Swarm Robotics - Modeling and statistical property checking of a decision making algorithm

Andreas Riget Bagge (anrb),
Marcus Skjold Pedersen (skmp),
Peter Klarskov Døssing (pekd),

Supervisor: Mahsa Varshosaz (mahv)

May 15, 2025

1 Abstract

This paper presents modeling and property checking of a swarm robotics consensus algorithm devised by Lee and Liu [*SN Applied Sciences*, 2, 2019]. The algorithm poses an interesting modeling challenge due to its constraints on local knowledge and limited communication, for which we present two variants as solutions. This is done in the model-checking tool UPPAAL[1], by modeling the swarm as a network of probabilistic timed automata, and using the features of UPPAAL SMC for doing statistical model checking on swarms with a large state space. The model's performance and quality is then compared to the results of the original paper, based on the results of checking formalized properties and the simulations of bounded random runs on the model. We discuss the strengths of UPPAAL as a modeling tool, as well as considerations related to the tool's documentation, performance and ease of use in collaborative work. Finally, we discuss the metrics for the Lee and Liu algorithm and the difficulty of evaluating the quality of an algorithm and the subsequent model.

[1]https://uppaal.org/

Contents

2 Introduction

Swarm robotics is a field of robotics concerned with the use of multiple simple and often identical robots to accomplish complex tasks by virtue of collaboration. As these robots are simple, it is often easy to verify the operation of a single unit, but as swarms increase in size, the emergent behavior becomes unfeasible to manually analyze and predict. Instead, researchers and practitioners in swarm robotics can use a variety of tools to analyze the swarm. This is usually done in one of three ways: Through real robot implementations, computational simulation, or formal verification (Konur et al., 2010). In this project, we will be using the modeling tool UPPAAL to model and analyze a swarm robotics algorithm using simulation and statistical model checking. The work chosen for implementation is a consensus decision making algorithm made by Yang Liu and Kiju Lee (Liu & Lee, 2019). As a swarm robotics algorithm, it is constrained to only make use of local communication between simple robots, while it must guarantee that it results in global consensus. Solving this constraint in modeling is one of the main areas of focus of this research paper. Additionally, the algorithm proposes a number of expected metrics that are measurable via our verification tools, and thus create benchmarks for our model implementation. We will use this groundwork to answer the following research questions:

2.1 Research Questions

- To what extend can we model the Lee-Liu algorithm, which relies on inter-robot communication and network/graph knowledge in the individual agent, in a modeling tool?
- To what extend can the model of probabilistic timed automata capture the characteristics of the algorithm?
- Which properties of the algorithm can be formalized into properties and checked by the model checking tool?
- Do the number of iterations needed to reach a consensus in our model conform to the theoretical estimation in the paper?

In section 3, we will introduce the modeled algorithm, and the modeling tool UPPAAL. In section 4, we will explain how we approached modeling the algorithm and formalizing properties for verificationof our model specifications. In section 5 we present the process of verifying properties and the results stemming from these. Lastly, in section 6, we will discuss the strengths and challenges of using UPPAAL as a modeling tool, and some of the challenges posed by ambiguities in the specification of the decision-making algorithm and its metrics.

3 Background

3.1 Overview of algorithm

Our research project and UPPAAL model are based on the work by Yang Liu and Kiju Lee ((Liu & Lee, 2019)). The article describes an algorithm for reaching a consensus in a decision-making process in an arbitrarily large swarm of independent, autonomous robots or agents that work together to accomplish some task. The algorithm was selected because it poses an interesting challenge in modeling the communication between agents, and because it suggests a number of experimentally decided metrics for how the algorithm should work. These metrics are useful for us to compare against our own model implementation.

The authors establish the following constraints on the algorithm (Liu & Lee, 2019, section 2):

- Individual robots are primitive with limited sensing, communication, and processing capabilities.
- Communication in the swarm is local; each robot can communicate only with nearby robots within the communication range. Robots within range will be referred to as neighbors for the rest of this report.
- Robots have no temporal memory (i.e. no log of history data) and function like finite state machines.

We also note three assumptions:

- The topology of the network will not change during the decision-making process (ibid.).
- Robots have perfect and consistent communication among their neighbors.
- Robots are assumed to be somewhat globally synchronized, as the network is evaluated in terms of iterations.

The assumption of a fixed topology was also a deciding factor for us in choosing the algorithm, as we expected that not having to model physical movement of robots would make it more possible to make a model capturing the actual behavior of the robots.

Apart from these assumptions that were explicitly stated in the original article, it is mentioned that the individual robot updates the members of its local consensus group, when the group changes. It is not specified how these updates are handled. As will be seen in our modeling section, handling these changes under the assumptions of the algorithm is not trivial.

In summary, the algorithm works as follows:

- There are m robots, identified by their index in the set R. The set of possible decisions, $\mathbf{Q}$, contains the indices for n different choices.
- Each individual robot is initialized with a distribution of preferences for each possible choice, given as a probability mass function P_k for robot R_k. Each robot exhibits a decision, which is the choice it has the highest preference towards. For example, if $n = 3$, R_k may have the set of preferences $\{P_k(1) = 0.25, P_k(2) = 0.5, P_k(3) = 0.25\}$, and it will exhibit a decision of choice 2.

- A robot has a connection group, C, which consists of all of its neighbors. The robot knows the IDs and preference distributions of its neighbours.
- A robot has a consensus group, D, which consists of the largest connected component of robots that share the same dominant preference. See figure 1 below. A robot knows the IDs, but not the preference distributions, of the members in its consensus group.
- A robot continuously updates its preference distribution by exchanging information with its neighbours. A robot will update its preferences to align with the predominant opinion among its neighbours, especially if their consensus groups are large.
- As the algorithm runs, the robots will become more certain in their decisions and stabilize their preference distributions, as long as nothing new is introduced in the network.
- When all robots share the same dominant preference, the algorithm terminates.

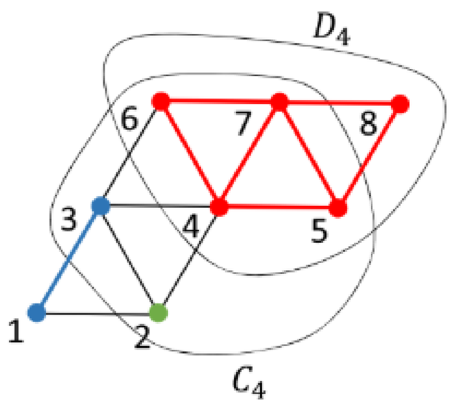


Figure 1: "A network showing R_4's local consensus group D_4 and local connection group C_4" (Liu & Lee, 2019, Fig. 2).

For a detailed explanation of the algorithm, we refer the reader to the original article (Liu & Lee, 2019).

3.1.1 Expected behaviour and results

Lee and Liu present a number of different expected outcomes of their algorithm, as well as some ideal and anticipated behaviour under different circumstances based on both theoretical estimations and simulations done in a python script. These expectations are important benchmarks for us to compare against our own findings in our UPPAAL implementation of the algorithm. Here, we list some of the findings that we will discuss further in the results section of this paper.

1. The network always reaches a consensus.

This property of the algorithm serves as an important sanity check for us to determine that the model acts as intended. It is also the first property that we will check with our modeling tool.

2. The convergence rate is moderately correlated with network dependency.

Network dependency is a term coined by Lee and Liu in the research article. Network dependency denotes the importance of the most critical node on maintaining the network connectivity (Liu & Lee, 2019, page 6). To facilitate testing the properties, we have coded a model generation tool in python that can also determine the network dependency value of the networks we use in simulation. We elaborate on this in section 5.1.

3. The number of iterations needed to reach consensus is positively correlated with network size m and the number of decisions n.

Larger networks and networks with a larger decision space take longer to correlate. We expect to see similar behaviour in our UPPAAL implementation. Next follow a number of tests that we will try to replicate in our model, although the results are somewhat difficult to exactly reproduce, as they are based on randomly generated networks and their specific topologies.

4. The network converges significantly faster when seeded. The size of the seeding group and its placement in the network significantly affects this property.

With 10 seed robots, a network of a 100 robots exhibits a 53% convergence rate towards the dominant opinion of the seed robots. We will create similar trials to see if we can observe similar behaviour in our model.

Apart from these assumptions that were explicitly stated in the original article, it is mentioned that the individual robot updates the members of its local consensus group. However, the provided explanation does not exhaustively address the issue that arises when one such change happens at a critical node in the local network of the consensus group. This, however, has become apparent as a result of our work in modeling the algorithm.

3.2 Modeling, verification and properties

As mentioned in the introduction, researchers and practitioners in swarm robotics uses a variety of tools to analyze the behavior of robot swarms. This is usually done in one of three ways: Through real robot implementations, computational simulation, or formal verification (Konur et al., 2010). These different tools for verification and validation each have different advantages and weaknesses. Generally, there exists a trade-off between realism and coverage, and between expressiveness and precision. (Webster et al., 2020). In this project, we will be using the modeling tool UPPAAL to model and analyze a swarm robotics algorithm using simulation and statistical model checking.

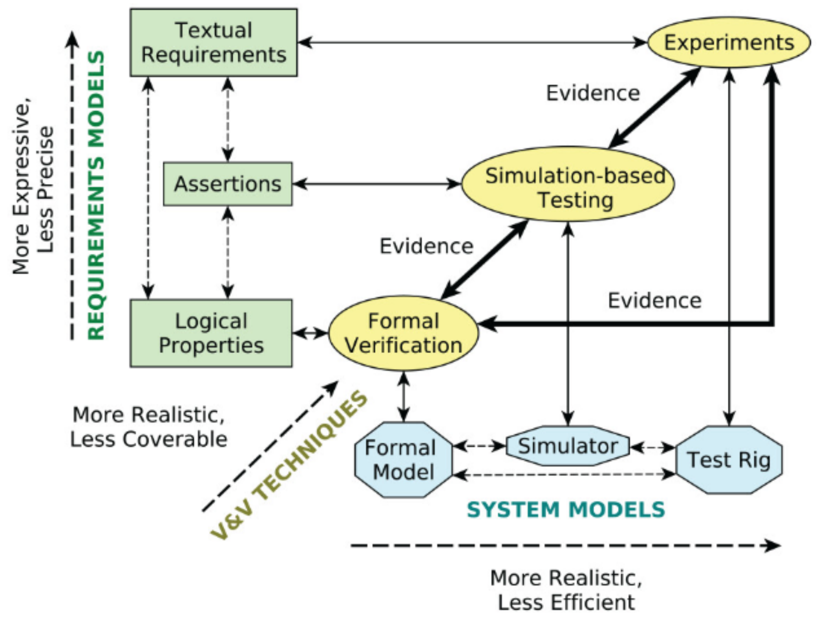


Figure 2: An overview of validation and verification (V&V) techniques. Figure taken from Webster et al., 2020.

Model checking is a formal verification technique for assessing whether a system satisfies properties obtained from its specification (Baier & Katoen, 2008, p. 3). Specifically, model checking can be defined as the automated technique of systematically checking whether a given formal property holds for a given finite-state model of a system (ibid., p. 11). As can be seen from this definition, model checking involves a number of activities.

