Networks of Realistic Robots

Adam Heriban

THÈSE DE DOCTORAT
DE SORBONNE UNIVERSITÉ

Spécialité : Informatique

École doctorale n°130: Informatique, Télécommunications et Électronique de Paris

réalisée

au Laboratoire d'Informatique de Paris 6

sous la direction de Sébastien Tixeuil

présentée par

Adam Heriban

pour obtenir le grade de :

DOCTEUR DE SORBONNE UNIVERSITÉ

Sujet de la thèse :

Réseaux de Robots Réalistes

sous la direction de Sébastien Tixeuil

soutenue le 09/12/2020

devant le jury composé de :

M^me^	Paola FLOCCHINI	Rapportrice
M.	David ILCINKAS	Rapporteur
M.	Quentin BRAMAS	Examinateur
M^me^	Maria POTOP-BUTUCARU	Examinatrice
M.	Sébastien TIXEUIL	Directeur de thèse

Contents

Acknowledgments

First, let me acknowledge that, had I not been fascinated as a child by the opening sequence of the 1986 movie Short Circuit, I probably would not have pursued Electrical Engineering, Robotics, and eventually Computer Science.

Let me thank Sylvie Delaët, who introduced me to mobile robots and self-stabilization back in 2016, which eventually lead me to choosing this subject.

I want to thank my advisor, Sébastien Tixeuil, for bearing with me during these almost four years, for his guidance and his advice.

I want to thank TEAM and Erasmus Mundus, for the incredible opportunity to spend a year of research in the gorgeous landscapes of Kansai. I am extremely grateful to Michiko Inoue and Fukuhito Ooshita, and all the NAIST personnel I had the pleasure to interact with for their invaluable help and hospitality, and to Xavier Défago for his hospitality at TiTech, and his help during our collaborations.

For accepting to take the time to review this thesis, and for their thorough feedback, I want to thank David Ilcinkas and Paola Flocchini.

For their decisive influence on my career choices, I want to thank the administrative staff of Sorbonne Université and EDITE de Paris

I want to give credit to my desktop computer, which spent all of the 2019 summer processing billions of simulations day and night during the scorching heat waves, and survived to this day.

Because he had to endure me during the past decade, I want to thank my dear friend Bertrand.

Finally, and most importantly, I thank my spouse, Magalie, for her unconditional support, keeping me sane during all these years.

Chapter 1

Introduction: Networks of Realistic Robots

1.1 Distributed Robotics

In the far future, you are ALICE, newly appointed leader of a technologically advanced civilization, and you just declared war on the kingdom of BOB. But, this is not the 20th century anymore, you do not send *people* to war. You send robots.

How do you make sure the hundred of thousands of robots you are sending to some unknown planet actually follow your orders? Charlie, one of your advisors comes up with a brilliant plan: build a massive computer-spaceship and put it in orbit. Then, robots send their sensor data in real-time to this central brain, which processes everything and sends back commands to the robots on the ground.

Because you have matters to attend to – civilization management is not trivial business – you accept, allot the funds and stop thinking about it. A few weeks later, you hear from Charlie that the armies of BOB did not, in fact, need to fight the robots on the ground. For some unclear reason – possibly involving child-soldiers – the computer-ship was destroyed, which, in turn, disabled all your billions-worth of robots[1]. You jettison your advisor with no oxygen.

This scenario is actually very representative of a fundamental problem when dealing with multiple 'agents', or robots: how to coordinate everyone?

The key flaw of your late advisor's proposal was *centralization*: every single robot in your army relies on the proper function of the computer-ship: a single point of failure.

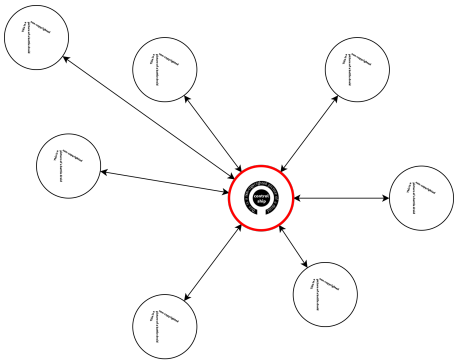


Figure 1.1 – A Centralized Network of Combat Robots
Destruction of the central computer-ship leads to a global shutdown

[1] This may sound familiar to people who watched a certain 1999 movie.

Your newly appointed advisor, Dave, suggests another approach: instead of a single, massive and fragile central brain for the entire army, each regiment of a thousand robots is given a local, smaller computer-ship, which are then all connected to coordinate the offensive. This means better latency, and taking down the entire army requires taking down dozens of well defended computer-ships. Your robot army would then be *decentralized*.

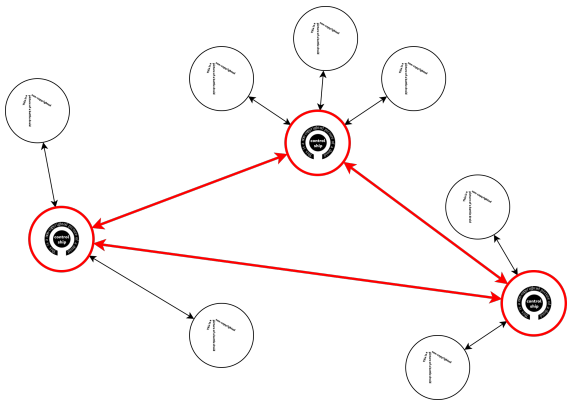


Figure 1.2 – A Decentralized Network of Combat Robots
Destruction of a computer-ship only hinders parts of the network

But you are now wary, and hiring a new advisor is a slow and expensive process. You tell Dave that these computer-ships are still points of failure, and so, priority targets, even if there are now more of them. You want **no** point of failure. Each robot must fight until the end. Each robot must be able to use the data from its sensors, process it, and coordinate with its neighbors to decide what to do next. You want the decision making to be entirely *distributed*.

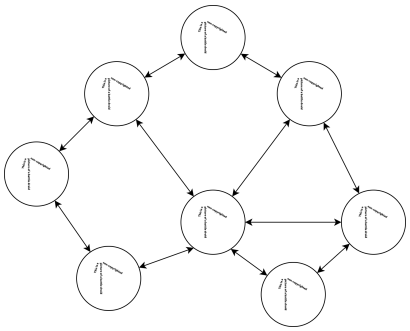


Figure 1.3 – A Distributed Network of Combat Robots
No single entity is more important than another

Dave, who recently learned the reason of his recruitment, is skeptical of your idea: how would a robot on one side of the battlefield know not to fire on the robots on the other side? How would you even make sure that when robots try to make a decision as a group, an agreement is eventually reached?

Have you seen what happens when a group of friends try to decide which movie to watch?

Fortunately, you stumble upon this manuscript of ancient distributed robotics science.

1.2 Motivation

One of the most pressing issues in current distributed robotics is the divide between the theoretical networks of robots being studied by computer scientists on one side, and the actual robots being built by robotics engineers on the other.

This is due to each group having very different design philosophies. To solve an issue for theoretical networks, we assume an algorithm and a set of hypotheses governing the robots, and prove whether or not the algorithm solves the issue under the given hypotheses. From an engineer's standpoint, this approach, while mathematically exact, is not necessarily the most useful. Engineering specifications are not necessarily compatible with formal hypotheses, and even the simplest robots are extremely complex, and models do not encompass their full behavior, as this would be orders of magnitude too complex for formal proofs[2]. But, from the point of view of a computer scientist, this could be seen as reckless. A million simulations does not guarantee that the worst case scenario has actually been found, and no set of tests of actual components can fully represent the entire set of conditions the whole system may be subject to[3].

Overall, there is no *"single best approach"*. There is always a level of details that falls outside the scope of formal proofs, and the more complex the systems, the less useful simulations and test become. However, the prevalence of formal proving has been increasing during the past two decades, as the tools become more accessible, easier to use, and allow for more complex systems to be formally proven.

As such, the more long term approach is to improve the viability of formal methods for proving actual systems.

We decide to investigate the current existing realistic models that can be formally used, and develop new such models to decide which ones should be used in the long term to bridge the realism gap between theoretical networks of robots and actual robots.

1.3 The $\mathcal{OBLOT}$ Model

The seminal paper for studying robotic swarms from a Distributed Computing perspective is from Suzuki and Yamashita [83]. They introduce a mathematical model for studying geometric pattern formation by networks of n possibly oblivious robots.

In the Suzuki and Yamashita initial model, commonly named $\mathcal{OBLOT}$, robots are represented as dimensionless points evolving in a bidimensional Euclidean space (that is, $\mathbb{R}^2$), and can accumulate on the same location. They operate in **LOOK-COMPUTE-MOVE** (**LCM**) cycles. In each cycle, a robot "Looks" at its surroundings and obtains (in its own coordinate system) a snapshot containing some information about the locations of all robots. Based on this visual information, the robot "Computes" a destination location (still in its own coordinate system), and then "Moves" towards the computed location. The robots are oblivious, so the computed destination in each cycle only depends on the snapshot obtained in the current cycle (and not on the past history of actions). The motivation for using oblivious robots is their apparent fault-tolerance: for example, a robot's memory might become corrupted, so ignoring past actions makes algorithms inherently more reliable. The visual snapshots obtained by the robots are not necessarily consistently oriented in any manner. Then, an execution of a distributed algorithm by a robotic swarm consists in having every robot repeatedly execute its LCM cycle. In general, executions are infinite (even if robots do not move after a while, they still look, compute and decide not to move) and fair (every robot executes an infinite number of LCM cycles).

Although this mathematical model is perfectly precise, it is deeply unrealistic. First, robots are endowed with infinitely precise, and instantaneous sensors. Even assuming such sensors existed, each snapshot would require an infinite amount of volatile memory to be used. Second,

[2] See most side-channel attacks on proven systems

[3] See the Ariane 5 maiden flight

movement is also infinitely precise, which again, is not actually feasible. Finally, the original $\mathcal{OBLOT}$ model robots required some sort of synchronization mechanism, which implies the existence of a centralized clock, which in turn contradicts the distributed nature of mobile robots. However, thanks to its simplicity, this model also allows for a great number of variants (developed over a period of more than 20 years by different research teams [43]), which we investigate now [28].

1.4 More Realistic Mobile Robots

1.4.1 Sensors

Robots perceive their surroundings through sensors, whose abilities have strong impact on task solvability. The most commonly considered types of sensors vary along several capabilities, described below. Obviously, there are other kinds of sensors not described here. For instance, in a completely opaque environment, one may imagine that the only available information is by direct contact through bumpers.

Range

The most obvious parameter of sensors is their range, that denotes how far a robot can sense another robot's location:

- **Full visibility:** robots are able to sense every other robot's location, regardless of distance.
- **Limited visibility:** there exists $\lambda > 0$ such that a robot is able to sense every other robot's location if their distance to the observing robot is smaller than λ, and is unable to sense other robots locations [3, 46]. This model is also called myopic robots.

Note that in general, robots are not aware of λ. Obviously, if an algorithm can achieve a task under limited visibility, it can also achieve it under full visibility.

Multiplicity Detection

Multiplicity refers to the ability of robots to distinguish (to some extent) the number of robots sharing a given location. There are three variants of multiplicity detection, depending on how accurate it is, ordered by decreasing strength:

- **No multiplicity detection:** sensors can only distinguish occupied and unoccupied locations, but no information about the number of robots at each occupied location is known.
- **Weak multiplicity detection:** sensors can distinguish between unoccupied locations, locations occupied by a single robot and locations occupied by more than one robots, but cannot know the exact number of robots in the second case [60].
- **Strong multiplicity detection:** sensors can accurately count the number of robot at any location.

Multiplicity detection can also vary depending on range:

- **Local multiplicity detection:** indicates that an observing robot can only obtain multiplicity information about its own location [60].
- **Global multiplicity detection:** indicates that an observing robot can obtain multiplicity information about all locations within its viewing range.

Overall, we thus have five variants for multiplicity: no multiplicity, weak local multiplicity, weak global multiplicity, strong local multiplicity, and strong global multiplicity. Obviously, an algorithm assuming no multiplicity detection is more powerful than one requiring any of the other assumptions. However, some assumptions are not comparable, e.g. weak global multiplicity and strong local multiplicity.

