algorithm where robot interaction will be greatly influenced by reaching the boundary of the environment which is an introduced limitation. The last variable that we have to define is the radius of the physical signal that we are simulating. The radius is related to both the environment size and the unit step. If we make the radius of the signal smaller or equal to the unit step size, we will promote swarm concentration. On the other side, if we make the radius of the signal bigger, we will promote swarm exploration. The upper limit on the radius of the signal is the size of the environment, we never want the signal of a robot to cover the whole environment. This would lead to a situation where a swarm is fully connected and unable to disconnect. Such configuration would lead to untruthful verification of the algorithm as most of its scenarios would not be reachable.

4.6 Automaton

In this subsection, we will present how the pseudo-code for the Beta algorithm in Figure 5 was transformed into a timed automaton in Figure 6. The resulting automaton does not consider limitations imposed on the robots' environment but is an accurate mapping of the original algorithm. Variables and states of the automaton will be introduced, along with their connection to the algorithm's source.

Pseudocode variables that represent the state of a robot:
`i` - loop iterator;
`cadence` - parameter that controls the frequency of robot operation;
`k` - number of robot neighbors;
`LastK` - previous number of robot neighbors;
`Nlist` - list of neighbor ID's;
`OldList` - previous list of neighbor IDs;
`LostList` - list of IDs that are in `OldList` but not in `Nlist`;
`nShared` - number of shared neighbors for each robot in `LostList`;
`beta` - threshold parameter on the `nShared` value;
`Back` - boolean value indicating whether the robot should turn 180 degrees;

We will use a smaller set of variables to represent the state of the timed automaton implementing the Beta algorithm.
`k` - number of robot neighbors;
`last_k` - previous number of robot neighbors;
`lost` - boolean indicator if the robot lost any of its neighbors;
`shared` - the smallest number of shared neighbors for each lost robot;
`beta` - threshold parameter on the `shared` value;
`t` - clock of the robot;
`T` - time threshold;

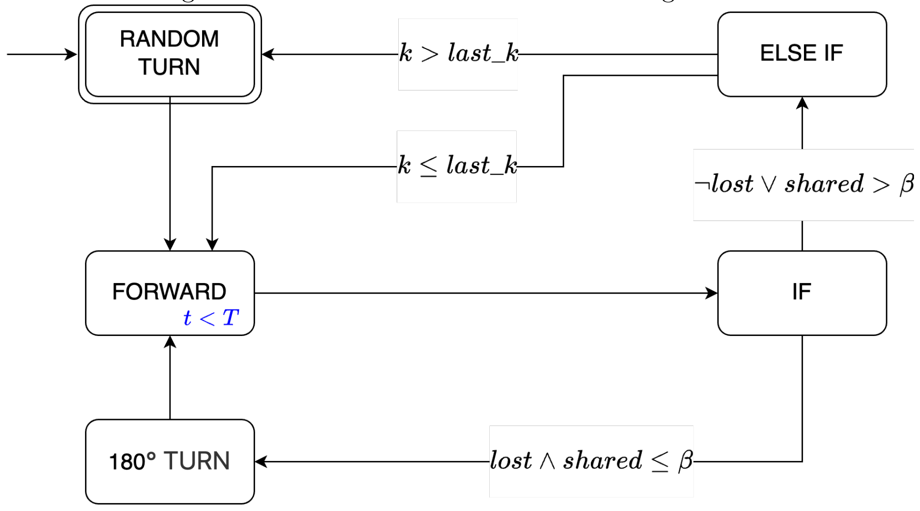
States of the resulting automaton:
RANDOM TURN - where the robot chooses one of four directions at random, it is also the initial state for the robot;
FORWARD - where the robot moves forward in the previously set direction;
180 TURN - where the robot changes direction by turning 180 degrees;
IF - where the robot checks if it should turn 180 degrees;
ELSE IF - where the robot checks if it should make a random turn or continue moving in the previously set direction;

Apart from the states and variables of the automaton it also has an invariant on the FORWARD state. Invariant is a condition on the robot's clock. The robot has to transition from the state with a defined invariant before reaching the time threshold `T`. After transitioning it resets its clock. The FORWARD state is the only state where time is allowed to pass. Transitions between the remaining states take no time and cannot be interleaved by another robot. If a robot is not in the FORWARD state, all remaining robots must be in that state. This means that robots operate asynchronously, take time to move forward, but react to the change in connectivity instantaneously. The asynchronous operation of a robot is limited by a time threshold, which does not guarantee uniform frequency across the swarm's activities. With slight modifications to the automaton, we can achieve different modes of concurrency. Asynchronous operation is one of the algorithm assumptions but we will also verify the algorithm under strict and ordered synchronization.

RANDOM TURN is the initial state of the automaton as we want the robots initialized with a direction. As that state is of the type committed, there is no time progress and a robot immediately transitions to the FORWARD state. A robot can remain in the FORWARD state as long as its internal clock does not exceed the time threshold. Upon transitioning to the IF state it resets its clock. If a robot has at least one lost neighbor with fewer connections to the other neighbors than beta, it will transition to the 180 TURN state and change its direction by 180 degrees. Otherwise, it will transition to the ELSE IF state to determine whether it has more neighbors than before. If a robot has more connections than previously it will transition to the RANDOM TURN state to possibly change direction and avoid congestion. If a robot has the same number of connections or fewer it will transition directly to the FORWARD state to maintain its current direction.