First of all, in model checking a *model* is constructed from the system, that is, a formal unambiguous representation of the possible behavior of the given system (ibid., p. 12). Whereas the concrete formalism of the model depends on the modeling tool used[2], the model is usually comprised of a number of states/locations, denoting evaluations of variables etc., and possible transitions between those states, denoting changes in the system (ibid., p. 12). These models are derived from the specification of the system and can be more or less implementation-independent.

The *properties* to check the system against are also expressed formally in a property specification language, e.g. a linear-temporal logic, expressing propositions about possible behavior over time, such as "It is always the case that *p* eventually holds" (Baier & Katoen, 2008, p. 12).

Lastly these properties are *checked systematically* on the model. This is done algorithmically by the model checking tool, analyzing possible behaviors of the model (ibid., pp. 7, 13). In this process the whole state space of the system is usually systematically covered, in order to give mathematically rigorous guarantees (ibid., pp. 7-8).

There are a number of motivating factors for using model checking for verification. First, model checking requires stating the system and its specification in formal terms. This encourages that the system under test is understood and expressed unambiguously early on, which can often highlight

[2]See section 4 for the case of UPPAAL

ambiguities in the specification of the system (ibid., p. 7). This has proven useful in our project, as we have found many ambiguities in the specification of the modeled swarm algorithm, many of them early on. We will discuss these in section 4.

Another motivation for using model checking is that the system can be modeled and checked against properties at design stage and independent of implementation details. Ideally the model contains only details necessary for property-checking and abstracts away implementation- and hardware specific details. This makes analysis of the system easier, and the model can, afterwards, serve as a guiding blue-print for subsequent (possibly many) implementations of the system (Brambilla et al., 2012, pp. 145-146). This is beneficial in the case of our project, as it makes it possible for us to check properties for the swarm algorithm in a general way before deciding on a specific platform for implementing it.

However, model checking still does not eliminate the potential for errors: The verification is only as good as the model of the system, that is, if the model faithfully represents the system (Brambilla et al., 2012, pp. 145-146, Baier and Katoen, 2008, p. 8). As such, we are aware that our model itself can present a source of error, if not done carefully. To account for this, we have made a number of formal properties expressing expected behavior of the system, which the system can be checked against as a sanity check. These are described in section 4.4.

While we aspired to do *formal* model checking, covering the all possible states of the system, as a way to guarantee properties of the algorithm, we abandoned this pursuit because the state space of our model of each robot is very large, and because the interesting, emergent behaviors of our model are not prevalent at very small swarm sizes. Combined, it is infeasible with current model checkers to exhaustively search the state space (Baier & Katoen, 2008, p. 15). Instead we elected to pursue model simulations and statistical model checking. Next, we will elaborate on this.

3.3 UPPAAL

We are using the model-checking tool UPPAAL 5.0 for modeling and property-checking the swarm algorithm. Specifically, we are using features from UPPAAL's statistical model-checker, UPPAAL SMC [3]. "Classical" UPPAAL uses an exhaustive symbolic model checking and simulation method. In contrast, UPPAAL SMC does not verify properties by exhausting the entire state space of the model. Rather it runs concrete simulations of the system model, and uses various statistical results to determine whether a given property is satisfied with a certain degree of confidence (David et al., 2015, p. 398). This *slacking* of the coverage of model-checking allows utilizing more modeling-features such as floating-point values and arbitrary clock-rates, and it makes checking properties much faster. As such UPPAAL SMC trades off exhaustive analysis for the feasibility of simulating and checking systems with a much larger state-space.

Early in the modeling process, we made a rough, simplified version of our model that could be exhaustively checked using UPPAAL's symbolic model checker. Even at very small network sizes, querying the model was impractically slow. The specific issue in our case is that to model the algorithm faithfully, each robot need to represent a set of real values. To represent real values, we either need to use randomly generated floating point numbers, which are not supported by

[3]https://uppaal.org/features/

UPPAAL's model checker, or we need to simulate it using non-deterministically chosen integer values. The second approach is possible in UPPAAL's model checker, but it comes at the cost of drastically reducing the resolution of a core part of the algorithm. We found that at almost any resolution, the state space of the model explodes, as we essentially then were modeling floating point numbers using integers. While real-valued clocks are supported in UPPAAL's model checker (described below), we did not find a way to use clocks to simulate the real-valued part of the algorithm. As we will go into detail in section 4, the central state of the algorithm we model is a probability mass function, whose values do not change non-deterministically at defined rates relative to each other, like clocks in UPPAAL, but rather change discretely according to a complex but deterministic function.

We also considered using PRISM[4] for our model checking. PRISM is a tool for modeling systems showing probabilistic behavior. But while PRISM also supports statistical model-checking, we found that UPPAAL provided better and more intuitive support for certain modeling features (floating-point-operations, data-structures and functions). And in general, we deem the statistical checking features in UPPAAL SMC sufficient for our purpose.

3.3.1 Basic modeling formalism

In UPPAAL a model (also called *system model* in UPPAAL) comprises a network of timed automata, each functioning in parallel (for a formal definition see: Larsen et al., 2006). A timed automaton consists, similar to a finite-state-machine, of a number of locations[5], and a number of edges connecting locations. Exactly one of these locations is the *initial* location, meaning that the automaton will always start in that location.[6] Each edge goes from a source edge to a target edge, and can have a number of actions (concretely called *updates* in UPPAAL) associated to them, which are performed when enabling the edge, as well a number of guards - conditions required to enable the edge. These actions are performed sequentially and atomically..[7] Additionally, an automaton can own a number of local variables.

The automaton is *timed* in the sense that it can also contain a number of clocks, the values of which is constrained by invariants on locations. All clocks progress synchronously and evaluate to a real number at any given moment. However, individual clocks can be stopped or reset, and alternative clock rates can be specified. A location can be *urgent*, meaning that time will not pass while an automaton is in that location. Or it can be *committed*, which has the effect of being urgent with the addition that when an automaton is in such a location, the next transition of the system model must always involve an automaton in a committed location.[8]

A *state* in a timed automaton is defined by the location, evaluation of its variables and evaluation of its clocks. The state of the entire system model is defined by the states of all timed automatons with the addition of evaluations of *globally* defined variables and clocks. A transition, changing the state of the system model, can happen in two ways: 1) an automaton enables an edge at a given time, triggering any updates associated with the edge, or 2) an automaton performs a *delay-transition*,

[4]https://www.prismmodelchecker.org/

[5]We (as well as UPPAAL) use the term 'location' rather than 'state' as to not conflate it with the state of the automaton and the system

[6]https://docs.uppaal.org/language-reference/system-description/templates/locations/

[7]https://docs.uppaal.org/language-reference/system-description/templates/edges/

[8]https://docs.uppaal.org/language-reference/system-description/templates/locations/

staying at it's current location for a given duration $d \in \mathbb{R}_+$, constrained by given invariants. The change is chosen non-deterministically.

3.3.2 Making a model in UPPAAL

In practice, a model made in UPPAAL comprises a number of templates for instantiating timed automatons (also referred to as *processes* in UPPAAL), global variable declarations, local declarations for instances of each template, and a system declaration for instantiating a number of automatons. An example can be seen in fig 3, which we will now consider.

The global declaration of the system model is placed in the "Declarations"-file. Here, as well as in the files for local declarations, a number of variables can be declared. UPPAAL supports specifying these in a number of types such as int, boolean, and double and also supports arrays[9]. Each type has a default initialization value, respectively 0, false, 0.0, and an array of default values. UPPAAL also supports defining functions globally or locally, which can be used in updates and guards.[10] In this case (see figure 3d) the system model has just one such value, `fastLimit`, which is a constant (immutable value) of type `int` with value 12.

These inbuilt types are extended in UPPAAL with user-defined types and bounded integers. The value of an integer can be restricted to a range by declaring it as a *bounded* integer. For example, `int[1,12] x;` declares an integer variable restricted to the values $N \in [1, 12]$. These bounded values can be named and reused as user-defined *types*, e.g. the above bounded int may be declared as the type month as such: `typedef int[1,12] month;`. More complex user defined types are possible, but are not used in this project. A bounded integer or type can be used as an iterable ordered set of values in loops. A simple loop over all values in a type can be written as such: `for (m : month) updateMonth(m);` . Examples of this can be seen in our UPPAAL-model.

In this example, the system model has just one template, called `Example`. The template has a diagram-representation associated, as seen in fig 3a. Also, indicated in the top, a number of parameters can be specified for the template, specifying arguments that must be given an instantiation of the template (Larsen et al., 2006, p. 4). The `Example`-template has just one parameter, `id` of type `int`.

The `Example`-template also has a number of local variables and functions associated to it, as seen in 3b. In this case, a clock, `localTime`, two boolean values, `guardValue` and `wasFast`, and a size-2-array of type double, `timeUsed`. Additionally `Example` has a function, `checkSpeed()`, associated with it. As these functions are declared locally in a template, they are similar to methods in object-oriented programming, and can read and modify the local state of the process calling it.

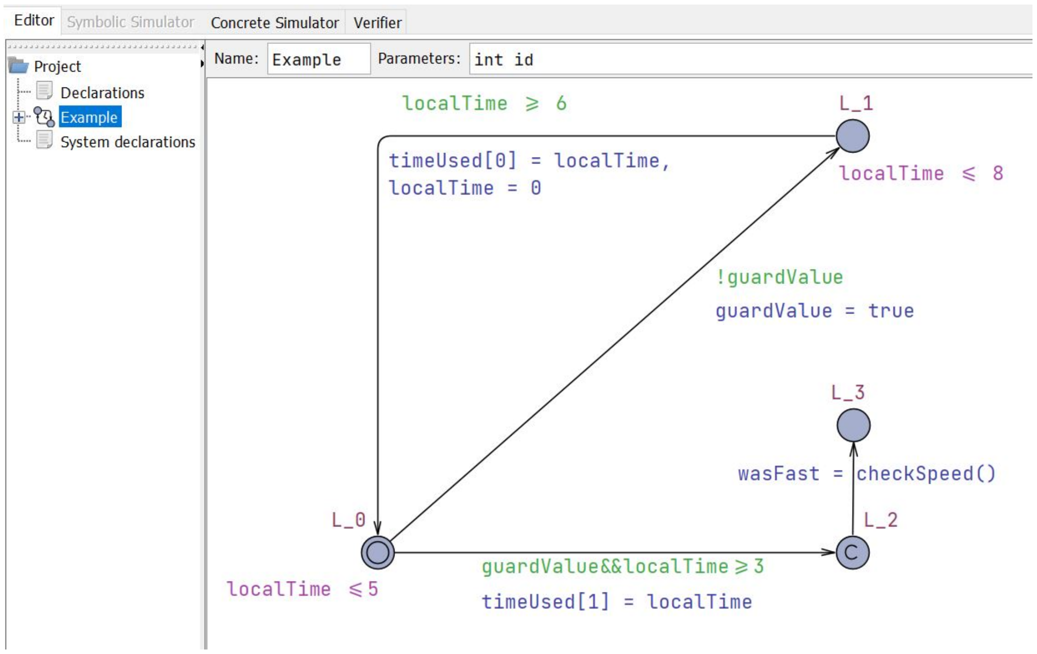
Looking at the diagram in 3, the `Example`-template has four locations, denoted by the circles `L_0`, `L_1`, `L_2` and `[L_3]`. The circle inside `L_0` indicates that it is the initial location of the automaton and the C inside `L_2` indicates that it is a committed state. At `L_0` and `L_1` invariants are associated, written in purple, indicating bounds for which interval of values of `localTime`