Orientation

Orientation refers to the ability of robots to share some common notion of direction or orientation. Again, there are many variants:

- **No common direction:** robots share no common axis and may not agree on handedness.
- **Common direction:** robots have a common axis (such as the North-South axis) but the direction along these axes may be inverted. Because robots use a Cartesian coordinate system, a second common direction can immediately be deduced. This model is also called two-axes direction.
- **Common orientation:** in addition to having the same two axes direction, robots may also share orientation on either one axis (e.g. North only) or two axes (e.g. North and West).
- **Common chirality:** robots have the same notion of clockwise and counter-clockwise.

Notice that it is entirely possible to have common chirality without sharing a common direction. When having common direction, orientation on one axis, and chirality, robots are said to have **full compass**. Note that orientation on one axis and chirality is equivalent to having orientation on two axes.

Consistency

Depending on the model, robots may not be able to keep their own coordinate system consistent between two LCM cycles. This yields two separate models:

- **Inconsistent coordinate system:** the local Cartesian coordinate system of a robot is not consistent between two LCM cycles.
- **Consistent coordinate system:** the local Cartesian coordinate system of a robot is consistent between two LCM cycles. In particular, each robot always uses the same local distance unit and axes throughout the execution.

Errors

To study the impact of inaccurate sensors, we consider three different models for vision error. For a robot r_1 looking at a robot r_2 located in (x,y) in the Cartesian coordinate system centered at r_1, and located at (r,θ) in the polar coordinate system centered at r_1, we define:

- The **absolute** error model [68] uses a constant value err so that the perceived position of r_2 is in the disc centered of r_2 of radius err.
- The **relative** error model [25] uses two constants err_{dist} and err_{angle}. The polar coordinates of r_2 are then perceived to be in $(r + r \cdot [-err_{dist}, err_{dist}], \theta + [-err_{angle}, err_{angle}])$
- The **absolute-relative** error model is similar to the relative error model, but the perceived polar coordinates are $(r + [-err_{dist}, err_{dist}], \theta + [-err_{angle}, err_{angle}])$

These models are illustrated in figure 10.1.

There appears to be no obvious hierarchy in difficulty between these models.

Another type of errors has been considered in the case where robots are assumed to have a full compass. In this case, there may be an error on the orientation of the axes perceived by each robot. Two different types of inaccurate compasses are introduced [61]. They both require an upper bound *err* so that:

- In the **static** error model, each robots picks an offset in $[-err, err]$ at the beginning of the execution which is always applied to the axes.
- In the **dynamic** error model, each robots picks an offset in $[-err, err]$ at the beginning of each LCM cycle, which is then applied to the axes.

Given the same upper bound, any algorithm that can achieve a task in the dynamic model obviously can also achieve it in the static model.

1.4.2 Transparency and Size

The most common models assume robots to be transparent. In other words, when multiple robots are collinear, every robot can see all other robots.

- **Transparent punctual:** robots are points, or discs with a null diameter, and do not impede each other's vision.
- **Opaque punctual:** robots are points, and if three robots are collinear, the middle robot prevents the other robots from seeing one another. This model is also called obstructed vision.
- **Transparent Disc:** robots are discs of non-null diameter, and do not impede each other's vision.
- **Opaque Disc:** robots are discs of non-null diameter, and a robot r_1 can only see a robot r_2 if there are no robots between r_1 and r_2. Note that some models require full vision of r_2 for detection [3], while others require only partial vision [29, 58]. This model is also called solid robots.

1.4.3 Environment

The original model for mobile robots [83] assumes the robot to be moving in an **infinite, continuous Euclidean plane**. However, this model allows for configurations that may not exist in a world of imperfect sensing and movement. A second model, where robots move on a **discrete graph**, has been introduced, and acknowledges the potentially discrete nature of robot snapshots. However, these two models do not entirely overlap. Some problems, such as Gathering, exist in both the continuous [42, 83] and discrete [10, 23] settings. Some problems, such as Exploration [59] can only exist in the discrete setting, while others like Convergence can only exist in the continuous [42] setting. Similarly, some variants of the $\mathcal{OBLOT}$ model such as limited visibility have been also used in discrete models [62, 72].

1.4.4 Memory and Communication

The benefit of using oblivious robots is that they easily recover from crashes and memory corruption. Nevertheless, several extensions with memory have been proposed, ordered from strongest to weakest:

- **Oblivious:** only volatile memory is available. Memory is reset at the beginning of each LCM cycle. Robots thus have no memory of past actions. Practically, robots can only use their current snapshot in the COMPUTE phase.
- **Finite memory:** robots may have persistent memory between LCM cycles. We investigate a variant of this model called the $\mathcal{LUMINOUS}$ model further in the following chapter.
- **Infinite memory:** robots may make use of an infinite amount of memory. This allows robots to remember a full snapshot, as robot position have to be encoded as actual real numbers (in each robot's own coordinate system).

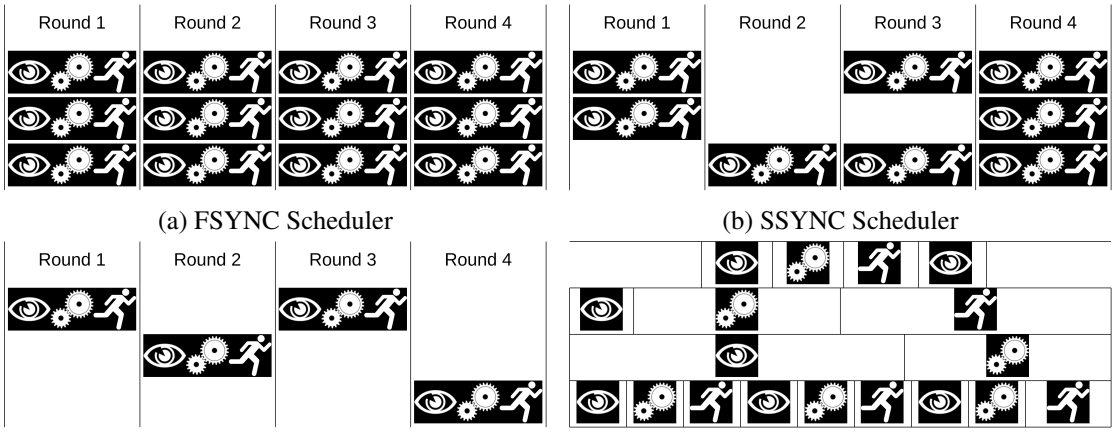
Since robots are assumed to be anonymous, there is no way of performing point-to-point communication with a particular neighbor. Therefore, communication is handled through broadcast, which is described as the robots having lights whose color may be adjusted during the COMPUTE phase. In addition to the number of available colors for a robot's light (hence the amount of states of information transmitted), there are three models of $\mathcal{LUMINOUS}$ robots:

- **Internal lights:** are only visible by the emitting robot itself, thus actually represent finite memory (the robot only communicates with itself).
- **External lights:** are only visible by other robots but not the emitting robot, thus they represent communication without memory.
- **Full lights:** combine internal and external lights: they are visible by all robots.

1.4.5 Synchronicity

The considered model is based on discrete logical time, that is, on a sequence of events, an event being any change in the conformation of any robot. The possible interleavings of those events define the synchronicity level of an execution. If the LCM cycles are considered atomic, that is no event can occur during a cycle, the model is said to be **semi-synchronous** (SSYNC): a non-empty subset of the network enters (and finishes) their cycle and each phase within it simultaneously while the others are idle, hence the notion of round. In the constrained version of SSYNC where no robot is ever idle, that is where all robots are activated simultaneously, the execution is said to be **fully-synchronous** (FSYNC). In the case where the cycles are not atomic and may overlap, the execution is said to be **asynchronous** (ASYNC) [45]. Clearly, ASYNC is the strongest model and FSYNC is the weakest. A fourth synchronicity model has been considered: **Centralized**. It is a subset of the SSYNC scheduler for which the subset only ever includes a single robot. It is therefore weaker than SSYNC but is not comparable to FSYNC.

These synchronicity hypotheses between the LCM cycles of robots are of paramount importance for proofs. Many proofs made in weak synchronization models were claimed to hold also under stronger synchronization models but turned out to be incorrect. This is actually the main source of errors in the literature [1]. In the FSYNC, SSYNC, and Centralized models, the actual duration of each phase does not matter since no observation occurs while a robot is moving, which justifies using discrete logical time. On the other hand, the ASYNC model represents the complete lack of synchronization between robots, and the duration of each phase is important, as a robot may observe others while they are moving.



(a) FSYNC Scheduler (b) SSYNC Scheduler

(c) Centralized Scheduler (d) ASYNC Scheduler

Figure 1.4 – Synchronization Models [28]

1.4.6 Fairness and Boundedness

In all models except FSYNC where all robots are active at all times, the subset of active robots is chosen by the environment. In all generality, nothing prevents the environment, a.k.a the demon, or scheduler, from starving some or all robots. Obviously, never activating certain robots is akin to triggering crash faults (see below). Thus, there are fairness constraints on demons. First, no demon can prevent progression of the network. In other words, if all robots do nothing and wait for one robot to be activated, the demon must eventually activate this robot, as otherwise, demons could simply prevent any algorithm from progressing by 'crashing' the entire network. A demon is **fair** if every robot is activated infinitely often. This is equivalent to saying that, for every robot r_1 and time t, there exists $t' > t$ at which r_1 is activated. Although fairness is usually enough for most protocols, it does not give any guarantee on the relative rates of robots activations: a robot may be activated arbitrarily more often than another. We introduce the boundedness condition: a scheduler is **k-bounded** if a single robot cannot be activated more than k times between two activations of other robots. In other words, a robot r_1 can be activated at most k times before another robot r_2 must be activated. In the case where k is not known by robots, the scheduler may simply be called **bounded**.

A scheduler that is both **k-bounded** and **fair** is **k-fair**: each robot is activated at least once, and between two successive activations of a robot r_1, every other robot has been activated at most k times.

Note that boundedness is not directly related to fairness for network of $n > 2$ robots.

1.4.7 Rigid Motion

The atomicity of cycles does not imply that the computed destination is actually reached by a robot before the start of its new cycle: the robot may be interrupted during its move by the environment. An execution where all robots always attain the destination returned by the protocol is said to be **rigid**. Conversely, if robots can start a new cycle before they completed their scheduled journey, the execution is **non-rigid**. In such a case, so as to avoid Zeno-like counter-examples, it is assumed that robots travel at least some minimal distance $\delta > 0$ towards the expected destination before stopping. In particular, any destination closer than δ is always reached. This minimal uninterruptible distance is unknown to robots most of of the time. They may however take into account that such a minimum exists.

1.4.8 Faults

In an adversarial environment, faults must be considered, either because malicious agents are present or because one agent has been corrupted. The most common fault hypotheses made in models are (from strongest to weakest):

- **Byzantine Faults:** some robots do not follow the protocol and are controlled by an adversary.
- **Crash Faults:** some robots may crash and never be activated anymore.
- **No faults:** every robot follows the protocol forever.

Notice that the faults described above are permanent. We also consider **transient faults** for which a robot's state may become arbitrary following a one time corruption. Note that this is equivalent to starting from an arbitrary configuration where robots may carry pending actions. Algorithms that are resilient to transient fault, and so, that can function from any initial configuration, are **self-stabilizing**. Unless specified otherwise, all algorithms considered in this thesis are designed to be self-stabilizing.

1.5 A Realistic Example: Collision Avoiding Blind Robots

In 2007, Yared *et al.* [86] introduced a model for blind mobile robots. This model heavily differs from the $\mathcal{OBLOT}$ model on several points:

- Blind robots do not have a visual sensor.
- Blind robots are able to send and receive messages.
- Blind robots are endowed with a position and angle sensor which allows robots to know their absolute position and orientation in the euclidean plane.
- Blind robots' sensors are not infinitely precise and have errors with known upper bounds.
- Blind robots do no move with absolute precision, but can make an error in both distance and angle.
- Blind robots are not points, but solid discs moving in the euclidean plane.

These hypotheses encompass multiple models we have defined previously. Most notably absolute sensor error, a movement error similar to the absolute-relative model for vision, and solid robots. The sensors model both inaccurate position and compasses, similar to a GNSS (Global Navigation Satellite System, such as GPS, GLONASS, BeiDou or Galileo) sensor for actual robots. The hypothesis of using no visual sensors also has merit in terms of realism.