It can be seen that states RANDOM TURN, FORWARD, and 180°TURN are related to the actions of the robot while states IF and ELSE IF are related to the logical conditions defined in the pseudocode. There are always two transitions coming out of IF and ELSE IF states. One of the transition guards will be an original condition from the pseudocode while the other will be a negation of the original condition. This approach guarantees that there will be always one and only one transition available which eliminates nondeterminism of where a robot will transition and prevents deadlock.

Figure 6: Timed automaton for the Beta algorithm



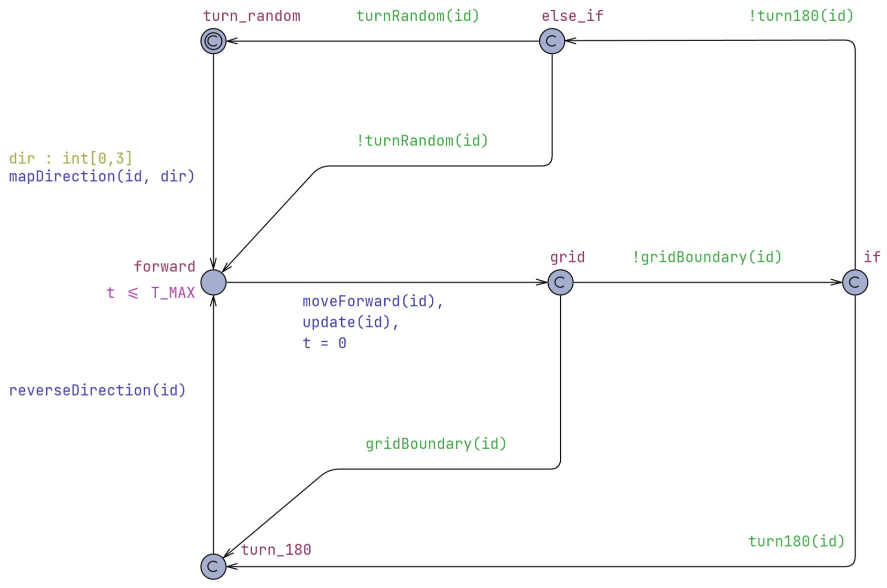
5 Implementation

In this section, we will describe the implementation of the timed automaton model from the previous section, shown in Figure 6. Our tool of choice is UPPAAL, an integrated solution for modeling, simulation, and verification of timed automata. We will present the implementation details and limitations imposed on the model in the following subsection. Model variables representing its state will be described along with the most important functions.

5.1 Model

The implementation of the timed automaton, shown in Figure 7, includes an additional state called `grid`, which is not present in the automaton depicted in Figure 6. Global functions of the implementation abstract away the underlying complexity. The `grid` state is a consequence of imposing limitations on the environment size defined in the previous section. This allows us to reduce the state space size and perform verification. It is the biggest change made to the original algorithm, defined in Figure 5. In that state, it is determined whether a robot has reached the boundary of the grid. If yes, the robot will transition to the state `turn_180` and change its direction by 180 degrees as it cannot continue moving forward outside of its environment. If a robot has not reached the boundary of its environment it will transition to the `if` state where it will transition to further states based on the original rules of the algorithm.

Figure 7: Asynchronous implementation of the Beta algorithm



The implementation of the Beta algorithm presented in Figure 7 is asynchronous as stated by the author of the algorithm. Robots can move at different pace and time, independent of each other. We also implemented a synchronized version of the Beta algorithm to investigate the influence of the concurrency mode on verification results. The synchronized implementation is presented in Figure 8, and its synchronization mechanism in Figure 9. The synchronized version enforces that all robots move at the same time. The synchronization mechanism, which we will call barrier, has a variable `n` initialized to the number of robots in the swarm. Each time a robot transitions to the `forward` state it will use a synchronization channel `done` to notify the barrier. The barrier will decrement its variable `n` upon receiving a signal from a robot. When the value of `n` reaches 0, all robots will be blocked in the `forward` state. The barrier then will send a signal using synchronization channel `step` and reset the value of the variable `n` to the number of robots in the swarm. Robots waiting in the `forward` state will receive this signal from the `step` channel and transition to `grid` state at the same time. They will then progress to the `forward` state in the random order, possibly changing its direction. The order in which they reach the `forward` state does not influence swarm behavior. A robot decides whether to change direction based on the state of the swarm. The state of the swarm, most importantly robot positions, will not change until the next collective step forward. In other words, the direction of the robot is not a variable that influences the behavior of another robot, unlike its position.

Figure 8: Synchronised implementation of the Beta algorithm

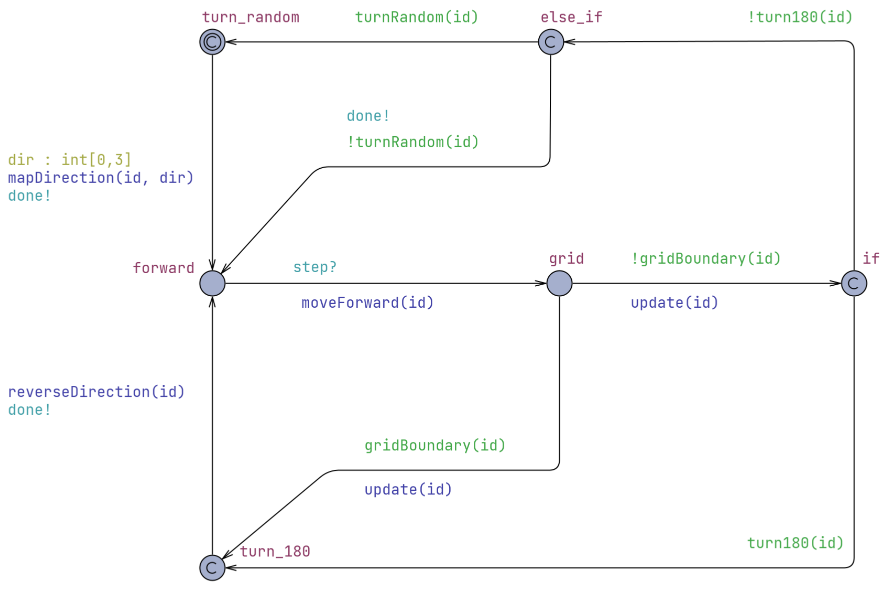
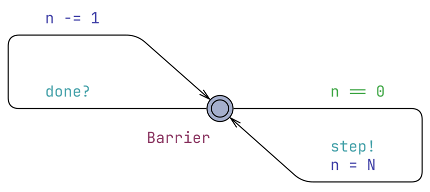