[9]Though doubles are simply ignored for symbolic verification: https://docs.uppaal.org/language-reference/system-description/declarations/types/

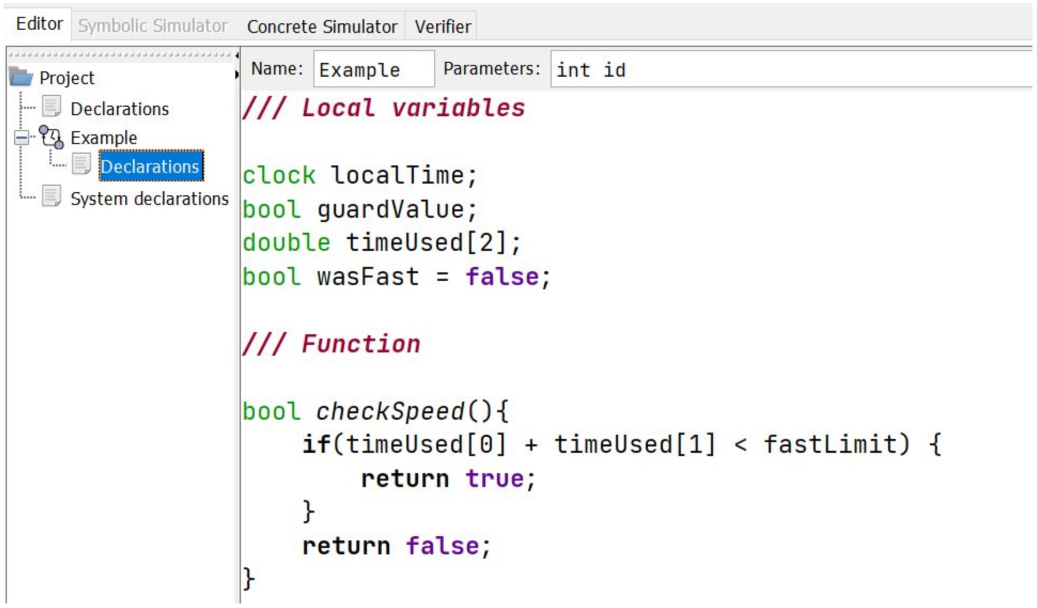
[10]https://docs.uppaal.org/language-reference/system-description/declarations/functions/

an automaton can be in said state. For example, the automaton can only be in state `L_0` when `localTime` $\in [0, 5]$.

There are also a number of edges connecting the locations, indicated by arrows. Each of these edges has a guard associated to it, written in green, and one or more updates, written in blue and separated by ",". For example, the edge from `L_0` to `L_2` can only be enabled if `guardValue` evaluates to true and `localTime` exceeds 3. When the edge is traversed, `timeUsed[1]` is set to



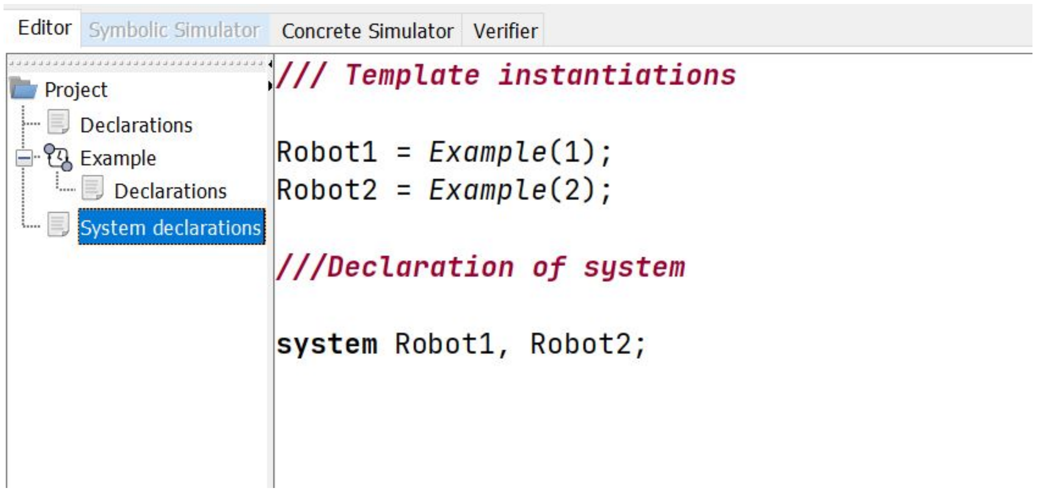
(a) Template for automaton



```
/// Local variables

clock localTime;
bool guardValue;
double timeUsed[2];
bool wasFast = false;

/// Function

bool checkSpeed(){
    if(timeUsed[0] + timeUsed[1] < fastLimit) {
        return true;
    }
    return false;
}
```

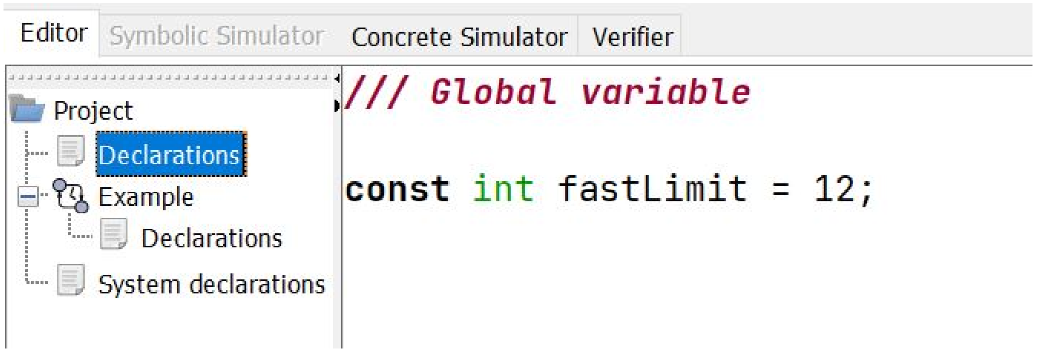
(b) Local declarations

Figure 3: Example system model in UPPAAL



```
/// Template instantiations

Robot1 = Example(1);
Robot2 = Example(2);

///Declaration of system

system Robot1, Robot2;
```

(c) System declaration



```
/// Global variable

const int fastLimit = 12;
```

(d) Global declarations

Figure 3: Example system model in UPPAAL (cont.)

the value of `localTime`.

Putting all of the above together, an instance of the `Example`-template, goes from `L_0` to `L_1`, back to `L_0` and finally to `L_2` with delays within the time-bounds, recording the value of `localTime` at the second and third enabled edge. When traversing the edge to `L_3` (which is enabled immediately when the automaton reaches `L_2`), total time used is evaluated using the `checkSpeed()` method. `checkSpeed()` compares values in the process-local `timeUsed` array to the global constant `fastLimit`. This order of enabling edges is determined by the guards of each edge.

As seen in fig 3c two instances of the `Example`-template is made with parameters 1 and 2 associated to variables `Robot1` and `Robot2`, after which the system model is declared with the `system`-keyword, consisting of the two instances.

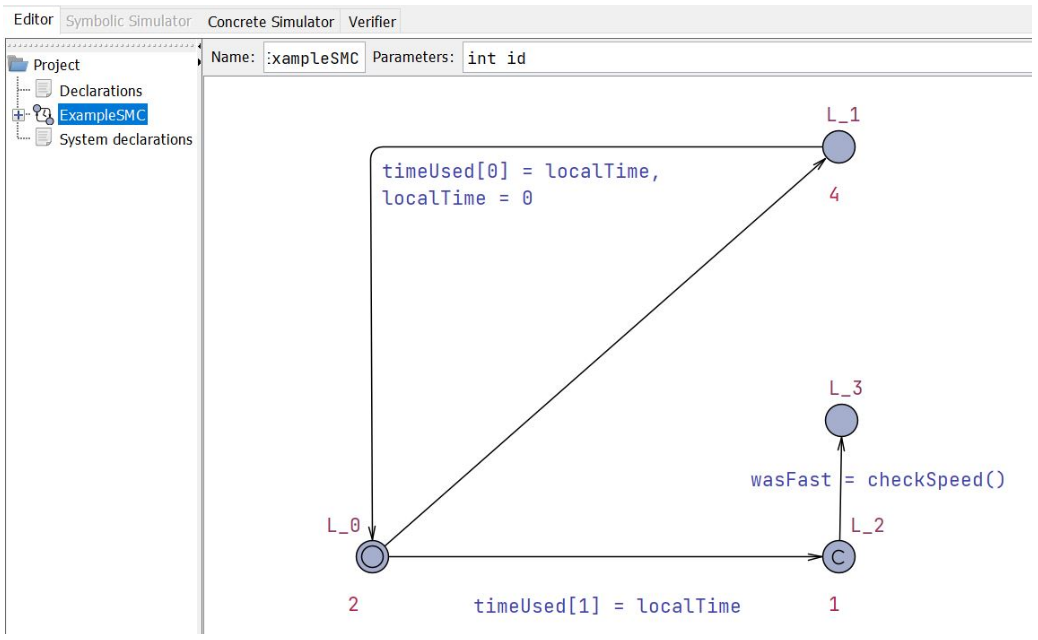
3.3.3 Statistical model checking in UPPAAL

In UPPAAL SMC these automata are extended with a stochastic component, replacing non-determinable transitions with probabilistic ones. If a location has a time-invariant, an edge transition is chosen by uniform distribution within that time-constraint. Otherwise one can specify a delay by choosing a rate, λ for an exponential distribution for the delay, d, with PDF $F(d) = \lambda e^{-\lambda d}$

(David et al., 2015, p. 398).[11]

When more (possible) edges are present in a given location, a given edge will be enabled according to uniform distribution. Or one can assign each of n edges from a location a given probability, $p_i \in [0, 1]$, where $\sum_{i=1}^{n} p_i = 1$.

Applied to our previous example, a modified version of the `Example`-template with this extension could look like `ExampleSMC` in figure 4.



```
:xampleSMC
int id
L_1
timeUsed[0] = localTime,
localTime = 0
4
L_3
wasFast = checkSpeed()
L_0
L_2
C
2
timeUsed[1] = localTime
1
```

Figure 4: Example template with stochastic extension

In `ExampleSMC` there are no guards on the edge from `L_0` to `L_1` nor the edge from `L_0` to `L_2`, so one of the edges will be randomly enabled with probability 0.5. Furthermore, instead of invariants, `ExampleSMC` has exponential rates for the PDF of delays in each location, indicated by the brownish integers at each location.