Under these hypotheses, Yared *et al.* describe and prove an algorithm to ensure movement with no collisions under the ASYNC scheduler. This algorithm relies on robots agreeing on reserved zones in which they could be located given the known upper bound of their error in both location and motion (see figure 1.5).
As we have noted, this model is extremely realistic when modeling cooperating mobile agents. Therefore, we hope to extend the presented algorithm to even more realistic settings. In particular, we wish to improve the algorithm to be able to deal with transient faults, crash faults, and even byzantine faults. This algorithm in particular is explicitly unable to tolerate crash faults.

However, after closer inspection of the algorithm, two issues arise. First, the algorithm relies on a discovery primitive *NDiscover* to establish how many robots are in the vicinity. However, upon closer inspection, this primitive requires that if a robot r_2 is located within a

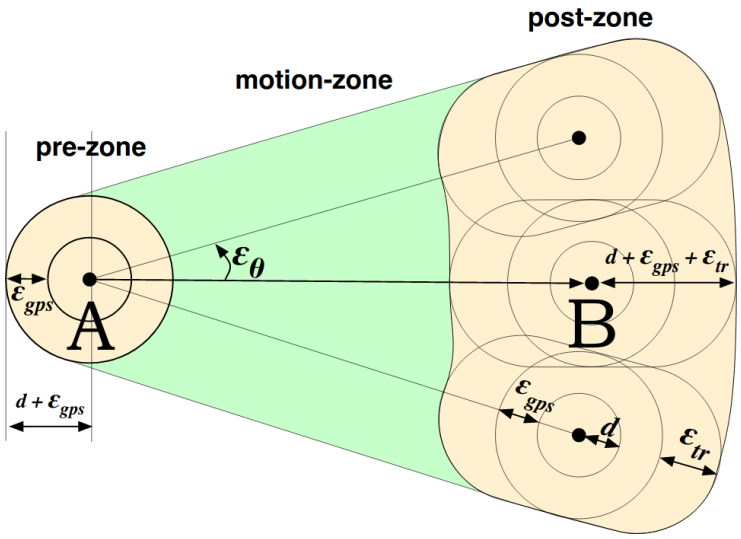


Figure 1.5 – Reservation Zone Given Known Error Upper Bounds

defined area, and the primitive is used by robot r_1 on this area, then after a known bounded time delay, robot r_1 is aware of the presence of robot r_2. If r_2 is crashed, r_1 should be aware of its presence but infinitely wait its answer to any message sent. So this primitive is actually based on a quasi-perfect failure detector. Therefore, for this step in particular, the algorithm does not actually function under the ASYNC scheduler. If we accept the premise that the scheduler in not completely asynchronous, and that bounded-time responses can be expected, then a perfect failure detector could be implemented, and crash faults could be dealt with.

So, the scheduling model in use already could tolerate crash fault, but is not actually ASYNC.

On the other hand, when looking at byzantine fault tolerance, a second problem arises. This algorithm relies on the basic premise that robots communicate their possible location (within bounded error) to robots around them so that they can decide whether or not a collision could occur. So the algorithm relies on absolute trust in every robot in the network, since robots have no sensor to verify the location messages they receive. Furthermore, since robots communicate their location every time they move, a byzantine robot would know where other robots are located in real time and could easily move towards their reserved area to trigger a collision. Therefore, we can trivially conclude that, under this fully cooperative model, any byzantine interference would trivially lead to a collision. Actually, collision avoidance against byzantine mobile agents is trivially impossible because a byzantine agent could home towards the closest robot, making the problem akin to avoiding a homing missile.

Because of this last point, we realize that, while extremely interesting, because of the limited capabilities of these realistic robots, some models may not be expanded to new settings. A realistic model which prevents the most basic problems from being solved is not a good compromise between mobile robots and actual robots.

So, we need to strike a balance between more realistic robots, and capable robots.

1.6 Our Contributions

We develop a new, optimal algorithm for 2-robot **Gathering**, or **Rendezvous**. However, because of major difficulties encountered when attempting to prove this algorithm with pen and paper, we build and prove a complete model checking framework for 2-robot **Gathering** in the continuous plane based on the SPIN model checker. We confront known state-of-the-art **Rendezvous** algorithms and find results consistent with the literature. We also introduce a new model for colors which matches the common notions of *safe*, *regular* and *atomic* registers and

test a **Rendezvous** algorithm for regular lights. Similarly, we use lights to build more robust **Leader Election** algorithms, which allow for stricter constraints on the election.

We then designed a new vision model for mobile robots, named *Uncertain Vision*, which introduces a vision adversary for sensors to register false negatives, *i.e.* not see robots that are actually there, and proved tight bounds under this new model for various benchmark problems. We also focus on the already existing obstructed visibility, or opaque robot model, and define a new problem, **Obstruction Detection**, which requires robots to not move, and compute which visible robot is obstructing them from seeing another robot. However, after proving several fundamental results and two unsuccessful attempts at solving the problem, including an algorithm based on token transmission, we acknowledge the massive difficulty inherent to both the problem and the model itself.

Because of how difficult working with more complex algorithms under realistic models turns out to be, we decide to change our approach: we develop a framework for Monte-Carlo simulations of mobile robots from the ground up. This framework is modular which allows us to simulate any robot model, scheduler or algorithm with minimal effort. This simulator is not a model checker and has known limitations. As such, it should currently be viewed as a replacement for researcher *"intuition"*, and used to look for unexpected behavior in mobile robot networks that would be then verified using formal techniques. We find several result which had not been predicted by pen and paper analysis, but can be confirmed after closer inspection of the published algorithms. We then use this simulator to implement and test errors in vision. We demonstrate that **Geoleader Election** is not possible in this vision model. Some details in behavior of **Rendezvous** algorithms in this error model remain to be explained. Finally, we introduce another two algorithms: the first algorithm uses two colors to ensure **Convergence** for two robots and guarantees the distance traveled is minimal ; the second allows for **Leader Election** with errors in vision: robots use the simulator itself to verify for possible errors in the election and move randomly if an error is detected. This particular design philosophy can be used to adapt some algorithms to function in a continuous setting using discretized snapshots, and therefore can be used to realistically implement the **SyncSim** protocol [31] to simulate a FSYNC scheduler in $\mathcal{LUMINOUS}$ ASYNC.

1.6.1 Published Work

Our two-color algorithm for ASYNC **Rendezvous** presented in chapter 2.1 was published with its original proof at ICDCN 2018 [53]. An extended version is currently under review for publication in TCS.

Our model-checking system for verifying **Rendezvous** algorithms presented in chapter 3.2 was first published as a brief announcement at DISC 2019 [33]. The full version was published at SRDS 2020 [34].

Our new model for uncertain vision presented in chapter II was first published at SIROCCO 2019 [55]. The full version was accepted for publication in PPL [56].

Our preliminary results for **Obstruction Detection** presented in chapter 6.5 were presented at the IEICE COMP / IPSJ-AL 2018 workshop [54].

Part I

The Power of Lights

Chapter 2

The $\mathcal{LUMINOUS}$ Model

The idea of endowing mobile robots with lights was first suggested by Peleg [76] in 2005. This idea stems from the disturbing results that the $\mathcal{OBLOT}$ model brings along with its extreme simplicity. Peleg first suggests introducing O(1) bits of memory to the previously purely oblivious model, or a simple communication system such as colored flags or *lights*.

In 2012, Das *et al.* [30, 31] introduced the $\mathcal{LUMINOUS}$ model, based on Peleg's suggestions, and demonstrated its potential.

The model reads as follows: each robot carries a *'light'* which can emit one color among a given, finite, set. At the end of its COMPUTE phase, the robot can change the color of its light according to the algorithm it is executing. When performing a LOOK, a robot not only perceives the position of other robots, but also the color of their light.

This results in four possible models for memory and communication between robots:

- Oblivious: robots have no memory and cannot communicate.
- FSTATE or *internal light*: robots carry O(1) bits of memory they can read and write to, but cannot communicate.
- FCOMM or *external light*: robots can emit a light of a given color, but cannot read the color of their own light.
- The 'FULL-LIGHT' model is, where robots can emit a light of a given color and read the color of their own light.

The *full-light* model is the most common, and is the one we study unless specified otherwise.

This new model yields important results [31]: first, a robot endowed with a light and 5 colors working under the ASYNC scheduler is more powerful than an oblivious robot working under the SSYNC scheduler. In other words, the $\mathcal{LUMINOUS}$ ASYNC robot can perform all the tasks that the oblivious SSYNC robot can, and there are tasks only the $\mathcal{LUMINOUS}$ ASYNC robot can perform. Using this result, we deduce that any $\mathcal{LUMINOUS}$ SSYNC algorithm can be run successfully by a $\mathcal{LUMINOUS}$ ASYNC robot. Second, a robot endowed with a light and 3 colors **and the ability to remember its previous snapshot**, working under the ASYNC scheduler is more powerful than an oblivious robot under the FSYNC scheduler. Whether or not ASYNC $\mathcal{LUMINOUS}$ robots are more powerful than oblivious FSYNC robots remained an open question until now.

From the point of view of more realistic robots, this model is extremely interesting. It is paramount that any model for "more realistic robots" we introduce does not impede the ability of robots to perform the most basic tasks. Realistic models need to balance more realistic hypotheses with capability. $\mathcal{LUMINOUS}$ robots perfectly fit this requirement, as they are realistic in bringing in memory and communication to the $\mathcal{OBLOT}$ model, and they massively increase the capabilities of the robot network. In particular, being able to simulate a FSYNC scheduler using only finite memory and communication would be an elegant way to solve any

problem under the ASYNC scheduler.

However, this new model raises two issues:

- As discussed in section 1.4, a single snapshot of infinite accuracy requires an infinite amount of memory to store.
  - This actually does not make the current model less realistic, as robots already need infinite volatile memory to process their snapshot in their COMPUTE phase.
  - We further discuss this issue in section 11.2
- Using Lamport's register terminology [64], lights represent atomic memory and communication, which is not a realistic assumption to make in ASYNC.
  - This actually does not make the current model less realistic either, as robots already carry a certain level of atomicity in communication through their instantaneous LOOK phases.

We decide to keep using the LUMINOUS model throughout the rest of this thesis, as it provides an elegant and simple model to manage memory and communication without necessitating many additional assumptions.

2.1 OBLOT FSYNC versus LUMINOUS SSYNC

In 2012, Das *et al.* [30, 31] show that some problems can be solved by a LUMINOUS SSYNC robot, but not by an oblivious FSYNC robot. However, it remained an open question whether or not a LUMINOUS SSYNC robot could perform all the tasks an oblivious FSYNC robot could. If this was true, then any problem could simply be studied under the FSYNC model, and then solved under the LUMINOUS ASYNC model.

Theorem 2.1. *There exists problems that can be solved under the oblivious FSYNC model and cannot be solved under the LUMINOUS SSYNC model.*

Proof. First, we look at the fundamental differences between the FSYNC and SSYNC model.

Property 2.1 (Identical Inputs)**.** *Throughout the execution, the sequences of perceived positions and colors of the network are identical for all robots, with respect to their own coordinate systems.*

Property 2.2 (Simultaneous Action)**.** *Given a configuration C, all robots in C are in the same phase.*

Lemma 2.1. *Under the full visibility model, properties 2.1 and 2.2 are true for the FSYNC oblivious model, and false for the SSYNC LUMINOUS model.*

Proof. By definition of the FSYNC model, robots' phases are always simultaneous, so properties 2.1 and 2.2 are trivially true.

To show that property 2.1 is false for the SSYNC LUMINOUS model, let us simply consider a configuration in which at least one robot r_1 can change its state, *i.e.* move or change color, at its next activation. Let the scheduler have robot r_1 perform a full cycle and not activate any other robot in the network. Let the scheduler now activate another robot r_2 to perform a full cycle. Since robot r_1 changed its state the sequence of snapshots of r_2 does not include the same snapshots as r_1. So, as long as the algorithm allows for a change of state, property 2.1 is false.

Property 2.2 is similarly false for the SSYNC LUMINOUS model whenever the scheduler only activates a subset of robots.