Figure 9: Synchronisation mechanism



5.2 System variables

Presented below variables fully specify the robot swarm in both asynchronous and synchronised version. Implemented functionality depends on those variables and verified properties are only valid in relation to its values.
`N` - number of robots in a swarm;
`R` - radius of the signal used for connection;
`STEP` - step size of a robot;
`BETA` - beta parameter of the algorithm;
`G` - grid boundary, resulting area of the grid = `2G` x `2G`;

Asynchronous implementation variables:
`T_MAX` - time threshold for state invariant;
`C` - global clock for the system;

5.3 Movement

Before explaining the functions that control the movement and direction of the robot, we will first introduce the data structures used by these functions.

`x[N]` - list of x-axis coordinates for each robot;
`y[N]` - list of y-axis coordinates for each robot;
`x_dir[N]` - list of x-axis directions for each robot;
`y_dir[N]` - list of y-axis directions for each robot;

Four global functions govern the movement of the robot, namely, `mapDirection`, `reverseDirection`, `moveForward`, `gridBoundary`. The `mapDirection` function is triggered upon transition from state `turn_random` to state `forward` and provided with `id` of the robot and randomly drawn integer `dir` that will be mapped to one of four directions. The `reverseDirection` function is triggered upon transition from state `turn_180` to state `forward` and provided with `id` of the robot whose direction is to be reversed. The `moveForward` function is triggered upon transition from state `forward` to state `grid` and provided with `id` of the robot that is to be moved forward. The `gridBoundary` function is used as a transition guard for states `if` and `turn_180`. It takes `id` of the robot as a parameter. If a robot reaches or crosses the grid boundary it will transition from state `grid` to state `turn_180` and turn 180 degrees to stay within the grid boundary. If a robot has not reached the grid boundary it will transition to the `if` state and move according to the original specification of the algorithm. The implementation of those functions is shown in Figure 10.

Figure 10: Implementation of functions controlling robot movement

```
void mapDirection(int id, int dir){
    if (dir == 0){
        // up
        x_dir[id] = 0;
        y_dir[id] = 1;
    }
    if (dir == 1){
        // right
        x_dir[id] = 1;
        y_dir[id] = 0;
    }
    if (dir == 2){
        // down
        x_dir[id] = 0;
        y_dir[id] = -1;
    }
    if (dir == 3){
        // left
        x_dir[id] = -1;
        y_dir[id] = 0;
    }
}


void reverseDirection(int id){
    x_dir[id] *= -1;
    y_dir[id] *= -1;
}


void moveForward(int id){
    x[id] += x_dir[id] * STEP;
    y[id] += y_dir[id] * STEP;
}


bool gridBoundary(int id){
    return abs(x[id]) >= G || abs(y[id]) >= G;
}
```

5.4 Connection

The connection part of the implementation focuses on two key tasks: simulating the physical signal real robots would use to detect their neighbors and providing the functionality and data structures needed for the robots to act on this connection information. A robot will detect its neighbor if the Euclidean distance between them is smaller than signal radius `R`. Data structures crucial for both of those tasks are listed below. Two-dimensional lists are adjacency matrices where an entry 1 indicates a connection between two robots and 0 indicates the absence of one.

`neighbours[N][N]` - is a matrix where each entry is 1 if a robot is currently connected to another robot in the system and 0 if not.;
`last_neighbours[N][N]` - is a matrix where each entry is 1 if a robot was previously connected to another robot in the system and 0 if not;
`lost_neighbours[N][N]` - is a matrix where each entry is 1 if a robot is currently considered lost as a neighbor of another robot in the system and 0 if not;
`shared_neighbours[N][N]` - is matrix where each element represents the count of mutual neighbors between each pair of robots in the system;
`shared[N]` - is a list where each element indicates the maximum number of neighbors connected to any one of the lost robots for each robot in the system;
`k[N]` - is a list where each element represents the current number of connections for each robot in the system;
`last_k[N]` - is a list where each element represents the number of previous connections for each robot in the system;

Both of the mentioned tasks are achieved by the `update` function which controls the state of introduced data structures. Its implementation is presented in Figure 11. The function updates connection information for only a single robot, the one identified by `id`. It utilizes the following functions to split the process into logical steps: `updateNeighbours`, `lostNeighbours`, `sharedNeighbours`, `updateConnections`. Implementation of those steps is presented in Figures 12 and 13. It determines the neighbors of the robot based on their positions and signal radius. It checks whether the robot has lost any of the neighbors and if yes it determines how many neighbors are still connected to the lost robot. Finally the function updates current and previous number of connections.