Given this extension, it is possible to check probabilistic properties against a model. In UPPAAL SMC this is done by running probabilistic queries, expressed in MITL, on a model (David et al., 2015, p. 401). These queries take on one of the following forms:

- Simulate $[bound; N]\{expression_1, \cdots, expression_n\}$: $SatRuns$: $SatExpression$
- $\mathrm{E}[bound; N]($ min | max : $expression$)
- $\Pr[bound](\phi)$
- $\Pr[bound](\phi) \geq probability$
- $\Pr[bound_1](\phi_1) \geq \Pr[bound_2](\phi_2)$

Here ϕ is an MITL-formula of the form [ ] | <> *expression*, where [ ] denotes ”always holds” and <> denotes ”eventually holds”.*expression* denotes some state- or variable-based expression. *bound*

[11]See also: https://docs.uppaal.org/language-reference/system-description/semantics/

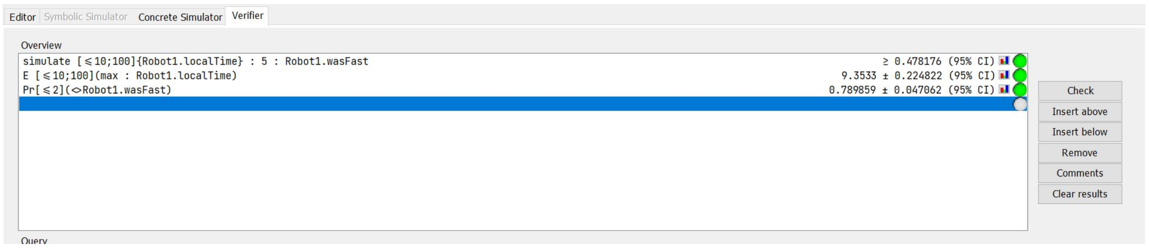
can be expressed in clock-values or number of edge-transitions.

To answer these queries, UPPAAL runs simulation of random runs of paths bounded by the length of the bounds. In the first two types of queries above, N number of those runs are simulated, in which the values of the given expressions are monitored (their evaluation over time in the case of "Simulate"-queries and their expected minimum/maximum value in the case of E-queries). Optionally the monitored runs in "Simulate"-queries can be filtered by an expression, *SatExpression* and a number of runs satisfying this expression, *SatRuns*.

As such, if we want to track the value of `localTime` over time in `Robot1` over the course of 100 runs of length less than 10 time-units, finding the first five runs where `wasFast` evaluates to true, we could run the query `simulate [<=10;100]Robot1.localTime : 5 : Robot1.wasFast` on the system model. If we instead wanted the average maximum-value of `localTime` for each of those runs, we could have ran the query `E [<=10;100](max : Robot1.-localTime)` on the system model.

In the last three cases a variable number of random, bounded runs are simulated, here modeled as Bernoulli random trials, where satisfying the given ϕ is equivalent to a success. A number of runs are simulated until, applying smc-algorithms (see: David et al., 2015), the query can be answered with the desired confidence, specified by a given number of statistical parameters. For example, for checking the query Pr[bound](ϕ), runs are repeatedly ran until the computed confidence interval has the a desired half-width ϵ and test-coverage α.[12] This mean that the running time of the statistical model checking depends on the desired level of confidence, and is otherwise linear on the bound-length of runs.

For example, if we want to check in the above system model, with confidence $\alpha = 0.05$ and half-width $\epsilon = 0.05$, the probability that `wasFast` eventually evaluates to true for `Robot1` within 2 time units, we can run the query `Pr[<=2](<>Robot1.wasFast)` on the system model. The value of the statistical operators α and ϵ, can be set under "Options → Statistical parameters" in the navigation bar.



```
simulate [≤10;100]{Robot1.localTime} : 5 : Robot1.wasFast          ≥ 0.478176 (95% CI)
E [≤10;100](max : Robot1.localTime)                           9.3533 ± 0.224822 (95% CI)
Pr[≤2](<>Robot1.wasFast)                                   0.789859 ± 0.047062 (95% CI)
```

Figure 5: Example of probabilistic queries in Uppaal

In summary, rather than exploring the state-space of the model, UPPAAL SMC confines itself to running a number of bounded simulations, which usually result in much faster running times for system models with huge state spaces.[13]

[12] https://docs.uppaal.org/language-reference/query-semantics/smc_queries/ci_estimation/
[13] https://docs.uppaal.org/language-reference/query-semantics/smc_queries/ci_estimation/

4 UPPAAL Model Implementation

4.1 Priorities and design decisions

For our project, considering the research questions, the aspects of the algorithm that we are interested in modeling is the global state of the network, specifically the notion of iterations as a measure of running time. That is, we are interested in the number of updates needed before the swarm reaches a consensus.

We prioritized respecting the constraints of the algorithm, specifically that robots should only have local information. We examined whether the process of transmitting local knowledge across the graph would introduce subtle changes to the theoretical behavior of the algorithm. As we did not have a concrete implementation of the algorithm, we were interested in how much of a challenge it was to respect these constraints.

As a result the model represents a quite faithful simulation of the robots interactions as described in the paper. However, this required us to make a few implementation decisions of note that were left under-specified by the original authors.

This design decision introduced additional steps in the modeling process, as it requires modeling a communication algorithm, which in turn required additional debugging, implementation and verification steps.

As we are interested in the state of the network as a whole, we abstract away the concrete communication between robots. Instead, we simulate communication limits using a mask for each robot over global arrays indexed by the id of each robot. This mask is called `C`, and determines the network by representing the local connection group for each robot. Following is an example of how a network of three robots in a line would look like:

```
bool C[3][3] = {{false, true, false},
                {true, false, true},
                {false, true, false}}
```

Where a true value means that the robot at that index is a direct connection or neighbor. This means that all information that a robot may request of a neighbor is available to be read without modeling concrete communication between those robots.

4.2 Overview of model

In our UPPAAL-model, each robot is a process with an ID and five possible locations, including an initial and a terminal location.

The network can have either a completely random distribution of initial preferences, or it can be random with a number of seeded robots. The number of seeded robots are controlled with the constant `SEEDNUM`. The seed values are manually defined along with the network. For cases of 30 choices we set the distribution at 13% preference towards choice 0, and a 3% preference towards all other options. As Lee and Liu do not specify the exact preference distribution given to seed robots, we have derived this distribution experimentally. The values we use have produced results that are very comparable to those of the original article (Liu & Lee, 2019). See 5 for further exploration.

At location `PreferencesUpdated`, the robot has updated their preferences and their exhibited decision. At location `FindingDecisionGroup`, the robot is in the process of determining its consensus group D. At location `AwaitingNeighbors` the robot either terminates (if a global consensus has been found) or it moves on to a new iteration of updating its preferences (by traversing the edge to `PreferencesUpdated`).

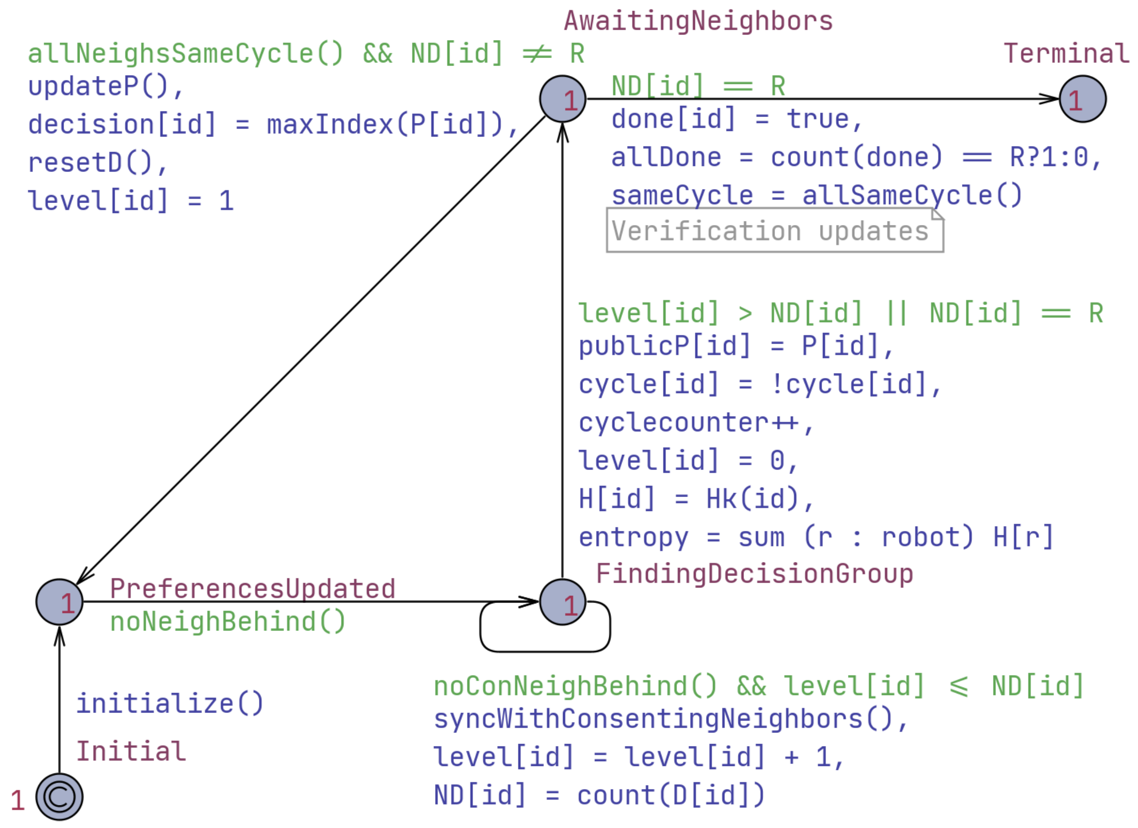


Figure 6: The Robot template in UPPAAL. Besides variable and function declarations, the system consists of m instantiations of this state machine with unique id values. Each state is named (in brown, capitalized), and has a rate of exponential of 1. Each edge transition has a guard statement (boolean expression) and a comma-separated sequence of assignments and method calls. Guard statements are in green and appear above assignments. They are placed as a block close to the edge they guard.

The edge from `Initial` to `PreferencesUpdated` directly corresponds to the initialization of the algorithm (Liu and Lee, 2019, section 2.1). The edge from `AwaitingNeighbors` to `PreferencesUpdated` corresponds to both (ibid., section 2.2) preference updating via local interaction and (ibid., section 2.3) internal processing for decision uncertainty reduction.