□

Let us now build a problem that cannot be solved under the LUMINOUS SSYNC model because of property 2.2.

Definition 2.1 (Balancing Problem)**.**
Let us assume a network of three robots with rigid motion.
The robot with the smallest angle between the two other robots is the **LEADER**.
The unit vector is defined as the altitude from the opposite base to the **LEADER**.
We define the base opposite to the **LEADER** *as having a y coordinate of* 0*. So the* **LEADER** *has a y coordinate of* 1*, following the unit vector.*
The problem is solved if:

1. *The y coordinates of* **NON-LEADER** *robots are* ***always*** *identical.*
2. *All robots are eventually located at coordinates* $y = 1$.

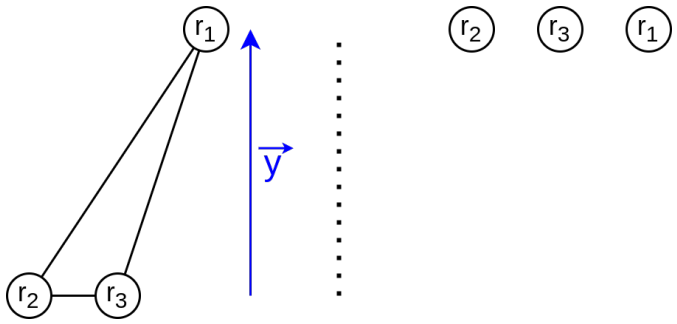


Figure 2.1 – Example of a Solved Instance of the Balancing Problem.

Lemma 2.2. *There exists configurations for which the* ***Balancing*** *Problem can be solved for the FSYNC oblivious model and cannot be solved for the SSYNC LUMINOUS model.*

Proof. For both models, the problem is not defined in the case of an equilateral triangle, or the case of an isosceles triangle where the angle of the apex is greater than $\frac{2 \cdot \pi}{3}$, as the **LEADER** is not defined.
Let us consider a configuration where robots are not forming any such triangle and not initially collinear. In FSYNC Robots first compute which robot is the **LEADER**. **NON-LEADER** robots are at $y = 0$ and move a distance of 1 along the $\overrightarrow{y}$ axis. The problem is solved.

We now show the impossibility for the SSYNC LUMINOUS model.
Let us assume, for the purpose for contradiction, that it is possible to solve the problem in SSYNC using luminous robots. For this algorithm to be successful, any execution must include a configuration where the **NON-LEADER** robots are located at a y coordinate of 0, such that they both move to a different y coordinate after their next activation. Otherwise, if robots are not configured in a way that they all move to a different y coordinate when all activated, then either they do not change their y coordinate and the problem is not solved, or they move at different times, which is a violation of the 'always identical' condition. So the configuration must exist for any execution.
Let us now consider this configuration and only activate a single base robot. It moves to a different y coordinate while both other robots stay at 0. The 'always identical' condition is violated so the problem cannot be solved.

□

□

Note that this counter example relies on property 2.2, and extreme safety conditions. Regarding property 2.1, the question remains if there exists problems that can be solved in oblivious FSYNC but not in $\mathcal{LUMINOUS}$ SSYNC, if for all problems that can be solved in oblivious FSYNC, there exists an algorithm that also solves it in $\mathcal{LUMINOUS}$ SSYNC, or if there exists a general algorithm which allows every problem that can be solved in oblivious FSYNC to be solved in $\mathcal{LUMINOUS}$ SSYNC.

Chapter 3

Benchmark: Two-Robot Gathering

The **Gathering** problem is one of the benchmarking tasks in mobile robot networks, and has received a considerable amount of attention (*e.g.* [2, 3, 16, 21, 24, 32, 35, 43, 46, 53, 61, 73, 82, 83]). The **Gathering** task consists in all robots reaching a single point, not known beforehand, in finite time.

Definition 3.1.

Gathering *is achieved if and only if, for any pair of robots in the network, the distance between the two robots is eventually always zero.*

The **Rendezvous** problem is another name for the **Gathering** of two robots. Intuitively, the problem may seem simpler to solve due to the smaller number of robots, but this is actually the opposite, due to symmetry. Indeed, with only two robots and no lights, the lack of a common coordinate system implies all configurations are symmetrical and hence convey no information other than distance.

A foundational result [26, 83] shows that in the SSYNC model, no deterministic algorithm can solve **Rendezvous** without additional assumptions. This impossibility result naturally extends to the ASYNC model [46].

To circumvent the aforementioned impossibility results, it was proposed to endow each robot with a *light*. This additional capacity first allowed to solve **Gathering** of two robots in the most general ASYNC model provided that robots lights are capable to emit at least *four* colors [31]. This result was further improved by Viglietta [84] in 2013 who provided a three color ASYNC algorithm. In the same paper, Viglietta also proved that being able to emit two colors is sufficient to solve the **Rendezvous** problem in the more restricted SSYNC model, and that that no algorithm that only uses observed colors to decide its next move can gather two robots under the ASYNC scheduler using only two colors. Both solutions in ASYNC [31, 84] and SSYNC [84] output a correct behavior independently of the initial value of the lights' colors.

Recently, Okumura *et al.* [74] presented an algorithm with two colors that gathers robots in ASYNC assuming *rigid* moves (that is, the move of every robot is never stopped by the scheduler before completion), or assuming non-rigid moves but robots are aware of δ (the minimum distance before which the scheduler cannot interrupt their move). Also, the solution of Okumura *et al.* [74] requires lights to have a specific color in the initial configuration.

The remaining open case was the feasibility of **Rendezvous** with only two colors in the most general ASYNC model, without additional assumptions.

3.1 2-color Impossibility ?

Viglietta observes [84] that, in order to solve the **Rendezvous** problem in SSYNC, an algorithm must accomplish two things:

- In case robots are synchronized, they need to move towards the midpoint.
- In case robots are activated alternatively, one needs to move towards the other. In that case, the other robot must not move.

In ASYNC, Viglietta [84] also shows that no algorithm using only two colors can solve **Rendezvous** if the destination computation solely relies on this form of calculation:

$$me.destination = (1-\lambda) \cdot me.position + \lambda \cdot other.position$$

With:

$$\lambda = f(me.color, other.color)$$

Where f is a function (that is, it associates to a 2-tuple a single image).

It is similarly assumed that the next color of a robot only depends on the current colors of the two robots and not on the distance between the robots.

Algorithms that follow these rules of computation are called class $\mathcal{L}$ algorithms. Then, from Viglietta [84] algorithm 3.1 is the only algorithm of class $\mathcal{L}$ that satisfies the above criteria, and is presented in figure 3.1.

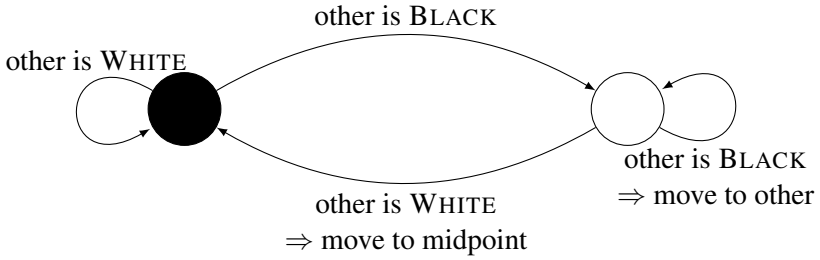


Figure 3.1 – Viglietta's [84] Algorithm

Algorithm 3.1 $\mathcal{L}$ Class Algorithm for Two Colors

```
if me.color = WHITE
    if other.color = WHITE
        me.destination ⇐ other.position/2
        me.color ⇐ BLACK
    else if other.color = BLACK
        me.destination ⇐ other.position
else if me.color = BLACK and other.color = BLACK
    me.color ⇐ WHITE
```

Now, there exists an execution of this algorithm that does not solve ASYNC **Rendezvous** (see lemma 4.9 in Viglietta's paper [84]) when both robots start in the BLACK color:

1. Let both robots perform a LOOK phase, so that both plan to turn WHITE and not move.

2. Let robot r_1 perform its COMPUTE phase and a new cycle with a LOOK and COMPUTE, while r_2 waits. Hence, r_1 remains WHITE and plans to move to r_2's position. Now, we let r_2 perform its COMPUTE phase and perform a new LOOK and COMPUTE. So, r_2 turns WHITE then BLACK and plans to move to the midpoint m between r_1 and r_2.

3. Let r_1 finish the current cycle, thus reaching r_2, and perform a whole new cycle, thus turning BLACK and moving to the midpoint m' between r_1 and r_2. Since r_1 reached r_2, r_1 moves towards its own position.

4. Finally, let r_2 finish the current cycle, thus turning BLACK and moving to m.

As a result, both robots are again set to BLACK, are in a WAIT phase, both have executed at least one cycle, and their distance has halved. Thus, by repeating the same pattern of moves, they approach one another but never gather.

Because of this execution, it is not possible to solve **Rendezvous** with two colors with an $\mathcal{L}$ class algorithm.

As a result, we do not design our algorithm to be of class $\mathcal{L}$, as our computation of the next color not only depends on the respective colors of the two robots, but also on multiplicity, that is whether or not robots share the same coordinates.

3.2 Our Algorithm: 2-color Rendezvous

We observe that in the problematic aforementioned execution, there is an instant when both robots are actually gathered, but are later separated because of pending moves.

We thus introduce a behavior change in the WHITE state of Viglietta's [84] algorithm to obtain our proposal, presented in figure 3.2 and algorithm 3.2.

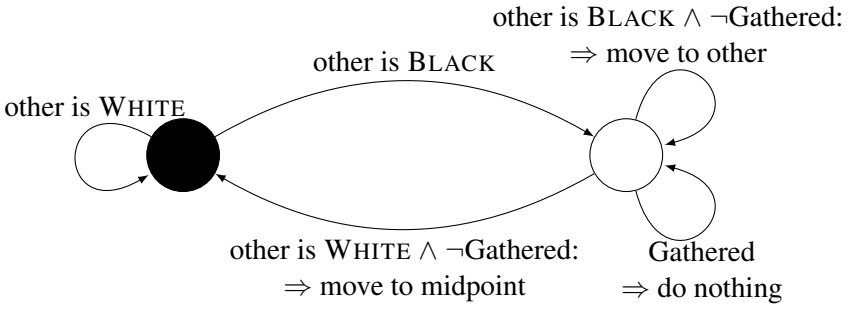


Figure 3.2 – Our ASYNC Robot **Rendezvous** with Two Colors

Our proposal breaks the infinite loop in the problematic execution, as it prevents robot r_1 from switching to color BLACK after reaching r_2 and forces it to remain WHITE. This implies that activating r_2 afterwards actually separates the robots into different colors, and prevents them from going back to both being Black.

Let us observe that our new algorithm no longer belongs to class $\mathcal{L}$, since the same observed 2-tuple of colors may yield different outcomes depending on the distance between r_1 and r_2. In particular, when both robots are observed WHITE, the next color depends on whether the two robots are gathered. So, the assumption of Viglietta [84] that a new color is solely determined

Algorithm 3.2 Our ASYNC Robot **Rendezvous** with Two Colors
Changes with Algorithm 3.1 Underlined

```
if me.color = WHITE
    if (me.position = other.position)
        do nothing
    else if other.color = WHITE
        me.destination ⇐ other.position/2
        me.color ⇐ BLACK
    else if other.color = BLACK
        me.destination ⇐ other.position
else if me.color = BLACK and other.color = BLACK
    me.color ⇐ WHITE
```

by the current colors no longer holds.[1].

We now need to prove that this new algorithm actually solves the **Rendezvous** problem in ASYNC in a self-stabilizing manner. Our main result can be stated as follows:

Theorem 3.1. *Algorithm 3.2 solves the **Gathering** problem for two robots in a self-stabilizing fashion for the non-rigid ASYNC model.*

[1]It is worth noting that, while the definition of class $\mathcal{L}$ does not explicitly mention that the new color is also obtained as a function of the two observed colors, the lemma 4.4 of Viglietta's paper [84] entirely relies on this implicit fact, and so does the 3-color algorithm for the ASYNC model.

Chapter 4

Model Checking Rendezvous Algorithms

Despite its simplicity, proving the validity of algorithm 3.2 is extremely tricky.