Figure 11: Implementation of `update` function

```
void update(int id){
    updateNeighbours(id);
    lostNeighbours(id);
    sharedNeighbours(id);
    updateConnections(id);
}
```

Figure 12: Implementation of functions: `updateNeighbours`, `lostNeighbours`, `updateConnections`

```c
void updateNeighbours(int id){
    // Updates: neighbours, last_neighbours
    int i;
    last_neighbours[id] = neighbours[id];
    for (i = 0; i < N; i++){
        // Check if robots are within signal radius
        if (connected(distance(x[id], y[id], x[i], y[i]), R)
            && id != i
        ){
            // Connected to i
            neighbours[id][i] = 1;
        } else {
            // Not connected to i
            neighbours[id][i] = 0;
        }
    }
}


void lostNeighbours(int id){
    // Updates: lost_neighbours
    int i;
    for (i = 0; i < N; i++){
        // Check if robot lost connection
        if (last_neighbours[id][i] == 1
            && neighbours[id][i] == 0
        ){
            // Lost connection to i
            lost_neighbours[id][i] = 1;
        } else {
            // Did not loose connection to i
            lost_neighbours[id][i] = 0;
        }
    }
}


void updateConnections(int id){
    // Updates k, last_k
    int i;
    int n;
    last_k[id] = k[id];
    // Count robot connections
    for (i = 0; i < N; i++){
        if (neighbours[id][i] == 1){
                n++;
        }
    }
    k[id] = n;
}
```

Figure 13: Implementation of sharedNeighbours function

```
void sharedNeighbours(int id){
    // Updates: shared_neighbours, shared
    int i;
    int j;
    // Clear entries for the robot
    for (i = 0; i < N; i++){
        shared_neighbours[id][i] = 0;
        shared[id] = 0;
    }
    // Check if robot lost any neighbours
    if (noLostNeighbours(id)){
        return;
    }
    // Count neighbours connected to each lost robot
    for (i = 0; i < N; i++){
        // Check if neighbour 'i' was lost
        if (lost_neighbours[id][i] == 1){
            // Check if robot 'i' is connected to any
                ↪ neighbours
            for (j = 0; j < N; j++){
                if (neighbours[id][j] == 1
                    && neighbours[j][i] == 1
                ){
                    // Neighbour connected to lost robot 'i'
                    shared_neighbours[id][i] += 1;
                }
            }
        }
    }
    // Find the maximum count of shared neighbours for each
        ↪ lost robot 'i'.
    for (i = 0; i < N; i++){
        if(shared_neighbours[id][i] > shared[id]){
            shared[id] = shared_neighbours[id][i];
        }
    }
}
```

The function `updateNeighbours` uses information about robot positions and the radius of the mimicked signal to determine the connectivity of the robot identified by `id`. It uses `distance` function to determine the Euclidean distance between two robots and `connected` function to determine if that distance is smaller or equal to signal radius `R`. Implementation of functions `distance` and `connected` is presented in Figure 14. The function `lostNeighbours` compares the current and previous state of connectivity of the robot identified by `id` to determine if the robot has lost any of its connections. The function `updateConnections` uses data structures updated by the function `updateNeighbours` to count the number of connections for a robot identified by `id`. The function `sharedNeighbours` uses connectivity information of neighbors of the robot identified by `id` to determine the number of neighbors connected to its lost connections. The algorithm assumption is that a robot can access its neighbors' connectivity information. In other words, the robot is aware of its neighbors' neighbours. The most important information determined by this function is the maximum count of shared neighbors out of all lost connections. This value is compared with `BETA` parameter to decide whether the robot should turn 180 degrees in an attempt to reconnect with one or more lost robots. The idea behind the implementation of connection was to partition the algorithm into explainable functions and data structures. Even though the functions are global, access to data structures stays at the individual level of the robot, where it is possible. In the case of modeling the physical signal, access to other robot positions was unavoidable.

Figure 14: Implementation of functions: `distance`, `connected`

```
double distance(int x1, int y1, int x2, int y2){
    return sqrt(pow(abs(x1-x2),2) + pow(abs(y1-y2),2));
}


bool connected(double distance, double threshold){
    return distance <= threshold;
}
```

6 Results

In this section, we will present verification results for implementation and the Beta algorithm for two modes of concurrency. The first mode of concurrency is asynchrony which is the intended mode for the Beta algorithm. The second implementation is synchronized to investigate the influence of the concurrency mode on algorithm properties. In the asynchronous case, presented in Figure 7, robots can move at different rates. Each robot is forced to move from `forward` location after `T_MAX` amount of time but can also transition sooner. In the synchronized implementation, presented in Figures 8 and 9, all robots are forced to move at the same time by the synchronization mechanism.

6.1 Verification of asynchronous implementation

Figure 15: System specification for asynchronous implementation

```
N = 2;       // Number of robots
R = 1;       // Signal radius
STEP = 1;    // Step size
BETA = 1;    // Beta parameter
G = 3;       // Grid boundary
T_MAX = 1;   // Time threshold
```