All the edges leading to or originating in `FindingDecisionGroup` are needed to update the robots' local understanding of their consensus group, which is not specified as a separate step in the algorithm, but which requires a separate step in the model to preserve the constraint of local

Code	Algorithm	Meaning
`D[id]`	D_k	Consensus group for robot `id` / k
neighbors, or `C[id]`	C_k	The connection group for robot `id` / k
`ND[id]`	N_k	Size of consensus group for robot `id` / k
`P[id][j]`	$P_k(j)$	Robot `id` / k's preference toward decision j
`decision[id]`	exhibited decision	The highest preference value of robot `id`
`R`	m	Number of robots in system
`Q`	n	Number of decisions
cycle	iteration	A full update of preferences
`EMPIRICAL`	λ_T	The empirical convergence threshold value
`s[j]`	$s(j)$	The rank of choice j for a given robot.
`Hk()`	H_k	(section 3.1) Entropy value for robot k
`entropy`	$\sum H_k$	Total entropy of system
`updateP()`	P_k	Eq. (1) Include uncertainty reduction if relevant.
`divergence()`	λ_k	Eq. (2) The degree of divergence for robot k
`accConvergence()`	P_k^{new}	Eq. (3) The method normalizes the new values.
`L(s[j], Ll, Lu)`	$\mathscr{L}(s(j))$	Eq. (4) The linear multiplier for choice j.

Table 1: Explanation and translation of coding variables to algorithm terminology, with specific references to sections and equations in Liu and Lee, 2019

communication. We describe the algorithm for calculating consensus groups below.

4.2.1 Timing and synchronization

Because the paper did not specify any absolute or relative timing requirements, we decided not to give time invariants to locations, and rather uniformly give each location an exponential rate of 1. In our system, then, time is a proxy for the average number of edge traversals of the robots.

In terms of synchronization, the robot uses two local variables: `cycle[id]` of type boolean and `level[id]` of type integer, bounded by the number of robots `R`.

`cycle[id]` represents the current cycle of the robot, relative to its neighbors. The cycle is flipped when traversing the edge between `FindingDecisionGroup` and `AwaitingNeighbors`. To continue from that location, all of a robots neighbors must be on the same cycle. Deadlocks are avoided because no robot will traverse a full cycle before all of its neighbors has exited `AwaitingNeighbors`, as they also have to synchronize at location `PreferencesUpdated`.

`level[id]` represents the progression of the robot within its current cycle, and is reset to 0 with each new cycle. Level 1 and above signifies that a robot has updated its preferences and exhibited decision. By waiting for all its neighbors to reach level 1 a robot ensures that it knows which neighbors exhibits the same decision (consents with) it, which is a precondition to determine its consensus group.

When determining the consensus group, a robot continuously updates its consensus group to be the union of all its neighbors. This process is synchronized using levels, such that all information is guaranteed to travel throughout the consensus group within `R` levels. We give a more thorough walkthrough of this algorithm below.

In total, local synchronization is quite strict. A robot waits for all its neighbors to reach a certain state twice within each cycle, and potentially waits for all consenting neighbors at each level between

1 and `R`. However, at a global level, this model makes no assumptions about neither speed nor interleaving of updates. There is not a global clock used for synchronization, and in a large network, at a given time before consensus is reached, robots may have completed quite different numbers of cycles, while in the end they will all have completed an equal number of cycles. This is interesting for two reasons: The loose global synchronization shows that the properties we test hold for networks without global coordination, and the strict local synchronization is part of the network's strategy to increase fairness.

4.2.2 Fairness and atomic updates

When considering the implicit assumption in the original article that robots are globally synchronized in iterations, we find that the specific synchronization policy influences the behavior of the algorithm. A robot that reevaluates their preferences at a higher rate than its neighbors becomes more influenced by its neighbors than the other way around. Also, robots out of sync introduce possibilities of data consistency errors as robots request and update their preferences, and determine their consensus groups concurrently. One way of solving this would be to have global synchronization with a predetermined order of updates for all robots, however as noted above, we were interested in respecting the requirement that robots do not communicate globally.

By using the above synchronization methods, we ensure that all robots complete the same number of preference updates, and that at any given time, a robot can at most be one update ahead or behind its neighbor. With no further modification, we would be left with the problem that the result of updating a robot's preferences is influenced by the sequencing of when the other robots in its connection group update their preferences.

We solve this by implementing a public / private distinction for the preferences. A robot updates its `P[id]` array by reading from each robot r of its neighbors' `publicP[r]` array. The `publicP` arrays are only updated when entering the `AwaitingNeighbors` location. As noted in the subsection above, no robot can enter `AwaitingNeighbors` before all of its neighbors have traversed the edge to `PreferencesUpdated` edge and thereby updated their preferences. In this way the updates to a robot and its neighbors preferences occur as if instantaneously from the perspective of a given robot.

4.3 Algorithm for locally determining consensus group

As it is not specified in the original article, how robots in the network updates each other on changes to their consensus groups, we've had to decide on a proper way to model this ourselves. The authors of the article specifically state that interesting new avenues of research for the algorithm includes improvements to the discovery and updating of local consensus groups, including possible changes to the algorithm. As such, we saw it as a relevant contribution to model this aspect of the algorithm more thoroughly.

To address the ambiguity in the specification, we considered two different solutions: one that makes use of a custom implementation of the depth-first search algorithm[14] to scan a global structure of the robot network, thus finding the size of their consensus-group during every update. This approach

[14](https://en.wikipedia.org/wiki/Depth-first_search)

adheres to the assumption of perfect local knowledge of the agents, but fails to adhere to the constraint of local communication. The other solution is more complex but adheres better to the constraint that individual robots are only able to communicate with their neighbors.

While we ended up using the latter solution, we still ended up using the dfs-solution for making a variant of our model, `combined_dfs_sync.xml` , for comparison-purposes (see below). The rest of this subsection discusses the custom algorithm we developed.

Since any change in exhibited decision in any robot can result in a drastic change in the consensus group of another robot, we consider the most transparent and faithful model to be one in which every robot recalculates its consensus group at each cycle.

The algorithm has three requirements: The consensus groups should be correct, it should not change the behavior of the algorithm and it should only use local information. Further, we want the process to be efficient enough to be practical, and it should be as simple as possible so it is easy to analyze and reason about.

In our approach, each robot starts out by including only themselves in their decision group. Then all robots continuously pull decision group information from their connected consenting neighbors, and eventually, all robots will be aware of all members of their consensus group.

To synchronize this behavior, a robot is only allowed to pull information if no consenting neighbor is at a lower level than it. When it has pulled information, it increments its level and recalculates its local understanding of its consensus group size. This ensures that all information will travel across the consensus group. In the worst case, the information from one robot travels in a path including all other robots in the consensus group before reaching a final robot, and so the total number of levels gained required to be certain that all information has been gained is equal to the number of robots in the correct consensus group. Because of this insight, we can reason that if a robot has a local understanding of its consensus group size d, and it has gained d levels, then it is certain that it has found the correct consensus group.

A more inductive reasoning of the above arguments goes as such:

In the base case, where the correct size of the consensus group $|D| = 1$, a robot initially believes $|localD| = 1$, which is correct. However, to be certain it is correct, the robot tries to pull information from its neighbors, does not find any new information, gains a level, and concludes that it is finished, since it has gained 1 level and has $|localD| = 1$. If $|D| = 2$, the robot must have acquired that information when asking its consenting neighbor. Similarly, the consenting neighbor must have come to this conclusion when asking the aforementioned robot.

If $|D| = 3$, at least one robot must discover both of its connected consenters in the first level, in which case it repeats for a few levels without learning anything new. If any robots do not gain this complete insight at the first level, they still find one other robot and so have a $|localD| = 2$. In that case, they have only gained one level, and so they repeat the process once more. In the next level, they are guaranteed to find the final robot, since they are necessarily connected to a robot that has discovered it.

Described in another way, if a robot has $|localD| = d$ and is level $d - 1$, it must pull information again. Then, it either does not find anything new, in which case is confident that $|D| = d$, or it does, in which case $|localD| = d$, $d > d$ and the robot must continue pulling until it is level $d - 1$, at which point the scenario repeats.

4.4 Verification of model

While we are using the model to verify properties of the algorithm, we have also separately worked on verifying the quality of our model itself.

We made another model using depth-first search to determine the decision group sizes and simpler synchronization policies. The idea is to compare the behavior of those models. This variant of our model, `combined_dfs_sync.xml`, uses the global `C`-array to strictly synchronize robots at each location of the model and furthermore uses dfs on the `C`-array to find the size of consensus-groups. While not being as faithful to the assumptions of the original algorithm, comparing our model to this one has helped us get an idea whether our model works as intended, and whether the modeling choices we have made has an influence of the behavior of the algorithm.

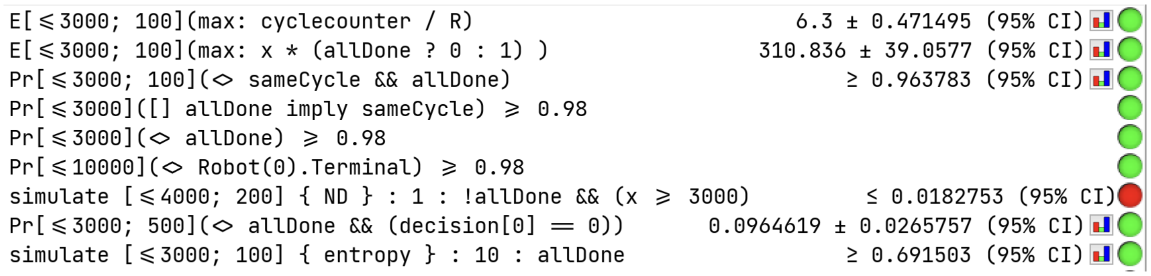
To support this evaluation, we have designed several test topologies of small networks for assessing the correctness of our model-implementation. These include a minimally connected (a line) and a maximally connected network, a circular network and a randomly generated network.

The basic properties we check to see if the networks behave correctly are:

1. The expected number of cycles and expected time before termination are measured as a sanity check to see if performance changes unexpectedly after changes to the network or to the model (Property 1 & 2 in fig. 7).
2. All robots terminate in the same cycle and within a bounded time frame (Property 3, 4, 5, 6).
3. The model-checker fails to find an example of a network that after a bounded time frame is not completed (Property 7).
4. The chance of the consensus being decision 0 is measured, to make sure it corresponds to random chance for unseeded networks, and is high for seeded networks. (Property 8).
5. The entropy is measured to show that entropy falls with a sufficiently high `empirical` value, and does not if `empirical` is 0 (Property 9).

See Fig. 7 for the concrete syntax.

We have also continuously inspected concrete simulation traces to compare actual behavior with expected behavior. Finally, we have used a few evaluation variables to construct basic verification queries to check in the statistical model checker.



```
E[≤3000; 100](max: cyclecounter / R)                              6.3 ± 0.471495 (95% CI)
E[≤3000; 100](max: x * (allDone ? 0 : 1) )                   310.836 ± 39.0577 (95% CI)
Pr[≤3000; 100](<> sameCycle && allDone)                             ≥ 0.963783 (95% CI)
Pr[≤3000]([] allDone imply sameCycle) ≥ 0.98
Pr[≤3000](<> allDone) ≥ 0.98
Pr[≤10000](<> Robot(0).Terminal) ≥ 0.98
simulate [≤4000; 200] { ND } : 1 : !allDone && (x ≥ 3000)         ≤ 0.0182753 (95% CI)
Pr[≤3000; 500](<> allDone && (decision[0] == 0))      0.0964619 ± 0.0265757 (95% CI)
simulate [≤3000; 100] { entropy } : 10 : allDone                    ≥ 0.691503 (95% CI)
```

Figure 7: An example of properties checked to verify model. These results are for a network of 30 robots and 10 decisions with no seeding and an empirical threshold value of 0.