Because of the problematic execution described above, the scheduler can execute a finite, but arbitrarily large number of cycles where the distance between the two robots decreases down to zero and increases back again. Because of this, most common methods for proving **Convergence**, let alone **Gathering**, are unusable, and we have not been able to find an invariant to reliably prove this algorithm in an elegant way.
Our first approach [53] was to manually map out the entire possible state space for the network, then show that every execution in that space leads to a gathered configuration. However, this type of proof is extremely tedious and prone to errors.

As a matter of fact, recent reports of errors in mobile robot papers published in established venues such as DISC, Distributed Computing, and SIAM Journal of Computing have been presented [12, 22, 40]. It is remarkable that the set of authors of both series of papers are disjoint, demonstrating the general difficulty of reasoning about asynchronous mobile robotic systems by human beings.

Formal methods encompass a long-lasting path of research that is meant to overcome errors of human origin. Unsurprisingly, this mechanized approach to protocol correctness was used in the context of mobile robots [6, 9, 11, 12, 15, 26, 36, 70, 78, 79].

When robots move freely in a continuous two-dimensional Euclidean space, to the best of our knowledge the only formal framework available is Pactole[1].

It relies on higher-order logic to certify impossibility results [6, 9, 26], as well as the correctness of algorithms [27, 36] in the FSYNC and SSYNC models, possibly for an arbitrary number of robots (hence in a scalable manner). Pactole was recently extended by Balabonski *et al.* [7] to handle the ASYNC model, thanks to its modular design. However, in its current form, Pactole lacks automation; that is, in order to prove a result formally, one still has to write the proof (that is automatically verified), which requires expertise both in Coq (the language Pactole is based upon) and about the mathematical and logical arguments one should use to complete the proof.

On the other side, model checking and its derivatives (automatic program synthesis, parameterized model checking) hint at more automation once a suitable model has been defined with the input language of the model checker. In particular, model-checking proved useful to find bugs (usually in the ASYNC setting) [12, 39, 40] and to formally check the correctness of published algorithms [12, 36, 78]. Automatic program synthesis [15, 70] was used to obtain automatically algorithms that are "correct-by-design". However, those approaches are limited to instances with few robots. Generalizing them to an arbitrary number of robots with similar models is doubtful as Sangnier *et al.* [79] proved that safety and reachability problems are un-

[1] `http://pactole.lri.fr`

decidable in the parameterized case. Another limitation of the above approaches is that they *only* consider cases where mobile robots *evolve in a discrete space* (*i.e.*, graph). This limitation is due to the model used, that closely matches the original execution model by Suzuki and Yamashita [83]. As a computer can only model a finite set of locations, a continuous 2D Euclidean space cannot be expressed in this model.

Overall, the only way to obtain automated proofs of correctness in the continuous space context through model checking is to use a more abstract model.

We study the possibility of using model checking methods in the case of **Rendezvous** for mobile robots in a continuous Euclidean plane.

4.1 System Model

We consider three new synchrony models: Centralized (where LCM cycles execute in mutual exclusion) and the more recent LC-atomic ASYNC and Move-atomic ASYNC schedulers [74], which are variants of the ASYNC for which the LOOK and COMPUTE, or the entire MOVE phase, respectively, must happen atomically. All schedulers are assumed to be fair in the sense that they activate every robot infinitely often.

Additionally, we consider that a robot is in a WAIT phase after finishing its MOVE phase and before entering its LOOK phase.

Additionally, during the compute phase, the snapshot may lead to a deterministic change in color. Then, we say the new color is *pending* during the COMPUTE phase. Similarly, we call *target* (or pending move) the destination dictated by a robot's snapshot that it tries to reach in its next MOVE phase. This target is also *pending* during the COMPUTE phase, actual during the MOVE phase, and undefined otherwise.

This chapter considers both the full and external light models.

While the existing literature only considers atomic lights, we introduce lights with weaker consistency guarantees in section 4.5.

An execution of robot r is defined as a possibly infinite sequence of activation cycles of r.

4.1.1 Configurations and Executions

The union of the local states (position, color, phase, pending move and color) of all robots defines a *configuration*. An *execution* is a sequence, possibly infinite, starting in an initial configuration and where each transition corresponds to the activation of a robot according to the constraints of the scheduler (see figure 4.4).

4.1.2 Self-Stabilization

A **Rendezvous** algorithm is self-stabilizing if robots eventually reach and stay forever at the same location regardless of the *initial configuration*. Algorithms that set constraints on the initial configuration (*e.g.*, must start with a specific color) are not self-stabilizing.

We introduce a more refined definition of self-stabilization.

Definition 4.1 (*Simple* Self-Stabilization)**.**
An algorithm is simply self-stabilizing *for problem $\mathcal{P}$ if it solves $\mathcal{P}$ starting from any initial position and any initial color, with all robots in the* WAIT *phase.*

Definition 4.2 (*Complete* Self-Stabilization)**.**
An algorithm is completely self-stabilizing *for problem $\mathcal{P}$ if it solves $\mathcal{P}$ from any initial position, color, phase, target and pending color.*

Following the same terminology, all initial configurations are *complete*, and an initial configuration where both robots are in the WAIT phase is called a *simple* initial configuration.

Similarly all executions are *complete*, and if the initial configuration of an execution is a *simple* initial configuration, then it is a *simple* execution. If a *complete* execution has a common suffix with a *simple* execution, we say it is *simple-reachable*.

4.2 From the System Model to the Verification Model

Implementing the system model we consider into a *verification model* that can be checked by a model-checker is difficult, as some elements are continuous (position of both robots, pending moves of both robots). This section is dedicated to proving that those problematic elements can be discretized in a way that enables mechanized verification.

4.2.1 *Simple* vs. *Complete* Self-Stabilization

This subsection is dedicated to proving that pending moves and pending colors can be removed from the verification model in the case of self-stabilizing **Rendezvous** algorithms. This is true for all self-stabilizing algorithms under the FSYNC, Centralized, and SSYNC schedulers (lemma 4.2), and true for specific self-stabilizing algorithms under the ASYNC scheduler (theorem 4.2).

Lemma 4.1. *Any* completely *self-stabilizing algorithm is also* simply *self-stabilizing.*

Proof. Since the set of initial configurations allowed for *simple* self-stabilization is a subset of the one allowed for *complete* self-stabilization, if all *complete* initial configurations lead to a successful execution, then all *simple* starting executions also do. □

We now want to prove that every *complete* execution is *simple-reachable*. If this is the case, then any *complete* execution eventually has a common suffix with a *simple* execution, and we only need to verify *simple* initial configurations to verify eventual gathering starting from a *complete* initial configuration.

Lemma 4.2. *Under the FSYNC, Centralized and SSYNC schedulers, any* simply *self-stabilizing rendezvous algorithm is also* completely *self-stabilizing.*

Proof. Under the FSYNC, Centralized and SSYNC scheduler, any *complete* initial configuration becomes a *simple* initial configuration after all robots finish their current cycle. □

Intuitively, it seems that this also holds for the case of ASYNC algorithms. Since both robots are oblivious, with the exception of color, it seems logical that the system eventually "forgets" its initial configuration, and becomes reachable from a *simple* initial configuration.

Surprisingly, we show that it is possible, for a well-chosen algorithm, ASYNC scheduling, and *complete* initial configuration to create an infinite execution that never becomes *simple-reachable*.

Theorem 4.1. *There exist algorithms for which, under the ASYNC scheduler, there exist* complete *executions which are not* simple-reachable.

For a well-chosen algorithm, scheduler, and *complete* initial configuration, it is possible for the system to have an emerging property of memory. This is because it is possible to have the current configuration depend on the initial configuration indefinitely. We prove this theorem for both oblivious and $\mathcal{LUMINOUS}$ robots.

Proof: Oblivious Robots.

Let us assume two robots r_1 and r_2 running the ToOther algorithm, *i.e.*, the target is always the other robot. Let us also assume an initial configuration where r_1 is in the WAIT phase and r_2 is in the COMPUTE phase and has targeted point P_1 such that $|r_1P_1| = |r_2P_1| = |r_1r_2|$. In other words, $r_1r_2P_1$ is an equilateral triangle. Note that this is a *complete*, but not *simple* initial configuration.

We first activate r_1, which is now in its COMPUTE phase and targets the current location of r_2. We then activate r_2 which starts moving towards P_1. We then activate r_2 again as it reaches P_1 and is now in its WAIT phase.

This current configuration is identical to the initial configuration. Thus, we have an execution that repeats infinitely often while never being reachable from both robots starting in the WAIT phase, because starting from the WAIT phase cannot yield a target outside the $[r_1r_2]$ segment.

While this is the simplest example of the behavior that we could devise, it should be noted that a similar execution could be achieved using a move-to-half algorithm, the only difference being the $r_1r_2P_1$ would be shrinking with each activation. Similarly, any initial configuration that included a target outside of the (r_1r_2) line could lead to a similar execution, given the right algorithm and ASYNC scheduling.

Figure 4.1 visually shows the execution: the colored cross shows the target of the robot of the corresponding color.

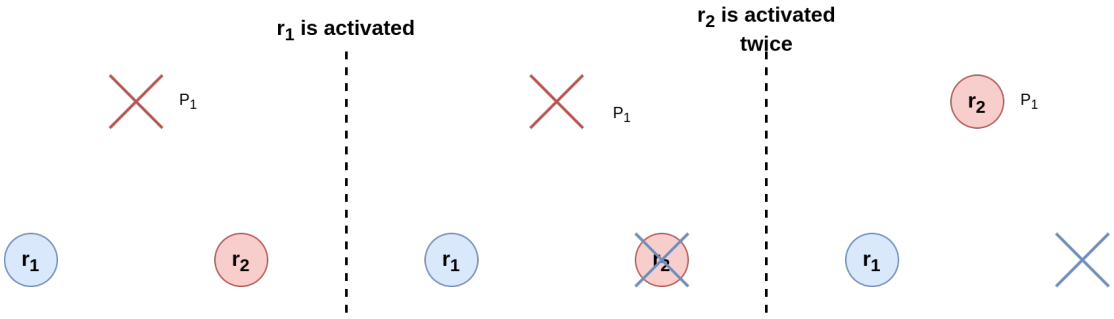


Figure 4.1 – Proof for Oblivious *Complete* Self-Stabilization

□

Proof: LUMINOUS Robots.

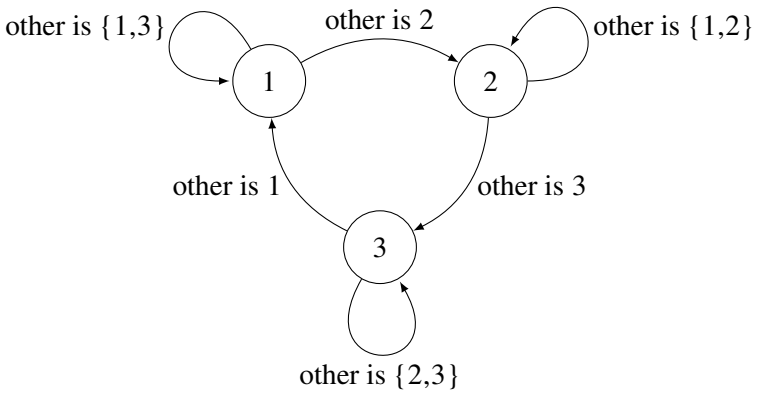


Figure 4.2 – Algorithm for the *LUMINOUS Complete* Self-Stabilization

Let us assume the three-color algorithm in figure 4.2, and a *complete* initial configuration of two robots r_1 and r_2, where r_1 starts in color 1, in the WAIT phase, and r_2 in color 2, in the COMPUTE phase, with 3 as a pending color.
We then follow the execution described in figure 4.3. We see that the last configuration is identical to the first one with r_1 and r_2 swapped, which means the execution can be repeated infinitely.