Figure 16: Successfully verified properties for synchronous implementation

```
1.  A[] forall(i : int[0, N-1]) x[i] != G or y[i] != G
2.  A[] forall(i : int[0, N-1]) abs(x[i]) <= G && abs(y[i])
   ↪ <= G
3.  E<> C > 0 && P0.turn_random
4.  E<> C > 0 && P0.turn_180
5.  A[] P0.forward imply P0.t <= T_MAX
6.  E<> P0.turn_180 && not gridBoundary(0)
7.  E<> P0.turn_180 && gridBoundary(0)
8.  E<> P0.forward && not turnRandom(0) && not turn180(0)
9.  A[] P0.t != 0 imply P0.forward
10. A[] (P0.turn_random or P0.turn_180 or P0.grid or P0.grid
   ↪  or P0.if or P0.else_if) imply P0.t == 0
11. A[] forall(i : int[0, N-1]) shared_neighbours[i][0] == 0
   ↪  and shared_neighbours[i][1] == 0
12. A[] forall(i : int[0, N-1]) C < 0 imply k[i] == N-1
13. A[] forall(i : int[0, N-1]) C > 0 imply x_dir[i] != 0 or
   ↪  y_dir[i] != 0
14. A[] not deadlock
```

Explanation of properties from Figure 16:

1. For all the paths through the system, no robot can have their x and y coordinates equal to the grid boundary at the same time. No robot will ever reach the corner of the grid.

2. For all the paths through the system, all robots will stay within the grid boundaries.

3. There exists a path through the system, for a robot to reach location `turn_random` after initialization. This means that following locations are reachable: `forward`, `grid`, `if`, `else_if`. It also means that following transitions are reachable: `turn_random` $\rightarrow$ `forward`, `forward`n $\rightarrow$ `grid`, `grid` $\rightarrow$ `if`, `if` $\rightarrow$ `else_if`, `else_if` $\rightarrow$ `turn_random`.

4. There exists a path through the system, for a robot to reach location `turn_180` after initialization.

5. For all the paths through the system, a robot will obey the invariant on location `forward`.

6. There exists a path through the system, for a robot to reach location `turn_180` without reaching the boundary of the grid. This means that transition from `if` to `turn_180` is reachable.

7. There exists a path through the system, for a robot to reach location `turn_180` as a result of reaching the boundary of the grid. This means that transition from `grid` to `turn_180` is reachable.

8. There exists a path through the system, for a robot to reach location `forward` without changing its direction.

9. For all the paths through the system, robot's clock value different than zero implies it being in the `forward` location. This means that time perceived by the robot is only allowed to pass in the `forward` location.

10. For all the paths through the system, a robot's presence in the listed location imply that its clock value is equal to zero. For the robot, time will not pass in any other location than forward.

11. For all the paths through the system, two robots will not share a neighbor. This is a consequence of a fact that in the system consisting of two robots, they cannot have a neighbor in common.

12. For all paths through the system, all robots are fully connected as part of system initialization.

13. For all paths through the system, all robots have their directions set after system initialization.

14. There is no path through the system, which will result in deadlock.

6.2 Verification of synchronized implementation

Figure 17: System specification for synchronized implementation

```
N = 2;      // Number of robots
R = 2;      // Signal radius
STEP = 1;   // Step size
BETA = 1;   // Beta parameter
G = 3;      // Grid boundary
```

Figure 18: Successfully verified properties for synchronized implementation

```
1.  A[] forall(i : int[0, N-1]) x[i] != G or y[i] != G
2.  A[] forall(i : int[0, N-1]) abs(x[i]) <= G && abs(y[i])
    ↪ <= G
3.  E<> (x[0] != 0 or y[0] != 0) and P0.turn_random
4.  A[] (P0.forward and P1.forward) imply C0.n == 0
5.  A[] C0.n == N imply not (P0.forward or P1.forward)
6.  E<> P0.turn_180 && not gridBoundary(0)
7.  E<> P0.turn_180 && gridBoundary(0)
8.  E<> P0.forward && not turnRandom(0) && not turn180(0)
9.  A[] P0.turn_180 imply (P1.forward or P1.grid)
10. A[] forall(i : int[0, N-1]) shared_neighbours[i][0] == 0
    ↪  and shared_neighbours[i][1] == 0
11. A[] forall(i : int[0, N-1]) (x_dir[i] == 0 and y_dir[i]
    ↪ == 0) imply k[i] == N-1
12. A[] not deadlock
```

Explanation of properties from Figure 18:
1. For all the paths through the system, no robot can have their x and y coordinates equal to the grid boundary at the same time. No robot will ever reach the corner of the grid.
2. For all the paths through the system, all robots will stay within the grid boundaries.
3. There exists a path through the system, for a robot to reach location `turn_random` not only as a result of initialization. This means that following locations are reachable: `forward`, `grid`, `if`, `else_if`. It also means that following transitions are reachable: `turn_random` → `forward`, `forward` → `grid`, `grid` → `if`, `if` → `else_if`, `else_if` → `turn_random`. This property is false for a system with signal radius equal to 1.
4. For all the paths through the system, if both robots are in the `forward` location, then the value of the synchronization mechanism must be equal to 0.
5. For all the paths through the system, if value of synchronization mechanism is equal to the number of robots in the system, none of the robots can be in the

`forward` location. 6. There exists a path through the system, for a robot to reach location `turn_180` without reaching the boundary of the grid. This means that transition from `if` to `turn_180` is reachable.
7. There exists a path through the system, for a robot to reach location `turn_180` as a result of reaching the boundary of the grid. This means 8. There exists a path through the system, for a robot to reach location `forward` without changing its direction.
9. For all the paths through the system, if the first robot is in the `turn_180` location, the other has to be in either `forward` or `grid` location. More generally, only one robot can be changing its direction at any given time.
10. For all the paths through the system, two robots will not share a neighbor. This is a consequence of a fact that in the system consisting of two robots, they cannot have a neighbor in common.
11. For all paths through the system, if none of the robots have their direction set they must be fully connected to other robots. This property aims to check the correctness of initialization.
12. There is no path through the system, which will result in deadlock.