4.4.1 Evaluation variables

We have introduced five global variables to monitor properties of the algorithm. `done[id]` reflects if a robot is in the terminal state. `allDone` reflects if all robots are in the terminal state. `sameCycle` reflects if all robots are in the same cycle after termination. `cyclecounter` counts the total times any robot has completed a cycle. If all robots end in the same cycle, then `cyclecounter / R` gives the number of (global) cycles before a consensus was reached. `H` and `entropy` are measures of the global entropy in the system, derived using the formula in Liu & Lee (section 3.2).

4.4.2 Uncertain quality of convergence acceleration algorithm

We have implemented the decision uncertainty reduction part of the original algorithm. It relies on an "empirical threshhold value" (ibid., section 2.3) λ_T. We took that implicitly to mean that the value should be determined by experiment based on how much of a performance improvement it produces in terms of cycles before a consensus is reached. However, we have not found a clear example of any value resulting in faster consensus times. We have tested with many different values in the range of 0.001 to 100.0, and conversely tend to see increased average cycles to reach consensus. We have even found very rare edge cases where circular topologies can end in a stalemate with two equally large and dominating consensus groups for large values of λ_T (> 1). It is unclear what leads to these unexpected results, but given the absence of detail in the original article, we have been unable to verify the best or most compatible empirical values to properly optimize the decision making process. It is not clear how the resulting values should be normalized (which they have to by definition of $P_k()$ as a probability mass function), nor is it clear if the convergence acceleration should be added to the preference update process or replace it.

On the other hand, the behavior makes sense intuitively. If a network at some point starts containing multiple locally converged competing clusters (which happens easily in linear or circular topologies), they will become increasingly confident in their decision and so it is to be expected that more cycles will be required for one cluster to convince another.

It is possible that this is intended behavior. Convergence acceleration may improve some other metric than number of cycles till consensus. E.g. the authors point out convergence acceleration reduces discrete entropy. We might also consider that the process qualitatively improves results in a way not easily measurable. We further discuss this question of result quality in section 6.2.

5 Results

In the following, we present the results of testing various properties of our model in UPPAAL. Our findings are then compared to the results from the algorithm paper's experiments, which were originally conducted in a python script using Pygame (Liu & Lee, 2019).

5.1 Graph Generation

For the purposes of testing and verifying their algorithm, Lee and Liu use a number of randomly generated networks. These networks have certain characteristics and properties that are relevant for the metrics of the algorithm. First, the networks are generated with a specific amount of robots m and opinions n. Second, all robots in the network have a maximum of 6 neighbours. This forms the equilateral triangle grids that is shown in the figures of the algorithm paper. Lastly, these graphs are each associated with a specific value of the previously mentioned *network dependency*, a descriptor of the importance of the most critical nodes in maintaining connectivity in the network (Liu & Lee, 2019). To stay as truthful to the original article as possible, we created a graph tool in python using NetworkX and Matplotlib (Hagberg et al., 2008; Hunter, 2007). The code for this tool can be found in the link in the references (Klarskov et al., 2025) or in the code repository. We leverage this tool to create semi-random graphs from set values of n and m. The tool can subsequently calculate the network dependency of the graphs and draw visual representations of the generated networks. The tool also creates a direct string-representation of the network that can be copied into UPPAAL to recreate the same exact network.

5.2 Property testing

5.2.1 Sanity checks and alignment

Besides using properties to verify the model as described in 4.4, we ran an experiment to compare the results of running our model with the custom consensus group identification algorithm with the results of the depth-first search variant. We did this by comparing our model with the dfs-variant on the suite of networks of different sizes (see below), tracking the average number of cycles used for each size.

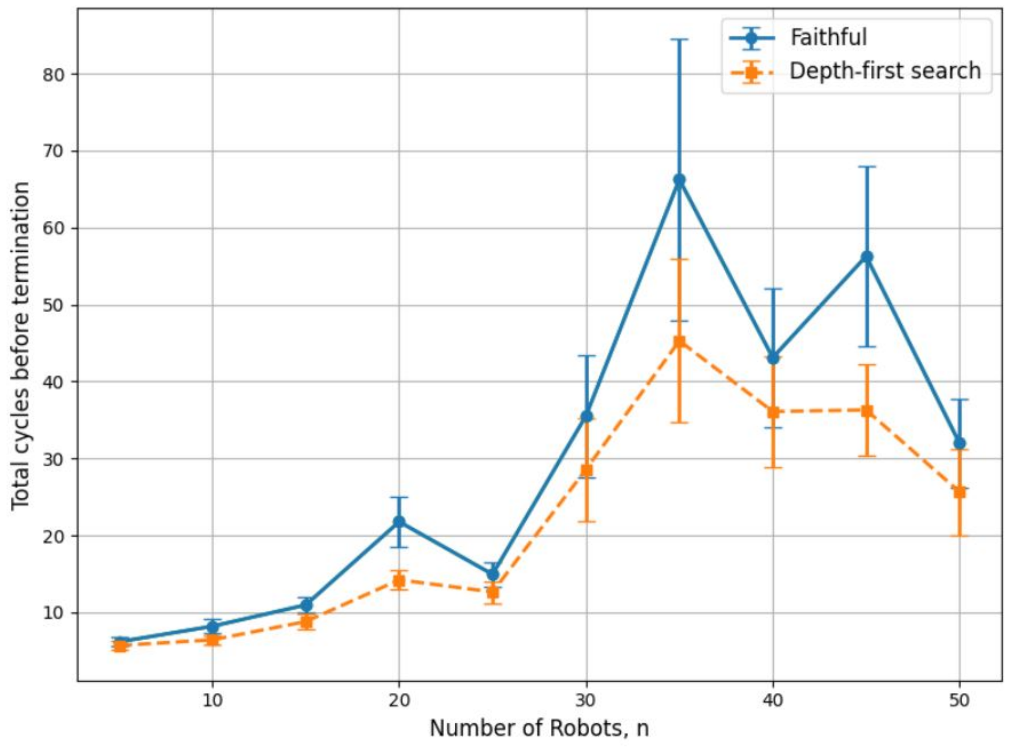


Figure 8: Average number of cycles used for consensus on networks of different sizes shown for our model (”Faithful”) and the dfs-variant of our model (”Depth-first search”)

As can be seen from 8, the results closely align, which indicates that our model actually represents the expected behavior of the system in the way we intended it to.

Lee and Liu experiment with the effects of a number of various parameters, namely the network topology and size, and the number of decision from which the robots can choose. In the following sections, we run similar tests to compare our model against the more idealized results from the original article (Liu & Lee, 2019). We defer this comparison to section 8.

5.2.2 Effects of network topology

Topology Experiment	Network Size	Number of Options	D_{rel}
Original (Liu and Lee, 2019)	30 robots	30 options	0.22 - 0.43
Our Experiment	30 robots	30 options	0.22 - 0.43

Table 2: Comparison of experimental designs for testing the effects of network topology

For the effects of network topology, we copied all of the experiment parameters. However, since the networks are randomly generated, they are never going to be exact duplicates of the original networks. Like Lee and Liu, we run each experiment for a 100 trials and post the average in the graph below. Our exact queries can be found in appendix A.

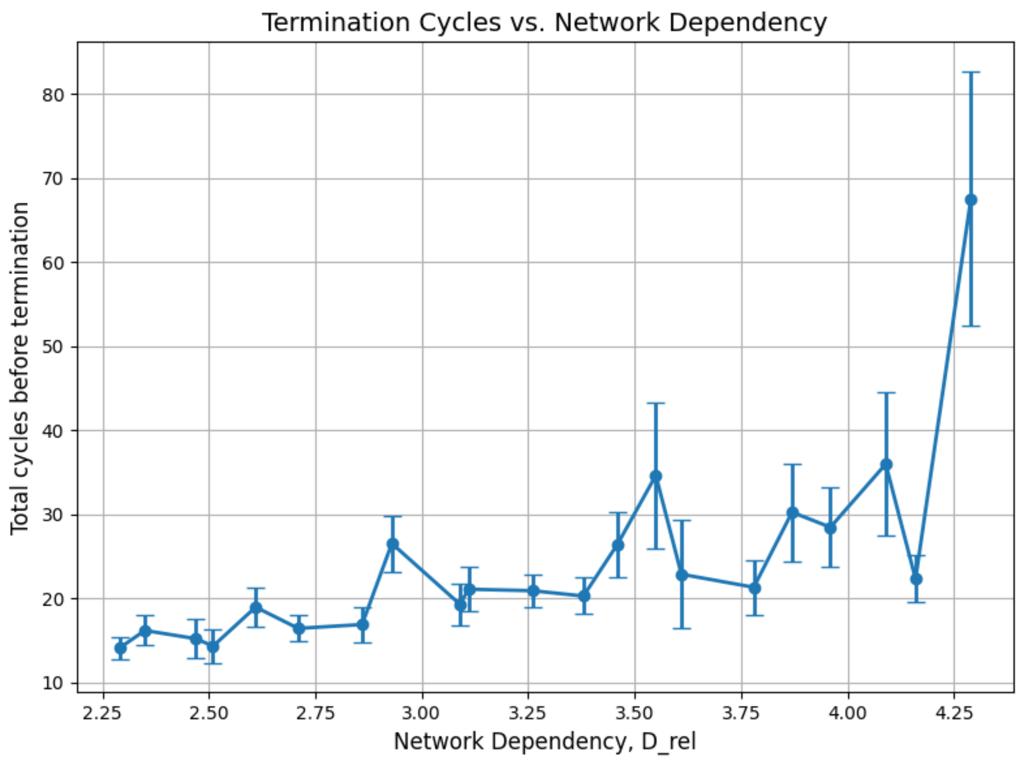


Figure 9: Results from experiments based on network dependency. The correlation between these variables have been found to have a Pearson value of 0.69

These variables are found to be moderately correlated with a Pearson value of 0.69 (Schober et al., 2018, p. 1765). The results are statistically significant with $P < .00001$.

5.2.3 Effects of network size

Network Size Experiment	**Network Size**	**Number of Options**	$\boldsymbol{D}_{\text{rel}}$
Original (Liu and Lee, 2019)	30 - 150 nodes	30 options	3.0 ± 0.1
Our Experiment	5 - 50 nodes	30 options	3.0 ± 0.1

Table 3: Comparison of experimental designs for the effects of network size

For the effects of network size, we copied all of the experiment parameters as well as possible, but were forced to user smaller network sizes. This is simply due to how UPPAAL conducts the experiments, which becomes computationally heavy and unfeasible before the program crashes. The results, however, are still of interest. Like before, we run each experiment for a 100 trials and post the average in the graph below. Our exact queries can be found in appendix A.

