Title
Research Project - KIREPRO1PE

Szymon Gałecki - sgal@itu.dk

July 2024

Contents

1 Background

1.1 UPPAAL

UPPAAL is a comprehensive tool for modeling, simulation and verification of real-time systems that can be modelled as a network of timed automata. Synchronisation within such system can be realised with shared variables or communication channels. The tool is typically used to implement and verify controllers and communication protocols. [6]

Swarm of robots fits within description of a system that can be modelled using UPPAAL. It is a collection of non-deterministic processes - single robots, that are defined using finite control structure, an algorithm that is transformed into a finite state machine. Swarm has to communicate and it can be achieved using channels or shared variables. Finally the emergent behaviour is a result of individual robot behaviour which is influenced by the passing time.

1.2 Timed automata in UPPAAL

Timed automata in UPPAAL is based on timed automata defined by Rajeev Alur and David Dill in [1], defined in Definition 1.1. UPPAAL builds on top of that definition extending the states of the automata with invariants. Invariant is a condition on the clock which controls how long an automaton can remain in the given state. Edge between states can be decorated with guard, synchronisation action, clock resets and variable assignments. Guard is a condition on the clock or integer variables that needs to be satisfied in order for the transition to be enabled. Synchronisation action is sending a signal to corresponding edge(s) or waiting to receive such a signal in order to synchronise transitions. Clock reset and variable assignment are used to update the state which will be used for edge conditions and determine further transitions. [6]

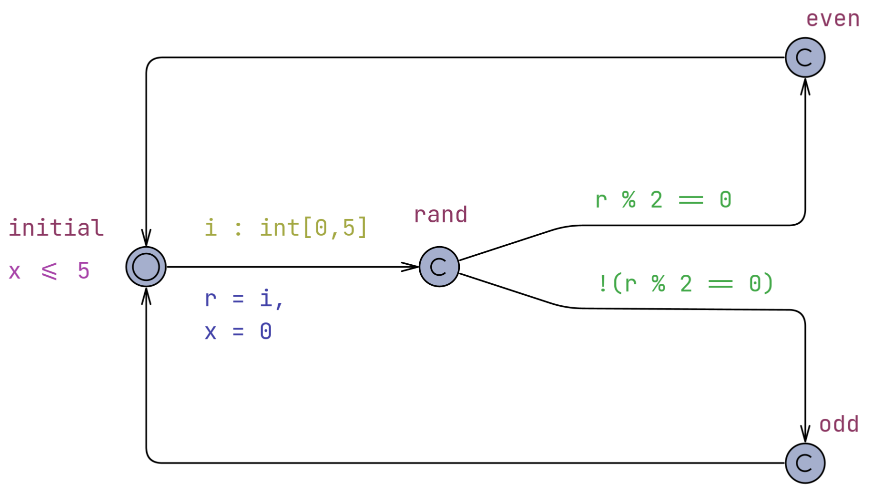
Definition 1.1 (Definition of timed automaton [1])**.** A timed automaton is a tuple (Σ, S, S_0, C, E) where:
Σ - input alphabet;
S - finite set of automaton states;
$S_0 \subseteq S$ - set of start states;
C - finite set of clocks;
$E \subseteq S \times S[\Sigma \cup \epsilon] \times 2^C \times \Phi(C)$ - 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;

1.3 Modeling in UPPAAL

To explain the process of modeling and verification using UPPAAL, we introduce an example model, presented in Figure 1. Description of an example model will allow the reader to understand the implementation section of this work.