6.3 Verification of Beta algorithm - asynchronous implementation

Figure 19: System specification for synchronized implementation

```
N = 3;       // Number of robots
R = 1;       // Signal radius
STEP = 1;    // Step size
BETA = 2;    // Beta parameter
G = 3;       // Grid boundary
```

Figure 20: Successfully verified properties for Beta algorithm, for system specification defined in Figure 19

```
1. E<> exists(i : int[0, N-1]) k[i] == 0 and last_k[i] == 0
2. E<> forall(i : int[0, N-1]) k[i] == 0 and last_k[i] == 0
3. E<> forall(i : int[0, N-1]) C < T_MAX and k[i] == 0 and
    ↪ last_k[i] == 0
```

Figure 21: System specification for synchronized implementation

```
N = 4;       // Number of robots
R = 1;       // Signal radius
STEP = 1;    // Step size
BETA = 2;    // Beta parameter
G = 3;       // Grid boundary
```

Figure 22: Successfully verified properties for Beta algorithm, for system specification defined in Figure 21

```
1. E<> exists(i : int[0, N-1]) k[i] == 0 and last_k[i] == 0
```

Explanation of properties for Figure 20 and Figure 22:
1. There exists a path through the system, for a robot to become disconnected for at least two steps.
2. There exists a path through the system, where all robots become disconnected for at least two steps.
3. There exists a path through the system, where all robots become disconnected for at least two steps with global time of the system not exceeding a time threshold used for invariants.

Property defined in Figure 22 was successfully verified for the system consisting of four robots. UPPAAL generated a diagnostic trace that is a path through the system satisfying the property. It presents a scenario where a single robot gets disconnected for at least two steps. This trace will be used as a starting point to show possible scenarios. In the positive scenario, presented in Figure 23, a disconnected robot is reconnected with the swarm. In the negative scenario, presented in Figure 24, a disconnected robot fails to reconnect with the swarm and may reconnect in the future only due to limited grid area. All the presented scenarios exceeding the trace were obtained using UPPAAL's symbolic simulator, therefore presented states of the system are reachable.

Diagrams present robot swarm system states. The state of the system is represented by robots placed on the two-dimensional grid. A grid is a square with a side of length equal to six. The grid center serves as the origin of the coordinate system. A robot is represented by its position on the grid marked with a dot, direction indicated by an arrow, signal radius visualized by a circle. Numbers inside of the circle indicate a number of robots in a given location. There might be multiple system states in between those presented in the diagram however, they are not crucial for illustrative purposes. States marked with the letter T are part of the trace generated while verifying property from Figure 22.

The scenario described by the diagnostic trace starts with all robots placed in the origin of the coordinate system and without a set direction. Then each robot gets its direction set before moving forward for the first time. The basis for this textual description is the implementation of the Beta algorithm presented in Figure 7. In the diagnostic trace, one robot has its direction set to 'right' while the other three robots have their direction set to 'up'. After the robot moves one step to the right it updates its connection information. It is connected to all of the robots as they are within the reach of its signal. Other robots move up and update their connection information. All of the robots react to the loss of the robot that moved to the right and perform 180 degree turn to change their direction. The robot that moved to the right did not update its connection information and perceives itself as fully connected to other robots. It continues to move to the right, updates its connection information and as a consequence performs a 180 degree turn.

Figure 23 presents diagnostic trace marked with T and one of possible positive scenarios marked with R. In the positive scenario the robot that was moving to the right, performs a 180 degree turn and reconnects with a swarm by moving left. All robots are within their signal range. Another positive scenario could involve one of three robots traveling to the center of the grid to serve as a bridge between the robot that moved to the right and two other robots that moved up.