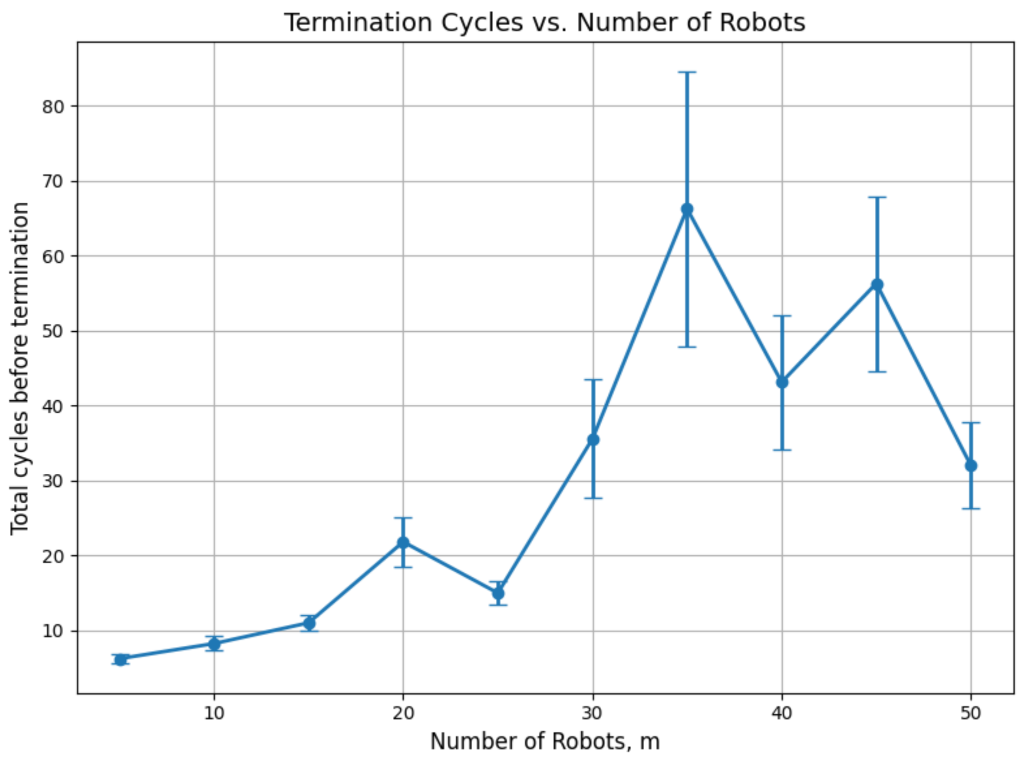


Figure 10: Results from experiments based on network size. The correlation between these variables have been found to have a Pearson value of 0.78

These variables are found to be strongly correlated with a Pearson value of 0.78 (Schober et al., 2018, page 1765). The results are statistically significant with $P < .00001$.

5.2.4 Effects of the size of the decision space

Options Experiment	**Network Size**	**Number of Options**	D_{rel}
Original (Liu and Lee, 2019)	30 nodes	10 - 300 options	2.672
Our Experiment	30 nodes	10 - 300 options	2.675

Table 4: Comparison of experimental designs for the effects of option space size

For the effects of the size of the decision space, we copied all of the experiment parameters as well as possible, but due to the randomly generated networks, we had to choose a network with a 0.003 higher network dependency score. We expect that this small parameter change will yield no significant change in the results. Once again, we run each experiment for a 100 trials and post the average in the graph below. Our exact queries can be found in appendix A.

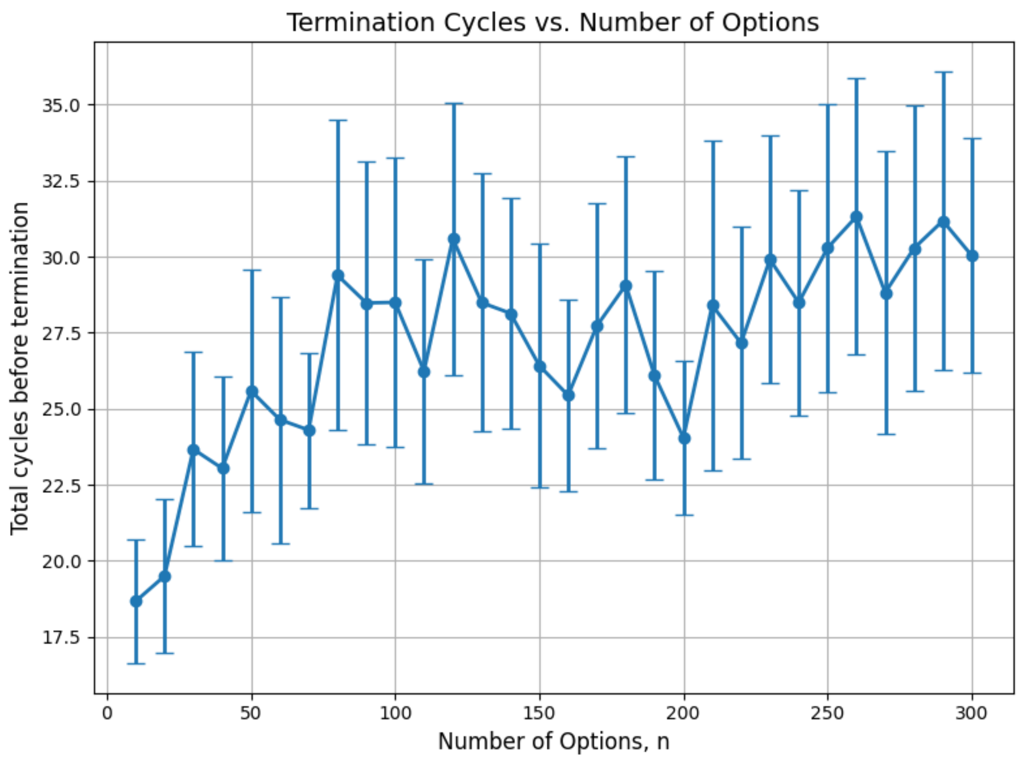


Figure 11: Results from experiments based on number of options. The correlation between these variables have been found to have a Pearson value of 0.73

These variables are found to be strongly correlated with a Pearson value of 0.73 (Schober et al., 2018, page 1765). The results are statistically significant with $P < .00001$.

5.2.5 Effects of the size of external interference

Lee and Liu additionally run a number of experiments on the effects of seeding (Liu & Lee, 2019). Seeding is meant as the act of predetermining the preferred decision of a number of robots. The effects of these seeds robots on the final outcome are then observed. The details for the original experiment (Liu & Lee, 2019) are somewhat unclear. For example, there is no mention of the network dependency value of the networks or the size of the decision space. For our experiments, we will be using a randomly generated network of 30 robots with a network dependency value of 2.84. We set the number of options to 30, and predetermine the seed robots to exhibit a preference of 13% for the first option at index 0, with the remaining options each having a 3% preference. This spread can be largely influential, and no precedence is set in the original paper. Consequentially, the results should be interpreted with this in mind.

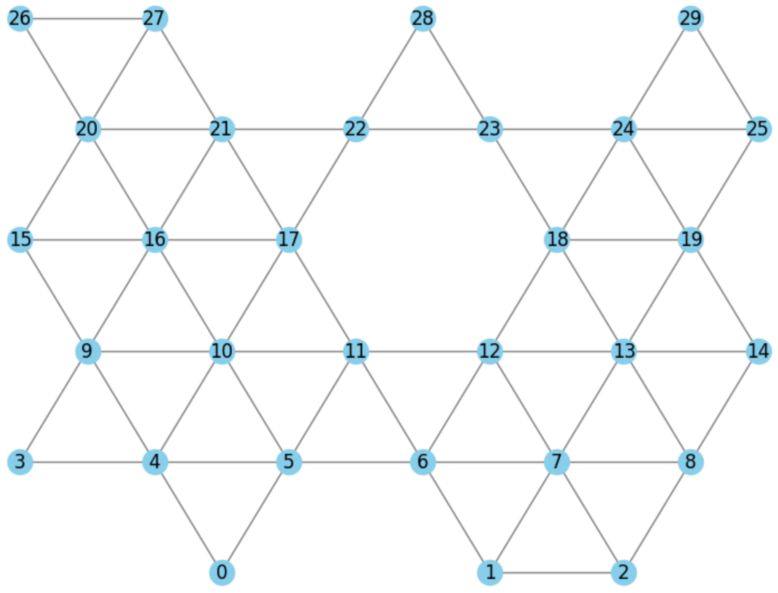


Figure 12: Network of 30 robots for testing the effects of seeding.

We initially run our queries with no seeding. In the simulations, 3/100 runs end up with consenting on decision 0, and UPPAAL estimates the probability of selecting this decision to be between about $4.6\% \pm 4\%$. This is all in line with the expected value of $1/n$. We then run a number of simulations, and get the following results.

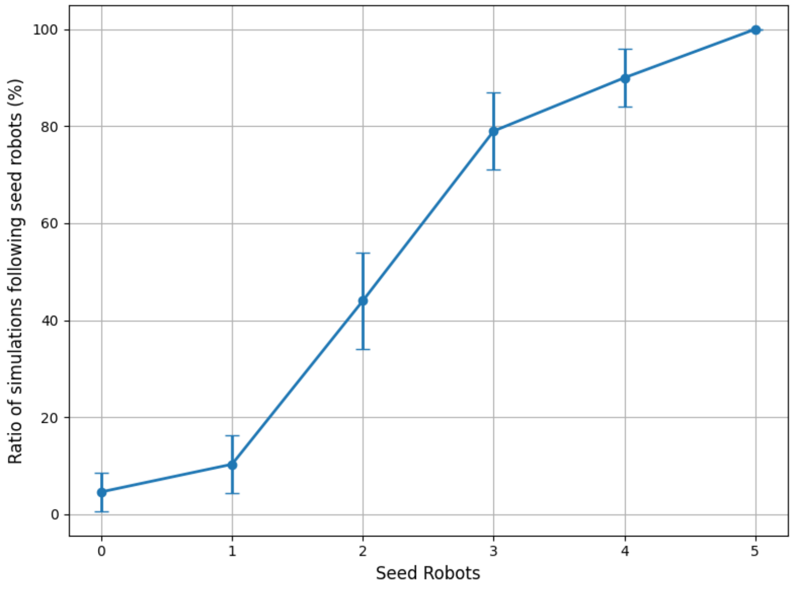


Figure 13: Effects of seeding on network consensus

From 5 seed robots and onward, the network always consents on the seeded decision.

5.3 Relation between expected behaviour and model results

In general, our results are very alike those presented in the algorithm article (Liu & Lee, 2019). We find similar correlations between the various variables and the iterations needed for the network to converge and consent on a singular decision. In some cases, we have been unable to replicate the exact experiment setup that was used in the original work, but even with the approximate setups we get similar results. In terms of our fourth research question, we conclude that the numbers match exceedingly well. For both network dependency values, network size, options and external interference, the results from our synthesized automata model closely conform to the proposed values from the original article.