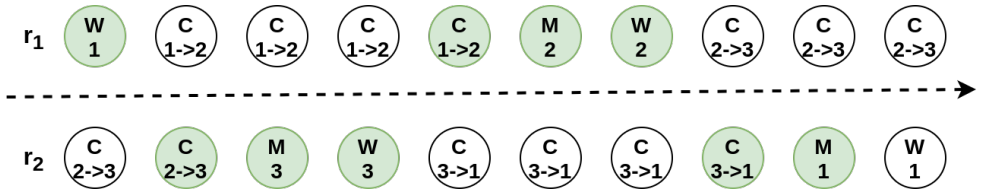


Figure 4.3 – Execution for the *LUMINOUS* Proof of Theorem 4.1
W indicates the WAIT phase, C COMPUTE, and M MOVE
1, 2 and 3 indicate the color and, if applicable, the arrow indicates a pending color
The activated robot is highlighted in green

Let us now prove that this execution cannot be reached from a WAIT/WAIT *simple* initial configuration.

A three-color algorithm allows for 6 different color combinations.

- Starting from {1,2}, robot r_2 is stuck in color 2 and robot r_1 turns to color 2. So {1,2} leads to {2,2}.
- Starting from {2,3}, robot r_2 is stuck in color 3 and robot r_1 turns to color 3. So {2,3} leads to {3,3}.
- Starting from {3,1}, robot r_2 is stuck in color 1 and robot r_1 turns to color 1. So {3,1} leads to {1,1}.
- Starting from {1,1}, no robot can change color.
- Starting from {2,2}, no robot can change color.
- Starting from {3,3}, no robot can change color.

We now see that, if starting from WAIT/WAIT, no cycle of changing colors can be reached. Therefore, the previously described *complete* execution cannot be reached either. □

In practice, most **Rendezvous** algorithms in the literature prevent this behavior by commanding robot r_1 to wait for robot r_2 without changing its color or moving, enforcing strong synchronization between r_1 and r_2. As the scheduler relies on its ability to feed outdated information infinitely often to the robot performing the LOOK to create executions that are not *simple-reachable*, when synchronization is enforced by the algorithm, the scheduler loses this capability.
Because no deterministic solution exists in the oblivious setting, we study the case of ASYNC $\mathcal{LUMINOUS}$ **Rendezvous** algorithms using at least two colors.

To tackle the case of color *'memory'*, we first consider two ASYNC $\mathcal{LUMINOUS}$ **Rendezvous** algorithms: Viglietta 3-color [84] and Heriban 2-color [53]. For both algorithms, we find a structural condition that we prove sufficient to prevent the scheduler from creating any execution where memory of the initial color emerges.

During the following proof, we consider that, whenever a robot has no pending color, its pending color is set to its visible color.

Definition 4.3 (Identical Color Condition)**.**
We define the identical color condition *(ICC) as: "For any pair of robots r_1 and r_2 whose colors are* C1 *and* C2*, respectively, r_1 can decide on a new color (different from* C1*) if and only if its snapshot shows* C1 *and* C2 *are identical."*

Theorem 4.2 (Identical Color Condition)**.** *For any $\mathcal{LUMINOUS}$ **Rendezvous** algorithm, if the algorithm satisfies ICC and is* simply *self-stabilizing, any* complete *execution with initial pending colors is reachable by a* complete *configuration where each robot has identical pending and visible color.*

Proof. Let us first assume a *complete* configuration where both robots r_1 and r_2 are in their COMPUTE phase, with each a pending color. We notice that after at most three activations, at least one robot (r_2) is in WAIT, while the other (r_1) is in WAIT, COMPUTE (with a pending color) or MOVE (with no pending color). We ignore the cases where r_1 is in WAIT or MOVE, as they trivially have identical pending and visible colors.

So, without loss of generality, we can only consider *complete* configurations where r_2 is in WAIT while r_1 is in COMPUTE, with a pending color.

Let us consider the case where one robot r_1 is in COMPUTE with different pending and visible colors A1 and A0, and robot r_2 is in WAIT with identical visible and pending colors, B0.
We first show that the first activated robot must be r_2, as we see in table 4.1. The left execution can be reached by the right execution with robots having the same arbitrary targets.

r_1		r_2	
Pending	Visible	Pending	Visible
A1	A0	B0	B0
A2(B0)	A1	B0	B0

r_1		r_2	
Pending	Visible	Pending	Visible
A1	A1	B0	B0
A2(B0)	A1	B0	B0

Table 4.1 – Activating Robot r_1 First (left) Leads to a Reachable Configuration (right)

We show an execution that is not reachable in table 4.2
We now note that, if the algorithm follows the identical color condition, in order for B1 to be different from B0, we require A0 to be identical to B0. If we do not force change in color and have B1 = B0, the configuration does not change and the first activation of r_1 leads to the counter example shown in table 4.1. Because of this, the scheduler now has two choices : Either activate r_1 or r_2.
These two executions are shown if tables 4.3 and 4.4.

r_1		r_2	
Pending	Visible	Pending	Visible
A1	A0	B0	B0
A1	A0	B1(A0)	B0
A2(B0)	A1	B1(A0)	B0
A2(B0)	A1	B2(A1)	B1(A0)
A3(B1(A0))	A2(B0)	B2(A1)	B1(A0)

Table 4.2 – A Possible Memory Execution
Note the continuing dependency on colors A0, B0, and A1

r_1		r_2	
Pending	Visible	Pending	Visible
A1	A0	B0=A0	B0
A1	A0	B1(A0)	A0
A2(A0)	A1	B1(A0)	A0

Table 4.3 – After Activating r_2 then r_1

r_1		r_2	
Pending	Visible	Pending	Visible
A1	A0	B0=A0	B0
A1	A0	B1(A0)	A0
A1	A0	B2(A0)	B1(A0)

Table 4.4 – After Activating r_2 Twice

In the case shown in table 4.3, following the condition leads to A1 being identical to A0, which is a contradiction.

In the case shown in table 4.4, for B2 to be different than B1, following the condition leads to B1 being identical to A0, so B2 is also identical to B1, a contradiction.

□

Definition 4.4 (Move and Stay Condition)**.**
We define the Move and Stay Condition *(MSC) as: "For any three colors* C1*,* C2 *and* C3*:*

- *if a robot in color* C1 *seeing the other robot with color* C2 *can switch to color* C3 *and perform a move to midpoint, then a robot in color* C3 *cannot move or change its color if it sees the other robot with color* C2*, and cannot move if the other robot is in color* C3*.*
- *if a robot in color* C1 *seeing the other robot with color* C2 *can switch to color* C3 *and perform a move to other, then a robot in color* C2 *seeing the other robot with color* C1 *cannot move or change its color.*

This condition holds for both Vig3 and Her2.

Theorem 4.3 (Move and Stay Condition)**.** *For any* $\mathcal{LUMINOUS}$ **Rendezvous** *algorithm A, if A satisfies both ICC and MSC, and is* simply *self-stabilizing, then A is also* completely *self-stabilizing.*

Proof. Let us look at the implication of ICC when applied to MSC.

We first note that C3 can only be different from C1 if C1 and C2 are identical.

In the first condition of MSC, C1 and C3 cannot be identical, as a robot in color C3 = C1 seeing a robot in color C2 would not be able to move, so, because of ICC, C1 and C2 must be identical.

In the second condition of MSC, if C1 and C2 are identical, then the robot in color C2 seeing the other in color C1 is able to move to other, so C1 and C2 must be different, and, because of ICC, C3 is identical to C1.

Because the algorithm satisfies ICC and by Theorem 4.2, the execution can be reached by a configuration in which each robot has an identical pending and visible color. So we consider this configuration, which may include pending targets.

Let us first consider a *complete* configuration where both robots r_1 and r_2 are in their COMPUTE phase, with each an arbitrary pending target, and no pending color. After at most three activations, at least one robot (r_1) is in WAIT, while the other (r_2) is in WAIT, COMPUTE (with its original arbitrary target), or MOVE (and moving towards its original arbitrary target). We ignore the case where r_2 is in WAIT, as it is trivially *simple-reachable*, since r_2 would have completed its move and hence deleted its pending target.

So, without loss of generality, let us now consider *complete* configurations where r_1 is in WAIT while r_2 is either in COMPUTE or MOVE, with an arbitrary target and no pending color.

Again, without loss of generality, let us only consider *complete* configuration where r_1 is in WAIT while r_2 is in MOVE, as r_2 carries no pending color. The colors of r_1 and r_2 at this stage are named C1 and C2, respectively.

We first note that the scheduler has to activate r_1 first, as otherwise r_2 reaches WAIT and the configuration becomes *simple-reachable*.

1. If r_1 decides not to move and not to change color, it does so indefinitely and the execution is *simple-reachable* after r_2 is activated and reaches the WAIT phase.

2. If r_1 decides not to move and to change color, then, from ICC, colors C1 and C2 are identical, and r_1 must be activated again, as activating r_2 leads both robots to carry no arbitrary targets. So the configuration is similar to the initial configuration.

3. If r_1 decides to move towards r_2, then, from MSC, it keeps color C1, and r_2 cannot move when activated next, so the execution is *simple-reachable*.

4. If r_1 decides to move to the midpoint, then, from ICC and MSC, colors C1 and C2 are identical, and r_1 prepares to switch to color C3, and r_2 must now be activated before r_1 finishes its cycle, as r_1 cannot move or switch color after switching to color C3.

   (a) If r_2 is activated while r_1 is in MOVE, ICC implies it cannot change its color, as C3 is different from C2. So, from MSC, it must perform a move to other. So, r_1 is not be able to move when activated next, so the execution is *simple-reachable*.

   (b) If r_2 is activated while r_1 is in COMPUTE, it also targets the midpoint and prepares to switch to color C3. The robot that finishes its cycle first is then stuck until the other starts moving and switches to color C3 because of MSC. Let us assume r_1 is moving while r_2 is in WAIT, both with color C3. The scheduler should obviously not activate r_1, so it activates r_2, which prepares to switch to color C4 without moving (MSC). If C4 is identical to C3, then r_2 is stuck and the execution is *simple-reachable* once r_1 completes its cycle. If r_1 is activated, it also prepares to switch to C4 without moving. Neither robots have movement targets, so this execution is *simple-reachable*.

   If r_2 is activated, then both are stuck with separate colors because of ICC. So, the only possible move is "to other" for one of the robots, while the other has to stay, and the execution is *simple-reachable*.

□

In this chapter, the model checker only uses *simple* initial configurations. From theorem 4.3, we can extend the positive results to *complete* initial configurations when considering algorithms that satisfy ICC and MSC such as Viglietta 3-color [84] and Heriban 2-color [53]. Of course, negative results are not impacted, as a counter-example in the simply self-stabilizing context is also a counter-example in the completely self-stabilizing context.

4.2.2 Self-Stabilization and Rigidity

The non-rigid assumption is another source of a continuous variable in the model: when the robot targets a point at some distance $d \geq \delta > 0$, the scheduler may stop the robot anywhere between δ and d. In this section, we explore under which circumstances we can restrict the verification model to rigid moves only without losing generality, and show that completely self-stabilizing **Rendezvous** algorithms satisfy the condition (theorem 4.4).

Using criteria such as (complete) self-stabilization and rigidity, we can define four different settings for **Rendezvous** algorithms according to the combination of $\{$*rigid*,*non-rigid}* and $\{$*self-stabilizing*, *non self-stabilizing}*. Studying the literature on **Rendezvous** algorithms, we were not able to find examples of self-stabilizing algorithms requiring rigid moves that failed with non-rigid moves. We now prove that, in fact, such algorithms cannot exist.

Lemma 4.3. *The three types of motion stay put (*STAY*), move to the midpoint (*M2H*), and move to the other robot (*M2O*) are both necessary and sufficient to achieve **Rendezvous** in SSYNC and ASYNC.*

Proof. First, these three motions are obviously sufficient since they are the only ones used by both Heriban *et al.* [53] and Viglietta [84] with two and three colors, respectively.

Next, we prove that it is necessary to use all of these motions to achieve **Rendezvous**.

Consider the case where both robots are at distinct positions, anonymous and have the same color. As proven by Viglietta [84] in proposition 4.1: "We may assume that both robots get isometric snapshots at each cycle, so they both turn the same colors, and compute destination points that are symmetric with respect to their midpoint. If they never compute the midpoint and their execution is rigid and fully synchronous, they never gather." Therefore, Move to Half is necessary.

Similarly, consider now a case where snapshots are not isometric because of different colors. Let us assume that their algorithm makes them target any point between them, but not their own positions.