Figure 23: Trace and reconnection

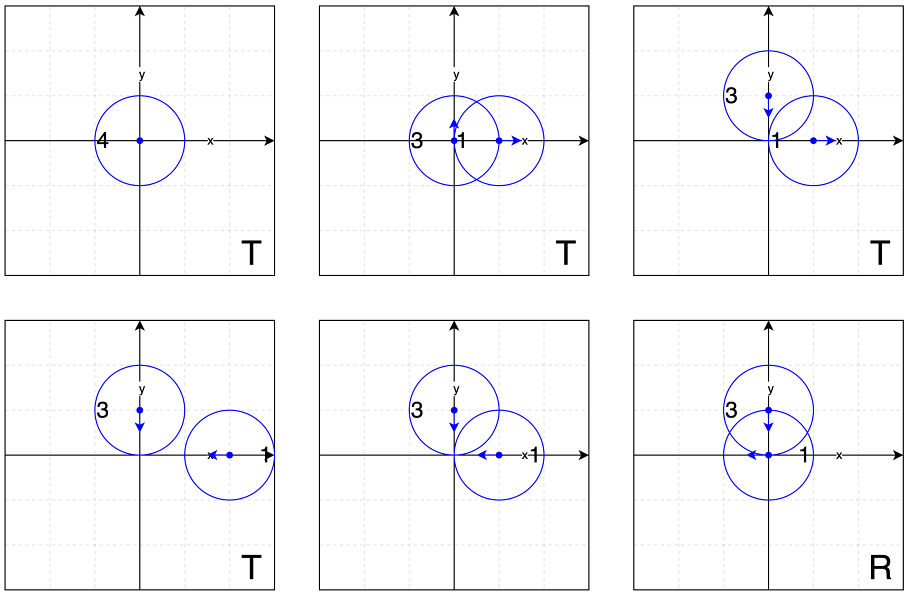
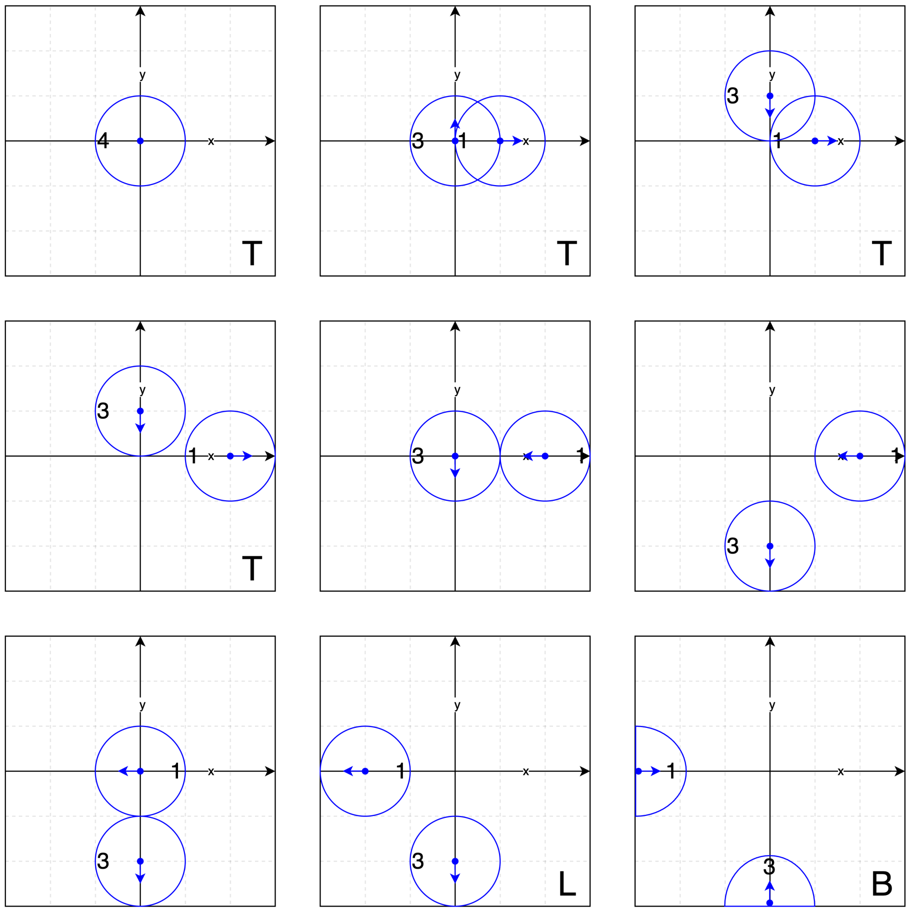


Figure 24 presents a diagnostic trace marked with T, a lost connection scenario marked with L, and the scenario of reconnection due to the grid boundary marked with B. Because the robots move asynchronously, it is possible for three robots that initially moved up to move down before the other robot makes a single move toward the center. This behavior is shown in the middle row of the diagram. After that, the robot that was on the right would traverse to the left side. If the grid was not limited, the robot would become disconnected forever, which is shown in the L scenario. Because the grid is limited and robots will perform 180 degree turn it is possible for them to reconnect as the result of reaching the boundary, which is shown in the B scenario.

Figure 24: Trace and lost connection



6.4 Verification of Beta algorithm - synchronized implementation

Figure 25: System specification for synchronized implementation (all combinations of N and R)

```
N = 3, 4;    // Number of robots
R = 1, 2;    // Signal radius
STEP = 1;    // Step size
BETA = 2;    // Beta parameter
G = 3;       // Grid boundary
```

Figure 26: Successfully verified properties for Beta algorithm

```
1. E<> exists(i : int[0, N-1]) k[i] == 0
2. E<> forall(i : int[0, N-1]) k[i] == 0
```

Explanation of properties from Figure 26:
1. There exists a path through the system, for a robot to become disconnected for a single step.
2. There exists a path through the system, where all robots become disconnected for a single step.

Figure 27: Falsified properties for Beta algorithm

```
3. E<> exists(i : int[0, N-1]) k[i] == 0 and last_k[i] == 0
4. E<> forall(i : int[0, N-1]) k[i] == 0 and last_k[i] == 0
```

Explanation of falsified properties from Figure 27:
3. There exists a path through the system, for a robot to become disconnected for at least two steps.
4. There exists a path through the system, where all robots become disconnected for at least two steps.

7 Discussion

Verification of the Beta algorithm for the asynchronous implementation showed that the algorithm coherence is not guaranteed for the Beta parameter equal to two. That value was used in the original paper [9] however, other values of the Beta parameter were used but their influence was not examined. For swarms of size three, it was shown that all of the robots can become disconnected for at least two steps. For swarms of size four, it was shown that a single robot can become disconnected for at least two steps. The diagnostic trace produced upon verification of property defined in figure 22, was used as a starting point for manual traversal through possible system states using UPPAAL's symbolic simulator. This led to the exploration of positive and negative scenarios for the swarm presented in Figures 24 and 23. Due to hardware limitations, it was not possible to verify whether a swarm consisting of four robots could become fully disconnected for at least two steps.