Figure 1: An example UPPAAL model



An example model generates a random integer from the range $[0, 5]$ and classifies it as either an odd or even number, at most every five time units. The model has four locations, namely, initial, rand, even and odd. UPPAAL specifies three types of locations, namely, initial, urgent and committed. Initial location is the starting point of the model. Each model can only have one initial location and in the case of the example model it has the same name as type. If location is of type urgent, the time stops progressing and outgoing transition must be taken immediately. Committed location has all the properties of the urgent location but disallows other models to transition as long as the model is in the committed location. Committed locations are used to build atomic sequences of transitions that cannot be interleaved by other models. Such sequence is created by locations rand, even and odd being of a committed type. The initial location has an invariant on the clock x. The model can remain in the initial location at most five time units. Upon transition from the initial location the clock x is reset. This is achieved by the label associated with the edge that is used for transition. Edge has four possible labels, namely, select, guard, sync and update. Select is used to randomly choose a value from defined range. Guard holds a logical condition that needs to be satisfied in order to enable transition. Sync is used for synchronisation between models. Synchronised edge is either waiting for the synchronisation signal to enable the transition or sending it when taking transition. Update is used for integer variable assignment. Transition from the initial location to rand involves selecting integer from the range at random, assigning it to variable and resetting the clock. Edges going from location rand have defined guards, logical conditions that enable transitions. Edge going to location even has condition that checks if the integer drawn at random is even. Edge going to location odd has a condition that is a negation of the condition defined for the edge going to location even. In this way we can guarantee that there will always be only one transition enabled from location rand.

1.4 Verifying properties in UPPAAL

UPPAAL uses timed computation tree logic to verify properties of the networks of timed automata [2]. Properties in UPPAAL are specified using logical quantifiers presented in Definition 1.2.

Definition 1.2 (Logical quantifiers in UPPAAL [2])**.** 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 check the correctness of implementation of the example model, presented in Figure 1, we positively verified following properties.
- `A[] not deadlock` - model never reaches deadlock.
- `A[] (P.rand || P.odd || P.even) imply P.x == 0` - model in location rand or odd or even, implies that time does not progress.
- `A[] P.even imply (P.r == 0 || P.r == 2 || P.r == 4)` - model in location even, implies that the integer selected at random is equal to 0 or 2 or 4.
- `A[] P.initial imply P.x <= 5` - model in location initial, implies that elapsed time is smaller or equal than five time units.

2 Methods

In the literature there are multiple approaches for verification and testing of implemented swarm behaviours. There are several works which document algorithms for those behaviours with the related experiments and results [5], [4], [3]. In some cases there is an automaton corresponding to the textual description of behaviour or pseudocode [5]. At times, used logic and tools are mentioned [3]. However we are not be able to reproduce the experiments because none of the works explained in detail how the chosen swarm behaviour was modelled. That is why this section will focus on explaining how algorithm's pseudocode and general assumptions are transformed into a functioning model. The selected algorithm to demonstrate the process of modelling is Alpha algorithm. It was chosen because there were many experiments that addressed Alpha algorithm but omitted the details of what was actually being tested or verified [4], [9].

2.1 Alpha algorithm

Alpha algorithm was introduced by Julien Nembrini in [7]. It was inspired by Kasper Støy's work [8]. Støy proposed and implemented a simple control system for aggregating robots. Instead of relying on environment and localisation information, it uses physical properties of the signal used for communication. Robot behaviour is solely determined by the change in the number of robots that are in the range of its signal.

Alpha algorithm is an approach to an aggregation task within the category of spatial organisation. It is based on the assumption that robots send and receive signals through omnidirectional channels like radio or infrared. Single robots make decisions about their movement only based on the number of connections to other robots. The inter-connectivity of the swarm is controlled by the alpha parameter which is a threshold on the desired number of connections for a single robot. Pseudocode defining the Alpha algorithm can be found in Figure 2. Variable **i** in the pseudocode is a loop iterator and **cadence** is a parameter that controls how often a robot will send its ID and check the number of neighbours.

In order to explain the Alpha algorithm in a greater detail we will divide the most important aspects of the robot behaviour into subsections. First three subsections, namely, Movement, Connection, Initialisation and clocks, will explain the algorithm's general assumptions and design choices. The fourth and last subsection - Automaton, will transform the pseudocode for the Alpha algorithm into timed automaton while incorporating design choices introduced in previous subsections.