6 Discussion

In this section, we reflect on the process and result of our work. From a software engineering perspective, we note some potential improvements that could be made to our tool, UPPAAL, based on our experience as newcomers. We also discuss a few missing specifications and assumptions of the original paper. We reflect on our approach to modeling, what stumbling blocks took a lot of time for us and what approach we have found to be helpful when modeling swarm robotics algorithms. Finally, we consider how our work may be improved and built upon.

6.1 UPPAAL as a modeling tool

We have benefited greatly from UPPAALs support for types, functions and templates. These features made it easier for us to translate our existing programming knowledge into modeling work - further aided by the C-like syntax. UPPAAL is a tool in development, and many useful and user-friendly features have been added in recent versions. The graphical editor and simulator also make it easy to inspect the model's behavior which greatly simplifies debugging.

However, UPPAAL suffers from outdated, lacking and at times incorrect documentation. There are official documentation pages, but many specifications, language idioms, and behaviors are distributed over presentations and tutorials written over multiple decades and major software versions – there are even some rules in the language we have not found documented at all. This leads to unexpected errors and inconsistent understanding of the rules of the language, which works counter to the purpose of formal verification. The lacking documentation encourages a "try and see" approach to writing functions and models, where understanding is derived from empirical observation of the simulator. Specifically for formal verification, it is crucial to be confident and exact in ones understanding of how changes to concrete syntax are translated into changes to the underlying model. Therefore, we argue that documentation is of special importance for a tool like UPPAAL.

Extending this observation, we found that while debugging and inspecting the model is easy, functions are not as well supported. Because functions are called atomically during edge traversals in the simulator, there is no way of inspecting evaluation, which makes it hard to catch errors and verify behavior.

In our specific case, another limitation of the platform became apparent. UPPAAL enforces a strict project structure. This means that the entire project is saved as a single XML file, and is only comfortable to edit using UPPAALs own GUI editor. Within the GUI, a project must have exactly one global declarations file, one declarations file per template, and one global declarations file. Combined, this makes it cumbersome to collaborate on projects, as merge conflicts are inevitable and complex projects cannot be broken down into components. The ability to modularise a project in some way would be very helpful for collaborative projects.

Finally, we note that UPPAAL ran into some severe performance issues on the MacOS machines used during modeling. This issue was not as apparent on the native Windows PCs.

We leave these observations for future researchers who consider using UPPAAL as their verification tool.

6.2 Quality of our model

The algorithm we chose to model is a best-of-n problem (Liu and Lee, 2019, section 1). While it is simple to measure the performance in terms of how fast the model achieves consensus, it is more difficult to evaluate the quality of that consensus given a network topology and initial distribution of preferences.

A very fast consensus algorithm is for every robot to always decide on choice 0. This is certainly not desired behavior, but by what metric should it be considered wrong?

The algorithm is presented as an improvement to simple majority rule, because it handles sparsely connected networks better. This seems correct, but clearly relies on some notion of global fairness or communication efficiency, in addition to simple cycle efficiency or speed that we do currently not know how to measure.

This means that one of the primary qualitative benefits of the algorithm remain opaque to us, as we do not have a metric to evaluate the performance of this algorithm compared to majority based algorithms, where robots "aim to gather direct information from as many other robots as possible." (Liu and Lee, 2019, section 1). Arguably, this algorithm does the same: The network communication efficiency claims may be true at the beginning of the process, where consensus groups are small. However, as our implementation of information-sharing illustrates, information has to traverse the entire network towards the end as the consensus process.

Liu & Lee do consider quality of result in section 3.4 in the specific case of controlling consensus by using seed robots, but otherwise do not.

In our verification, we include the expected value of the consensus landing on choice 0, which is an apt proxy for fairness in networks with completely random initial preference distributions. However, that metric does not allow us to investigate how network topology and non-random initial preferences affect quality of result.

While the network dependency value, the definition of which stems from the original article (Liu & Lee, 2019, section 3.2) was found to correlate with the amount of iterations needed to finish a decision process, we have also found that it is not a perfect metric. Often during random tests and setup, we would encounter somewhat large variance in the iteration counts, even if two networks had the same network dependency value, size and number of options. This suggests that there are other factors of network topology that could be very relevant for how the algorithm performs, and certain structures that may cause long, unintended processes that impede and slow down the decision making process. We have not been able to pin down any other specific, significant structure, but we theorize that they exist, and leave the discovery to possible future work.

Metrics or methods for evaluating fairness and quality are needed, and this is seen in the original report.

First, in section 3.1, the authors quantifies the consensus process using global entropy of the system. This seems to be an attempt at evaluating the quality of results. As the authors note: "The presented consensus algorithm aims to achieve a consensus with lower uncertainties in the preference distributions," (section 3.1) with a small entropy implying lower uncertainty.

A straightforward reading of this metric is to consider that falling entropy results in better performance in terms of cycles until global consensus This is supported by the statement that "the lower the value, the faster the convergence rate" (Liu & Lee, 2019, figure 3).

As described earlier in the report, we have modeled this entropy. We observe that entropy only falls when λ_T is sufficiently high and uncertainty reduction becomes a major factor in preference updates. However, networks are perfectly capable of reaching consensus with $\lambda_T = 0$, and in all networks we have tested, efficiency in terms of cycles and edge traversals is reduced as λ_T is increased. In extreme cases (high λ_T) the network may get stuck and fail to reach consensus within any time frame we have measured, as it settles into opposing locally converged camps – in which case lowering uncertainty seems to be in direct conflict with the promise of guaranteeing consensus (Liu and Lee, 2019, section 1).

This leaves us to consider that falling entropy may be a metric of some unspecified quality of result, but as the authors do not give explicit theoretical justification for this metric, we are unable to evaluate it on those grounds.

7 Conclusion

Through the translation of the algorithm presented in Liu and Lee, 2019 into UPPAAL, we have created a working model that adheres to the constraints of local knowledge and strict inter-robot communication. We have then formalized a suite of properties that act as sanity checks to verify expected behavior, and to record the results of experiments based on the work by the authors of the original algorithm. Based on the verification and analysis of these properties, we construe that the model adheres faithfully to the characteristics and expectations set in the original article, even if abstractions and additions had to be made. The verification results reveal that the model runs very similarly to the algorithm specifications, and we find very alike results for all tested variables. We have highlighted and described the use of UPPAAL for the purpose of implementing and checking this specific algorithm, and discussed various challenges, such as outdated documentation, lack of tools for collaboration and performance issues, that may guide future researchers when choosing their suite of tools for model applications. Finally, we have discussed the lack of metrics for sufficiently verifying the quality of an algorithm or a model, and how our research indicated that some of the metrics established in Liu and Lee, 2019 do not always improve performance in the intended way.

References

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. The MIT Press.

Brambilla, M., Pinciroli, C., Birattari, M., & Dorigo, M. (2012). Property-driven design for swarm robotics. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, 139–146.

David, A., Larsen, K., Legay, A., Mikučionis, M., & Poulsen, D. B. (2015). Uppaal smc tutorial. https://link.springer.com/article/10.1007/s10009-014-0361-y

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (pp. 11–15).

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Klarskov, P., Pedersen, M., & Bagge, A. (2025). Graphtool.ipynb. https://colab.research.google.com/drive/1skP0HkTQiTxBsxoD64tONoxlUlDsSI9I#scrollTo=iTt2s3O_8nB_

Konur, S., Fisher, M., & Dixon, C. (2010). Formal verification of probabilistic swarm behaviours. *Swarm Intelligence*, 440–447.

Larsen, K., David, A., & Behrmann, G. (2006). A tutorial on uppaal 4.0. https://uppaal.org/texts/new-tutorial.pdf

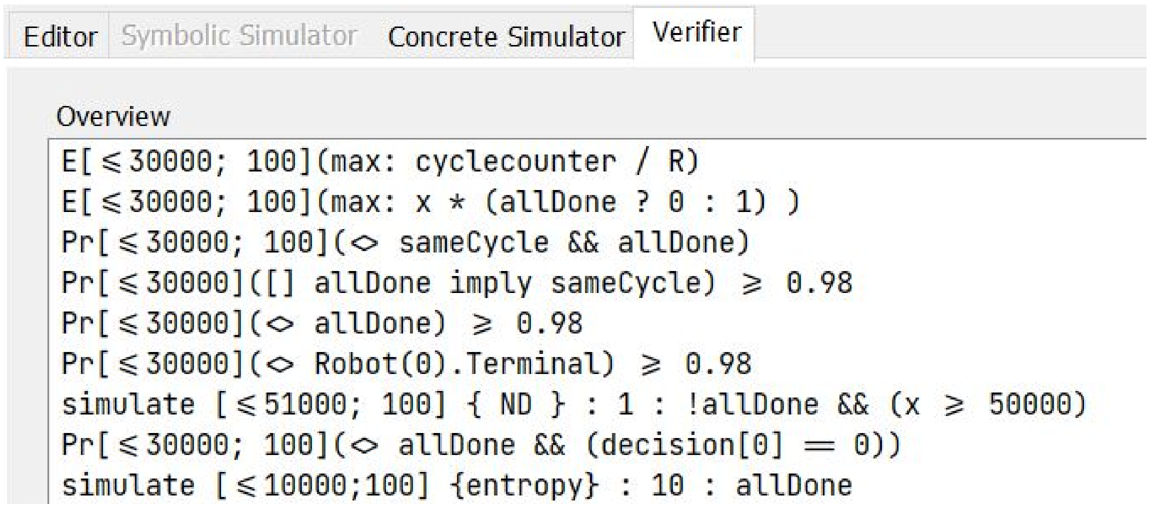
Liu, Y., & Lee, K. (2019). Probabilistic consensus decision making algorithm for artificial swarm of primitive robots. *SN Applied Sciences*, *2*. https://doi.org/10.1007/s42452-019-1845-x

Schober, P., Boer, C., & Schwarte, L. (2018). Correlation coefficients: Appropriate use and interpretation. *Anesthesia & Analgesia*, *126*, 1. https://doi.org/10.1213/ANE.0000000000002864

Webster, M., Western, D., Araiza-Illan, D., Dixon, C., Eder, K., Fisher, M., & Pipe, A. G. (2020). A corroborative approach to verification and validation of human–robot teams. *The International Journal of Robotics Research*, *39*(1), 73–99. https://doi.org/10.1177/0278364919883338

Appendices

A Queries



```
E[≤30000; 100](max: cyclecounter / R)
E[≤30000; 100](max: x * (allDone ? 0 : 1) )
Pr[≤30000; 100](<> sameCycle && allDone)
Pr[≤30000]([] allDone imply sameCycle) ≥ 0.98
Pr[≤30000](<> allDone) ≥ 0.98
Pr[≤30000](<> Robot(0).Terminal) ≥ 0.98
simulate [≤51000; 100] { ND } : 1 : !allDone && (x ≥ 50000)
Pr[≤30000; 100](<> allDone && (decision[0] == 0))
simulate [≤10000;100] {entropy} : 10 : allDone
```

B Networks

The networks can be found in the attached .txt files.