While the swarm will react to the formation of bridges described in the original paper [9], it will not prevent them from forming. This can be observed in the scenario presented in Figure 24. This can be partially attributed to the concurrency mode used by the system. Property that describes the situation of a single robot becoming disconnected from the swarm for at least two steps was verified as true for the system operating in asynchronous mode and falsified for the synchronized system. By examining scenarios presented in Figures 24 and 23 one can speculate that robots that operate in similar frequency are more likely to reconnect. If robots operated in a more uniform way it could be easier for them to reconnect. In the situation of asynchrony, a group of robots may move away before the lost robot realizes that it is disconnected.

When verifying properties of the Beta algorithm for the synchronized system it was found that all of the robots can become disconnected for a single step. However, it was also found that a single robot can't become disconnected for two or more steps. Properties that were verified and falsified can be found in Figures 26 and 27. It can be argued that one of the reasons for this outcome is the reactivity of the system in which all robots move forward at the same time. This could be treated as an indication that the uniformity of robot operations might be a crucial aspect of the algorithm.

There were several limitations affecting the verification process and results, the most significant being the state-space explosion problem. A swarm of size four produced too many states for full verification on a single laptop. To make verification possible, the grid was bounded, and the robot algorithm modified: robots performed a 180-degree turn upon reaching the grid's boundary. This altered the original algorithm and affected its effectiveness, as robots could reconnect by bouncing off boundaries. As a result, defining a property to detect a disconnected robot was straightforward unlike defining one for a connected swarm that did not become connected as a result of interacting with the boundary.

8 Conclusions

In this paper, we have presented the process of modeling, implementing, and verifying the Beta algorithm for a robot swarm using timed automata and UPPAAL. Through the implementation and verification of both asynchronous and synchronized implementations, we highlighted the influence of the concurrency mode on the algorithm's effectiveness in maintaining swarm coherence. While the Beta algorithm demonstrated responsiveness to lost connections, challenges emerged in guaranteeing reconnection in the asynchronous system. Furthermore, the reconnection of the swarm cannot be attributed solely to the Beta algorithm, as it is also influenced by the limited area of the grid. These insights highlight the difficulties in verifying swarm robotic algorithms and the influence of robot uniformity on the effectiveness of the algorithm.

The results of our verification showed that the synchronization mode significantly affects the coherence of the swarm. Synchronized implementations completely eliminated prolonged disconnections. While this demonstrated the importance of uniformity in robot behavior for maintaining connectivity, a fully synchronized solution defeats the most basic assumption of swarm robotics: decentralization. In contrast, asynchronous systems, while aligning more closely with real-world scenarios, faced challenges with timely reconnection. Additionally, the bounded grid environment introduced limitations that, while necessary for computational feasibility, influenced the outcomes of the algorithm. This highlights the need to balance practical constraints with algorithmic integrity when implementing and verifying swarm robotics algorithms.

Future work should focus on developing additional properties to verify the Beta algorithm, ensuring a more comprehensive understanding of its behavior under various conditions. Exploring other approaches to limiting the grid, such as implementing different boundary behaviors or using varying grid sizes, could provide more flexibility in verification while maintaining computational feasibility. Additionally, verification should be expanded to larger swarms with different parameter values, such as varying the Beta parameter and robot count, to assess the algorithm's scalability. Furthermore, a new mode could be explored where robots are not fully synchronized but operate at the same frequency, balancing the benefits of both presented implementations. This would provide valuable insights into the algorithm's effectiveness.

References

[1] David Alexandre, Amnell Tobias, and Stigge Martin. Uppaal 4.0 : Small tutorial. `https://uppaal.org/texts/small_tutorial.pdf`, 10 2009. Accessed: 2024-08-27.

[2] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP '90, page 322–335, Berlin, Heidelberg, 1990. Springer-Verlag.

[3] Laura Antuña, Dejanira Araiza-Illan, Sérgio Campos, and Kerstin Eder. Symmetry reduction enables model checking of more complex emergent behaviours of swarm navigation algorithms. In Clare Dixon and Karl Tuyls, editors, *Towards Autonomous Robotic Systems*, pages 26–37, Cham, 2015. Springer International Publishing.

[4] Johan Bengtsson and Wang Yi. *Timed Automata: Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[5] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 7:1–41, 03 2013.

[6] Clare Dixon, Alan Winfield, and Michael Fisher. Towards temporal verification of emergent behaviours in swarm robotic systems. In Roderich Groß, Lyuba Alboul, Chris Melhuish, Mark Witkowski, Tony J. Prescott, and Jacques Penders, editors, *Towards Autonomous Robotic Systems*, pages 336–347, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] Panagiotis Kouvaros and Alessio Lomuscio. Verifying emergent properties of swarms. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[8] Kim Guldstrand Larsen, W. P. Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1, 1997.

[9] Julien Nembrini, Alan Winfield, and Chris Melhuish. Minimalist Coherent Swarming of Wireless Networked Autonomous Mobile Robots. In *From Animals to Animats 7: Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior.* The MIT Press, 08 2002.

[10] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12:115–116, 1981.

[11] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer Berlin, Heidelberg, 12 2012.

[12] Melanie Schranz, Martina Umlauft, Micha Sende, and Wilfried Elmenreich. Swarm robotic behaviors and current applications. *Frontiers in Robotics and AI*, 7, 2020.

[13] Ali Narenji Sheshkalani and Ramtin Khosravi. Verification of visibility-based properties on multiple moving robots in an environment with obstacles. *International Journal of Advanced Robotic Systems*, 15(4):1729881418786657, 2018.

[14] Alan FT Winfield, Jin Sa, Mari-Carmen Fernández-Gago, Clare Dixon, and Michael Fisher. On formal specification of emergent behaviours in swarm robotic systems. *International Journal of Advanced Robotic Systems*, 2(4):39, 2005.