Figure 2: Pseudocode for Alpha algorithm from [7]

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

loop forever {
        i = i modulo cadence

        if (i = 0){
                Send ID message

                Save copy of k in LastK
                k = number of neighbours in Nlist

                if ((k < lastK) and (k < alpha)){
                        turn robot through 180 degrees
                }
                else if (k > LastK) {
                        make random turn
                }
        }

        Steer the robot according to state
        Listen for calls from robots in range
        Grow Nlist with neighbours IDs

        i++
}
```

2.2 Movement

There would be no need for Alpha algorithm without the robot movement so it is important to define how it is modelled. Robot always moves in one of four directions: up, right, down or left. Initial direction is chosen at random and mapped to vertical and horizontal components. Direction will not change unless the robot performs random turn or 180 degree turn. Random turn chooses new direction in the same way that initial direction is determined. This means that random turn may result in maintaining the current direction of the robot with approximately 25% chance.

Robot movement is achieved by incrementing the robot's coordinates with vertical and horizontal direction components. Direction component values are $\in \{-1, 0, 1\}$, but either vertical or horizontal direction component has to be equal to zero. This means that robot will move in one of four possible directions with a step size equal to one. Every time the robot moves, it will update its coordinates in the globally available data structure. Environment in which robot exists is an unbounded grid that can be continuously traversed in four directions.

Robots are not aware of other robots positions. This means that they can occupy exactly the same point of the grid. Additionally, there is no collision avoidance mechanism that would prevent them from crashing into each other while moving. This simplification could be eliminated but at the cost of implementation being further apart from the original idea of what is an Alpha algorithm. We would have to make arbitrary decisions about robot behaviour in the case of collision or inability to move to the occupied point on the grid.

2.3 Connection

Number of robot connections is the main parameter determining the robot behaviour. Therefore it is important to accurately model what it means for robots to be connected. One of the assumptions in Julien Nembrini's [7] is that physical signal is omnidirectional. However, robot movement is modelled to be two dimensional. Consequently we will assume that two robots are connected if their distance is smaller than radius of physical signal that we mimic. Length of the radius will determine how far apart the robots can move before losing connection.

To model the physical signal we need information about robot coordinates, mutual distances, previous and current number of neighbours. This allows for evaluating whether robots are in the signal distance and therefore connected or not. Every time robot moves it will update its own coordinates, previous number of neighbours for itself, current number of neighbours for all robots and mutual distances for all robots. It may seem like the robot is using more information than it is assumed in the Alpha algorithm. Information that it passes is used to mimic the physical signal of the real robot. Although, it updates its coordinates, robot is not aware of the position of its neighbours. Its movement is derived solely from information about previous and current number of neighbours.

The desired number of connections is set by the alpha parameter. If the number of connections falls below alpha, robot will react and turn 180 degrees in order to try to reconnect with other robots. Alpha parameter influences connectivity of the swarm. If the parameter is set to higher values the robots will try to maintain more connections and the resulting swarm will be more compact. If the parameter is set to lower values, robots will be able to move further apart as they will need to maintain less connections. What is low and high value for alpha parameter is subjective to the size of the swarm.