Because of the case where they are both activated at the same time, their targets need to be identical to gather. We model this as $\frac{D}{x}$ for robot r_1 and $D \cdot \frac{(x-1)}{x}$ for robot r_2, with D the distance between r_1 and r_2 and x is a real positive number. This is mandatory in the case where r_1 and r_2 are activated at the same time.

However, if r_1 and r_2 are activated sequentially for a full cycle and $x \neq 1$, then r_1 and r_2 have different targets. As long as no motion where $x = 1$ exists, no **Rendezvous** can happen if r_1 and r_2 are separated. When $x = 1$, r_1 uses Move to Other and r_2 uses Stay and **Rendezvous** can be achieved. Therefore, M2O and Stay are both necessary. □

Theorem 4.4. *Any* completely *self-stabilizing algorithm that achieves **Rendezvous** assuming rigid moves also achieves it assuming non-rigid moves.*

Proof. Because lemma 4.3 shows STAY, M2H and M2O are necessary and sufficient, we now assume that all ASYNC **Rendezvous** algorithms use only these three types of motion.

- Stay (STAY)
- Move to the midpoint (M2H)
- Move to the other robot (M2O)

First, it is trivial to see that, using these three types of motion, if both robots start in a WAIT phase at a distance X, then the distance between r_1 and r_2 cannot become greater than X .
Next, we look at what happens after each robot completes at least one full cycle. We assume r_1

	r_1 has a pending STAY	r_1 has a pending M2H	r_1 has a pending M2O
r_2 executes k STAY	X	$\frac{X}{2}$	0
r_2 executes k M2H	$\frac{X}{2^k}$	$\{\frac{X}{2}, 0, \frac{X}{2^k}\}^\star$	$X - \frac{X}{2^k}$
r_2 executes k M2O	0	$\frac{X}{2}$	$\{0, X\}$

Table 4.5 – Distance after a full cycle of r_1 and k cycles of r_2 with an initial distance of $X \leq \delta$
$^\star \frac{X}{2}$ for $k = 0$ and 0 for $k = 1$

	r_1 has a pending STAY	r_1 has a pending M2H	r_1 has a pending M2O
r_2 executes k STAY	X	$\left[\frac{X}{2}, max(\frac{X}{2}, X-\delta)\right]$	$[0, X-\delta]$
r_2 executes k M2H	$\left[\frac{X}{2^k}, max(\delta^\star, X-k\cdot\delta)\right]$	$[0, max(\delta^\star, X-(k+1)\cdot\delta)]$	$\left[0, X-\frac{X}{2^k}\right]$
r_2 executes k M2O	$[0, max(0, X-k\cdot\delta)]$	$\left[0, \frac{X}{2}\right]$	$[0, X]$

Table 4.6 – Distance after a full cycle of r_1 and k cycles of r_2 with an initial distance of $X > \delta$
*This upper bound is not strict, as the distance may decrease to less than δ depending on X, δ and k

performs a LOOK, and r_2 performs k cycles before r_1 finishes its MOVE. The distance after r_1 finishes its cycle is presented in tables 4.6 and 4.5.
In the case where r_2 performs k cycles with different types of motion (that is, a sequence of sequence of cycles of different types), the resulting interval is a composition where X in sequence i is substituted by the upper bound of the interval from sequence $i-1$, and the lower bound of the second interval is the minimum of the lower bounds of both sequences.

For the purpose of contradiction, let us assume the existence of a *completely* self-stabilizing algorithm A_R that achieves **Rendezvous** with rigid moves, but not with non-rigid moves. This implies that there exists a non-rigid execution that does not lead to **Rendezvous**.

Since the algorithm works in the rigid case, it cannot allow any of the following executions:

- No robot moves.
- Robot r_1 has a pending M2O and robot r_2 reaches robot r_1, either by:
  - executing at least one M2O.
  - executing an arbitrarily large number of M2H.

Indeed, from table 4.5, any infinite sequence of these three executions allows the scheduler to ensure the distance between robots never decreases, so the algorithm cannot allow them to be repeated infinitely.

Since these infinite sequences are not part of the algorithm, they should also not happen when the algorithm is used with non-rigid motion. So, from table 4.6, any execution of the algorithm necessarily decreases the distance towards δ, with δ being the distance between r_1 and r_2 below which the behavior becomes rigid. Thus, A_R ensures that, for all non-rigid executions, the distance between r_1 and r_2 is eventually at most δ. When that is the case, the behavior of the algorithm is strictly the same as the rigid-motion behavior. Since there exists an execution that does not achieve **Rendezvous**, this means that there exists a state (possibly with pending colors and targets), which is part of the states of the rigid-motion behavior that does not achieve **Rendezvous**. Hence, a contradiction. □

Theorem 4.4 implies that in order to prove complete self-stabilization, it is only necessary to prove the property assuming rigid moves.

4.2.3 Proving Rendezvous Algorithms

The last mile to the verification model is to show that the remaining continuous variable of the current configuration (the distance between the two robots) can be abstracted into two states only.

We observe that, for any **Rendezvous** algorithm execution that solely uses the three required movements, and where robots r_1 and r_2 start in the WAIT phase, the entire execution happens on the line $(r_1 r_2)$.

Theorem 4.5 (Rigid motion model)**.** *When proving the correctness of a **Rendezvous** algorithm with rigid-motion that solely uses the three required movements, only using the two model states* gathered *and* not-gathered *is sufficient to properly represent the Euclidean plane.*

Proof. We know that a single line is sufficient to model the plane for the **Rendezvous** problem in the general case. Therefore, all positions of r_1 and r_2 where the $|r_1 r_2|$ distance is the same are identical.
Since robots have no common notion of length, the actual distance between the two robots, other than being gathered or not, is not useful for deciding to move or change their color.
We assume robots only use the three required movements (lemma 4.3). Because the motion is rigid (theorem 4.4), robots always reach their destinations. Therefore, if an execution leads to **Rendezvous**, changing the initial distance between the two robots does not change the outcome of the execution.
So, any initial distance is equivalent to a *not-gathered* state. □

Thanks to theorem 4.5, we now have a finite number of states to model the entire Euclidean plane in the case of rigid **Rendezvous**. Note that this holds for both self-stabilizing and non-self-stabilizing algorithms. In turn, this implies that we may use model checking to verify the validity of a **Rendezvous** algorithm in the particular case of rigid motion. Furthermore, we can verify (simple) self-stabilization by checking all possible pairs of colors, complete self-stabilization if the algorithm satisfies ICC (by theorem 4.2) and non-rigid completely self-stabilizing algorithms (by theorem 4.4).
The only remaining family of algorithms is non-rigid, non-self-stabilizing algorithms.

Theorem 4.6. *To prove non-rigid non-self-stabilizing algorithms to achieve **Rendezvous**, verifying rigid behavior is necessary but not sufficient to prove the correctness of the algorithm.*

Proof. Consider an algorithm that achieves **Rendezvous** with non-rigid moves when both robots start from color C, but fails to do so for any other initial color combination. To achieve **Rendezvous**, the algorithm must work for any initial distance between both robots, including a distance smaller than δ. Therefore, we know that checking the rigid behavior for color C is necessary to prove non-rigid behavior.
On the other hand, Vig2Cols [84] achieves **Rendezvous** with rigid moves when both robots start from color BLACK. However, we also know this algorithm fails with non-rigid moves when starting with the same initial colors. Therefore, solely checking the rigid behavior would lead us to incorrectly consider Vig2Cols [84] to be a working non-rigid non-self-stabilizing algorithm. Hence, the condition is not sufficient. □

We present a possible approach for checking these algorithms in section 4.3.6. To the best of our knowledge, the 4-color external light algorithm by Okumura *et al.* [73] is the only known **Rendezvous** algorithm that satisfies these two criteria.

Note that our reasoning is only true in the case where the behavior of the algorithm is the same for any distance between r_1 and r_2 that is greater than zero. Recently, Okumura *et al.* [75] introduced an algorithm under the additional assumption that robots have the knowledge of δ. Because of this, the behavior of the algorithm is different when the distance is less than δ, and when it is between δ and 2δ, which means that the rigid and the non-rigid behavior of the algorithm are different. To prove non-rigid, non-self-stabilizing algorithms, we need to both prove the rigid and non-rigid behaviors.

Theorem 4.7. *To prove completely self-stabilizing* **Rendezvous** *algorithms whose behavior differs depending whether the distance between r_1 and r_2 is smaller than δ, between δ and 2δ, or greater than 2δ, it is sufficient to consider the case where robots are initially 3δ apart.*

Proof. In this particular case, we need to check three things:

1. That the rigid algorithm achieves **Rendezvous**.
2. That the farthest non-rigid algorithm leads to the closest.
3. That the closest non-rigid algorithm leads to the rigid.

The first point is proven in the rigid part of the chapter. We now only need to prove the second and third points.
This is easily done by remembering from the proof of theorem 4.4 that any algorithm using only the three moves either allows static executions or not. If it does, then the distance is never reduced, and checking at 3δ is equivalent to any other distance. If it does not, then the distance is eventually reduced to zero and hence eventually enters the second behavior.
Similarly, we check that the second behavior leads to rigid by reducing distance. We only need to check that this decrease happens once for every initial configuration to ensure that distance is reduced towards rigid behavior. □

4.3 Verification Model

4.3.1 Position

Our verification model only needs to consider two different positions (called NEAR, SAME) depending on the distance between the two robots. This choice is justified by the definition of the model (two robots, no shared coordinate system, no landmarks, oblivious robots) and theorem 4.5.

4.3.2 Activation and Synchrony

Let us now define the model we use in the model checking framework.

We consider the activation cycle of a robot r to be a sequence of four consecutive atomic events: LOOK, COMPUTE, MOVE_B, and MOVE_E. Each of the four events is as follows:

LOOK (L) The robot obtains a snapshot observation of the environment which consists of the color of both robots and the location of the other robot with respect to r's local coordinate system where r is always at the origin.

COMPUTE (C) The robot executes the algorithm which is defined as a function of the latest observation that returns a new color and a movement target. In the verification model, we assume that the light of the robot changes as part of the compute event.

MOVE$_B$ (B) The robot begins moving according to the movement target. Although it can be observed while moving, the actual position of the robot is actually undefined until the movement is completed with the MOVE$_E$ event.

MOVE$_E$ (E) The robot ends its move and has moved a distance of at least δ towards the target. If the distance to the target was equal or less than δ, the robot has reached its target.

We use this four-events verification model instead of the classical three phases model because of the flexibility it allows when defining variants of the ASYNC model. We can set the LOOK phase to be instantaneous by linking a COMPUTE event to happen right after each LOOK event, or set to LC-atomic ASYNC by linking LOOK, COMPUTE and MOVE$_B$, or Move-atomic ASYNC by linking MOVE$_B$ and MOVE$_E$, and so on.

A global execution is a sequence of events on both robots, such that the event of each robot r is a robot execution of r, and the interleaving of events follows the rules of the activation model:

Centralized The activation cycle of a robot is atomic. In other words, a single robot is activated at a time and executes a full activation cycle each time (see figure 4.4b).

FSYNC The activation cycles of both robots are executed simultaneously. Equivalently, the robots always follows the following atomic sequence: each robot executes a LOOK event and then, in turn, each robot sequentially execute COMPUTE, MOVE$_B$, and MOVE$_E$. This is depicted in figure 4.4c.

SSYNC Activation cycles can be either centralized or combined (as FSYNC).

ASYNC Each event is atomic but there is no atomicity between events (see figure 4.4a).

LC-atomic ASYNC Same as ASYNC, but the LOOK and COMPUTE events execute atomically.