2.4 Initialisation and clocks

This subsection explains the initial state of the robot and the influence that the time has on its behaviour. Every robot gets initialised with the same set of parameters apart from its ID. In the beginning all robots are placed at exactly the same point. They have no direction until transitioning to the next state - RANDOM STATE, and will not move forward until transitioning further. In the INITIAL state robots cannot access information about number of current neighbours and number of last neighbours. The INITIAL state becomes unreachable after the robot transitions from it.

Pseudocode variables **i** and **cadence** model the frequency of robot actions. Specifically **cadence**, it controls how often a robot sends a signal and checks if it should change direction. While we are able to model such frequency without a timed automaton we would not be able to model a real-time system like robot swarm. The timing aspect allows us to model a system with time dependent behaviours and to verify a time dependent properties. We can observe a system behaviour throughout time and evaluate the stability of the implemented algorithm. That is why each robot has its own clock that controls how long it can remain in the current state before being forced to transition. After each transition to the next state the robot will reset its own clock. Maximum time for the robot to remain in state can influence the behaviour of the whole system. If we decrease the maximum time of inactivity, we will obtain a system that is on average more reactive and even. With increasing the maximum time of inactivity we increase the chances of robots operating in different pace. As we are modelling swarm we should aim for uniform operation and therefore lower maximum times for inactivity.

2.5 Automaton

In order to implement the Alpha algorithm presented in [7], we utilised the pseudocode from Figure 2, with design choices described in the previous subsections. The implementation is a timed automaton - Figure 3.

We will describe how pseudocode was mapped into the timed automaton. Pseudocode starts with creating and initialising variables that are needed for algorithm execution. This part is not explicitly visible in the automaton but achieved during initialisation of the model.

Pseudocode variables that represent the state:
- **Nlist**, a list of neighbours for a robot;
- **ID**, a unique robot identifier;
- **k**, a number of neighbours in Nlist;
- **lastK**, a previous number of neighbours in Nlist;
- **alpha**, a threshold on the number of connections;

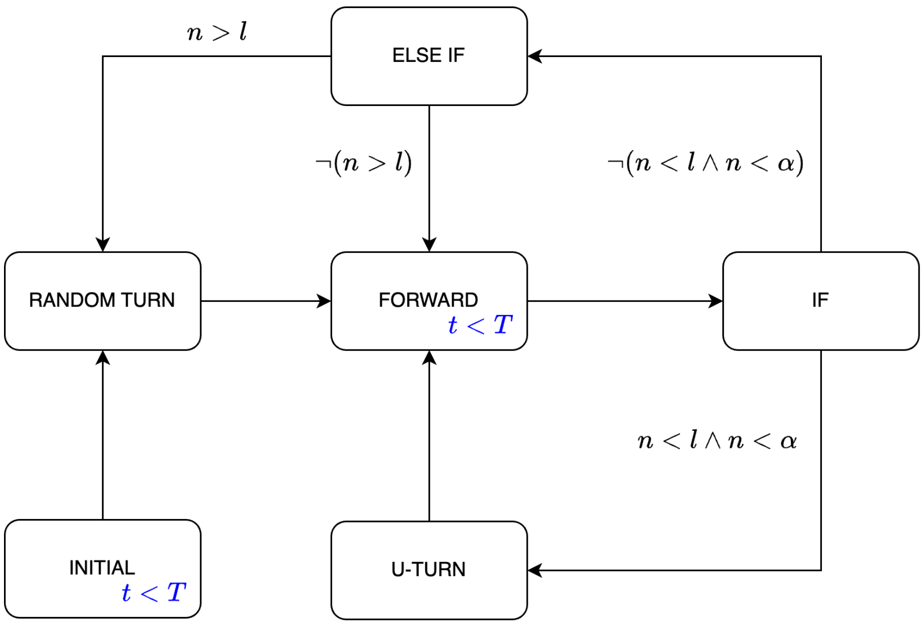
The state of the algorithm before entering the infinite loop is represented by the INITIAL state. Transitioning from the INITIAL state is equivalent to entering the infinite loop defined in the pseudocode. The entry point for the loop is a RANDOM TURN state where the robot obtains its initial random direction. Infinite loop defines that robot will send its ID and make a decision about turning only every 'cadence' loop iterations. To mimic this behaviour cadence was implemented as the maximum time of inactivity T. In the pseudocode decision about turning is realised with 'if' and 'if else' statements. Statements and their corresponding conditions were transformed to IF and IF ELSE states with conditions guarding subsequent transitions. Guarded transitions utilise original conditions and their negations. In this way we can guarantee that there is always only one transition possible at any point. Variables used in the conditions correspond to the current number of neighbours, previous number of neighbours and desired number of neighbours. Those variables control the behaviour of a single robot alongside the maximum time of inactivity T which acts as a cadence parameter of the loop. Single robot is represented by the timed automaton presented in Figure 3. Composition of such timed automata results in a robot swarm implementation of the Alpha algorithm. The movement and interactions of robots cause them to update their variables. Those variables are then used to determine the logical value of conditions guarding the state transitions. Finally, the available transitions are taken, influencing the robots behaviours. This process restarts and lasts indefinitely as the system composed of timed automata has no final state.

State descriptions:
- INITIAL, where the robot is not moving and has no direction;
- RANDOM TURN, where the robot randomly chooses one of four directions;
- U-TURN, where the robot changes direction by 180 degrees;
- FORWARD, where the robot moves forward in the set direction;
- IF, where the robot checks if it is disconnected from neighbours;
- ELSE IF, where the robot checks if it is connected to more than desired number of neighbours;

States are not enough to define the behaviour if the robot is not forced to transition. Each robot has its own clock that controls how much time it can remain in a given state. Time - t a robot can remain in states FORWARD and INITIAL is limited by the maximum time T. Robot has to transition before time t is equal to T. After leaving mentioned states, the robot resets its clock. In remaining states the time doesn't pass and therefore time t is not incremented. However other transitions are guarder by associated conditions. Transition from one state into another can only happen if the corresponding condition is satisfied. If there is no condition over the edge, transition can always be taken.

Conditions controlling the robot are associated with the following variables:
n - current number of neighbours;
l - previous number of neighbours;
α - Alpha parameter, desired number of neighbours;

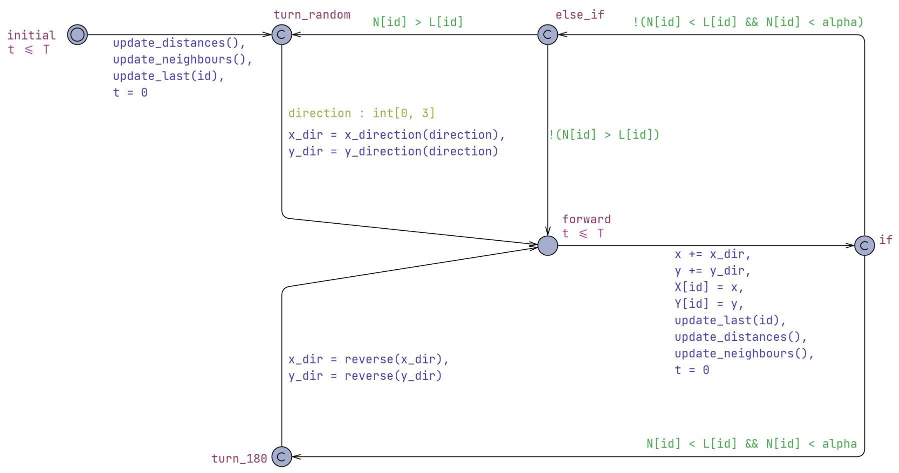
Figure 3: Timed automaton for the Alpha algorithm



3 Implementation

In the previous section, we introduced the Alpha algorithm and taken assumptions which were transformed into a timed automaton. The description provides the details to implement the Alpha algorithm with any suitable tool of choice. In this section, we will focus on implementation with regards to the specific tool of our choice, UPPAAL. UPPAAL is one of the major tools for model checking and verification of real-time systems that can be modeled as timed automata. As the previous section finished on timed automaton, we will start by presenting timed automaton implemented in UPPAAL. We will draw connections between them as presented in Figure 3 and Figure 4. Then, we will explain global functions and state.

Figure 4: UPPAAL implementation of the timed automaton



Variables of the timed automaton presented in Figure 4:
id - unique identifier of the robot;
x_dir - horizontal direction component;
y_dir - vertical direction component;
alpha - alpha parameter;
T - time threshold for inactivity;

To start with comparison between general automaton and its specific implementation in UPPAAL, let us notice that both automata have the same set of states and transitions. Guarded transitions use the same conditions. Each robot accesses its own set of condition variables by providing **id** to the global arrays. Array **N** holds the number of neighbours for each robot. Array **L** holds the last number of neighbours for each robot. Those variables are held in the global arrays so that a single robot could update the number of neighbours for all of the other robots every time it moves. It may seem counter intuitive that a single robot updates the variables of other robots. However, the execution of the system composed from multiple robots is based on interleavings. At any given time a single robot transitions between states. This means that only the moving robot is guaranteed to have true information about number of connections. Global updates of the number of neighbours is a way to mimic physical, continuous signal that would determine whether robots are connected or not. It eliminates the situation where robot would make a decision on already outdated information. The implementation assumes that each time a robot moves it will update its coordinates, update the last number of neighbours for itself and update the current number of neighbours for all robots. Use of the coordinates is necessary as the state of connection is determined by the Euclidean distance between robots and radius of the assumed communication technology. The update of the number of neighbours is achieved by calling **update_last(id)** and **update_neighbours()**. Figure 5 shows the process of updating the number of connections for all robots.

Figure 5: Implementation of **update_neighbours()** and **neighbours()**

```
int neighbours(int id){
    // returns the number of connections for a robot with a given id
    int x = X[id];   // x coordinate
    int y = Y[id];   // y coordinate
    int k = 0;       // neighbour counter k
    int i = 0;
    for (i = 0; i < n; i++){
        double d = distance(x, y, X[i], Y[i]);
        bool c = connected(d);
        // is robot connected to another robot
        if(i != id && c){
            k++;
        }
    }
    return k;
}

void update_neighbours(){
    // updates the array that stores number of neighbours for each robot
    int i;
    for (i = 0; i < n; i++){
        N[i] = neighbours(i);
    }
}
```

The number of connections will then determine the next transitions. Transition that is forced by the clock in the **forward** state will always result in the robot moving by a unit step in its current direction. Then, based on logical value of the conditions guarding transitions, it can also change its direction. The direction will change after robot reaches and transitions from states **turn_random** or **turn_180**. The random turn is realised as a random choice between four directions: up, right, down, up and translating it into x, y coordinates. The 180 degree turn is realised as choosing the opposite direction to the current one. Change of direction is achieved by functions **x_direction(direction)**, **y_direction(direction)**, **reverse(direction)**. Figure 6 show the process of changing direction.

Figure 6: Implementation of **x_direction(direction)** and **reverse(direction)**

```
int x_direction(int direction){
    // maps direction to x axis
    if (direction == 1){
        // right
        return 1;
    }
    if (direction == 3){
        // left
        return -1;
    }
    // neither
    return 0;
}

int reverse(int direction){
    // reverses direction
    return direction * -1;
}
```

The last element of the implementation is time. Each robot has its own clock that limits how long it can stay in the states with defined invariants: **initial**, **forward**. After transitioning from those states, a robot will reset its clock. Only in those states, the time passes. Other states are so called committed locations. In committed locations time does not pass which means that individual robot clocks do not progress. As a consequence the time progresses during the forward motion of the robot but decision to turn happens instantaneously.

The implementation of the Alpha algorithm in UPPAAL and the presented timed automaton model depct the behavior of a single robot. In order to create a robot swarm we need to instantiate multiple such robots and compose them into a system. This is achieved in UPPAAL with code presented in Figure 7.

Figure 7: Composing multiple robots into a system in UPPAAL

```
// id, x_dir, y_dir, alpha, T
P1 = Robot(0, 0, 0, 1, 5);
P2 = Robot(1, 0, 0, 1, 5);
P3 = Robot(2, 0, 0, 1, 5);
system P1, P2, P3;
```

We started from general timed automaton implementing the Alpha algorithm. We implemented it using UPPAAL and recreated states, transitions and conditions. We explained the reasoning behind using the global state for condition variables and the mechanism of updating them. We showed how the robot movement and changes of directions are realised and when it happens. Finally we described the timing aspect and its practical implications on the behaviour of the model.

References

[1] 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.

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

[3] Manuele Brambilla, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. *Property-driven design for swarm robotics*, pages 139–146. 01 2012.

[4] 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.

[5] Savas Konur, Clare Dixon, and Michael Fisher. Formal verification of probabilistic swarm behaviours. In Marco Dorigo, Mauro Birattari, Gianni A. Di Caro, René Doursat, Andries P. Engelbrecht, Dario Floreano, Luca Maria Gambardella, Roderich Groß, Erol Şahin, Hiroki Sayama, and Thomas Stützle, editors, *Swarm Intelligence*, pages 440–447, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[7] Julien Nembrini, Alan Winfield, and Chris Melhuish. Minimalist coherent swarming of wireless networked autonomous mobile robots. 09 2002.

[8] Kasper Stoy. Using situated communication in distributed autonomous mobile robotics. pages 44–52, 01 2001.

[9] 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.