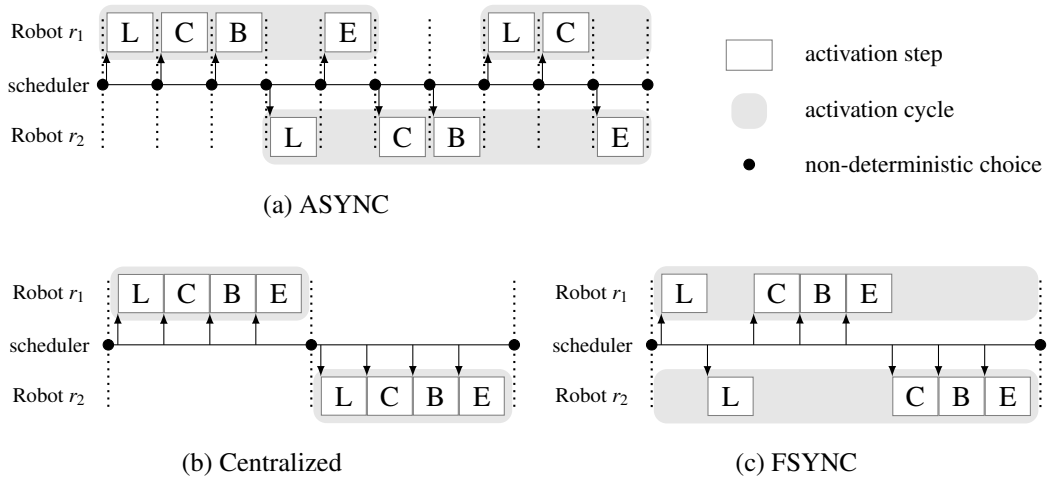
Move-atomic ASYNC Same as ASYNC, but the MOVE$_B$ and MOVE$_E$ events execute atomically.

Again, we assume that robots are always activable and that the scheduling is fair. Consequently, both robots are activated infinitely many times.

Theorem 4.8. *The FSYNC scheduler can be properly simulated by activating sequentially the* LOOK *phase of robot r_1, the* LOOK *phase of robot r_2, the* COMPUTE*,* MOVE$_B$*, and* MOVE$_E$ *of robot r_1 and finally the* COMPUTE*,* MOVE$_B$*, and* MOVE$_E$ *of robot r_2, infinitely often.*

Proof. In the FSYNC model, the LOOK, COMPUTE, MOVE$_B$, and MOVE$_E$ phases of all robots are executed simultaneously. However, since LOOK is a read-only operation, and COMPUTE a write-only operation, with regards to color, these operations can be executed sequentially as long as no LOOK happens after a COMPUTE (*i.e.*, all *read* operations happen before the first *write* operation). A similar reasoning holds for the MOVE$_B$ to MOVE$_E$ being the beginning and end of a continuous write operation. □

Theorem 4.9. *Two simultaneous events $E1$ and $E2$ can be properly simulated by exploring both sequences $(E1, E2)$ and $(E2, E1)$.*



(a) ASYNC

(b) Centralized

(c) FSYNC

Figure 4.4 – Simulation of Main Schedulers as a Promela Process.

Proof. We consider two cases:

1. In the first case, one event ($E1$) is a *read* operation (LOOK), and one event ($E2$) is a *write* operation (COMPUTE, MOVE_B or MOVE_E). Since information is being read and written at the same time, the result of the read operation cannot be determined. In the case of mobile robots, where the *write* operation is a color transition from color C1 to color C2, we assume for now that the only colors that can be seen by the LOOK are C1 or C2[2]. We then need to consider the case where the LOOK saw a C1 ($E1$ then $E2$) and the case where it saw a C2 ($E2$ then $E1$).
   Similarly, if the *write* operation is either beginning of ending the MOVE phase, then the reading robot can either see the writing robot as moving or not moving, so the *read* event either happens before or after the *write* event.

2. In any other case, since no read and write operation are happening at the same time, activating $E1$ and $E2$ simultaneously is identical to activating either $E1$ then $E2$, or $E2$ then $E1$.

□

Theorem 4.10. *Except for non-rigid non-self-stabilizing* ***Rendezvous****, a fair scheduler can be properly simulated by an 8N-bounded scheduler, where N denotes the number of colors available to the algorithm.*

Proof. To properly limit the number of activations of the scheduler, we need to ensure that any change in the configuration made by r_1 that may impact the snapshot of r_2 has been explored. When performing its cycle, there are two such elements: r_1's color, and the distance between r_1 and r_2.
Since there are only two distance states {SAME, NEAR }, a fixed number N of colors, and it takes 4 activations to perform a cycle, limiting the fair scheduler to an $8{\cdot}N$-fair scheduler, that is, a fair scheduler that can perform at most $8{\cdot}N$ activations of robot r_1 between two activations of robot r_2 still ensures that every possible snapshot has been explored. □

Because we check for a maximum of 5 colors, the model checker uses a 40-bounded scheduler.

[2]This assumption is analogous to assuming regularity with registers.

4.3.3 Movement Resolution

The key to our verification model is the idea of movement resolution. When a robot completes its movement, this translates into a change of the verification model according to specific rules, which are described below.

Stationary moves

When the computed move is invariant or stationary (e.g., M2O when the observed position is SAME), its pending move is systematically translated to an equivalent STAY move. A robot that has a STAY pending move is stationary, and hence is not observed as moving between MOVE_B and MOVE_E.

From NEAR or SAME position

The key aspect of the verification model is the case when the robots are in rigid motion (NEAR and SAME), and we only detail the resolution of moves for this combined case.

STAY No change.

MISS The pending move is a sure miss. A miss happens for instance if r observes the other robot while it moves. It also happens indirectly during movement resolution. The result of a MISS move is always NEAR (in particular, it can happen if the position was SAME).

M2O If the position is SAME, then the move is treated as a STAY. If the other robot is either STAY or $\perp$, then the position is now SAME. Else, if the other robot has a pending move, it is converted to a MISS and the position is now SAME.

M2H If the pending move of the other robot is STAY or $\perp$, then neither the distance nor the pending move of the other robot changes.
If the pending move of the other robot is also M2H, then the move is potentially successful and the pending move of the other robot is changed into an M2O move that targets the location just newly reached. Thus, provided that the first robot does not move in the meantime, the movement of the other robot later leads to SAME.
Else, the distance is still NEAR and the pending move or the other robot is now MISS.

from NEAR or SAME	
(* , STAY, *)	$\rightsquigarrow$ (– , $\perp$, –)
(* , MISS, STAY | $\perp$)	$\rightsquigarrow$ (NEAR, $\perp$, –)
(* , MISS, *)	$\rightsquigarrow$ (NEAR, $\perp$, **MISS**)
(* , M2O, STAY | $\perp$)	$\rightsquigarrow$ (**SAME**, $\perp$, –)
(SAME, M2O, *)	$\rightsquigarrow$ (SAME, $\perp$, –)
(NEAR, M2O, *)	$\rightsquigarrow$ (**SAME**, $\perp$, **MISS**)
(* , M2H, M2H)	$\rightsquigarrow$ (– , $\perp$, **M2O**)
(* , M2H, STAY | $\perp$)	$\rightsquigarrow$ (– , $\perp$, –)
(* , M2H, *)	$\rightsquigarrow$ (– , $\perp$, **MISS**)

Table 4.7 – Movement resolution.
Each tuple represents (position, me.pending, other.pending).
Major changes appear in boldface
A bottom value ($\perp$) represents the absence of pending moves.
Wildcard (*) replaces any value and placeholder (–) retains the original value.
Rule precedence is from top to bottom.

4.3.4 State Variables

The state of the system is represented by the following *explicit* variables:

- `distance` $\in \{\text{NEAR}, \text{SAME}\}$
  The distance between the two robots. NEAR means that it is equal or smaller and they have distinct positions, and SAME that they share the same location. When checking non-rigid non-self-stabilizing algorithm, the distance FAR can also be used.

- `robot[_].color` $\in \{\text{BLACK}, \text{WHITE}, \text{RED}, \text{YELLOW}, \text{GREEN}\}$
  The observable color of the robot. The color RED is used only in 3 colors, 4 colors, and 5 colors algorithms. The color YELLOW is used only in 4 colors and 5 colors algorithms. The color GREEN is used only in 5 colors algorithms.

and the following *implicit* variables:

- `robot[_].phase` $\in \{\text{LOOK}, \text{COMPUTE}, \text{MOVE}_\text{B}, \text{MOVE}_\text{E}\}$
  Keeps track of the next event to execute in the activation cycle of the robot. This variable is managed by the scheduler and is particularly important for the ASYNC scheduler.

- `robot[_].pending_move` $\in \{\text{STAY}, \text{M2O}, \text{M2H}, \text{MISS}, \perp\}$
  Holds the movement computed by the robot. This is not observable but used to resolve movements during the MOVE_E phases. The variable is updated during the LOOK phase, based on the movement computed by the algorithm.

- `robot[_].pending_color` $\in \{\text{BLACK}, \text{WHITE}, \text{RED}, \text{YELLOW}, \text{GREEN}, \perp\}$
  New color for the robot, as computed by the algorithm. The color is computed during the LOOK phase of the robot and used during the COMPUTE phase to update the visible color of the robot.

Depending on the model and algorithms, this can lead to at most $2 \cdot (5 \cdot 4 \cdot 5 \cdot 6)^2 = 720'000$ different states.

4.3.5 Activation Phases

LOOK Reads the current state of the environment and saves it as an observation. Applies the algorithm to compute the pending move and the new color.

COMPUTE Updates the color of the robot in the environment.

MOVE_B Begins moving. Unless the pending move is STAY, the robot's position is undetermined until MOVE_E and any that occurs in the interval causes an automatic MISS move for the other robot.

MOVE_E The pending move is resolved into a new position for the robot and the environment is updated accordingly.

4.3.6 The Case of Non-Rigid, Non-Self-Stabilizing Algorithms

Because of theorem 4.6, we cannot directly use our model checker in its current form to check this type of algorithms.

This is because a non-rigid algorithm can reach its rigid behavior in an unpredictable configuration. Our method for solving this issue is to notice that the number of those configurations is finite. More precisely, parameters include:

- The current color of each robot.
- The pending color of each robot.
- The phase of each robot.
- The pending move of each robot.

This means we have a number N_{conf} of possible configurations.

We use this fact by creating a counter, which starts at 0 and increases up to N_{conf}.

We start the validation process in the FAR state. When executing the first movement resolution that leads to NEAR, we create two branches:

1. In the first branch, the robot reaches NEAR and we continue the process.
2. In the second branch, the robot actually did not reach NEAR and is kept at FAR. We increment the counter by 1.

We repeat this process until the counter reaches $N_{conf} - 1$. We have then created $2^{N_{conf}} - 1$ branches to verify. However, we have ensured that any possible rigid motion behavior configurations has been checked. Repeating this process for every possible non-rigid motion initial configuration is enough to ensure that any possibly failing execution would have been detected.

4.4 Checking Rendezvous Algorithms

4.4.1 Verified Algorithms

To assess the verification model, we have checked three trivial baseline algorithms as well as seven known algorithms from the literature. For each of these algorithms, it is widely-known in which models they achieve Rendezvous or fail. Unless explicitly stated otherwise, algorithms are non-rigid and self-stabilizing. The latter seven algorithms are detailed in figure 4.5:

1. "NoMove": the robots never move and the algorithm never achieves Rendezvous regardless of the model.
2. "ToHalf": the robots always target the midpoint and the algorithm fails in all models but FSYNC.
3. "ToOther": the robots always target the other robot's position and the algorithm fails in all models but Centralized.
4. "Vig2Cols": the algorithm (figure 4.5b) was originally proved correct in SSYNC [84] but was later proved to also achieve Rendezvous in LC-atomic ASYNC [74]. The algorithm is known to fail in ASYNC [84].
5. "Vig3Cols": the algorithm (figure 4.5a) is known to succeed in ASYNC and consequently in all other weaker models [84].
6. "Her2Cols": the algorithm (figure 4.5c) is an extension of "Vig2Cols" that uses only two colors but succeeds in ASYNC [53]. The algorithm is optimal in the sense that Rendezvous in ASYNC cannot possibly be achieved with fewer colors.
7. "Flo3ColsX": the algorithm (figure 4.5d) achieves Rendezvous in SSYNC with external colors. The algorithm is known to succeed in SSYNC and to fail in ASYNC [48].
8. "Oku5colsX": the algorithm achieves Rendezvous in LC-atomic ASYNC in a model with external colors. The algorithm is known to succeed in LC-atomic ASYNC and to fail in ASYNC [73].
9. "Oku4colsX": the algorithm achieves Rendezvous in LC-atomic ASYNC in a model with external colors. It is quasi-self-stabilizing, meaning it requires the starting colors of the robots to be identical. The algorithm is known to succeed in LC-atomic ASYNC and to fail in ASYNC [73].
10. "Oku3colsX": the algorithm achieves Rendezvous in LC-atomic ASYNC in a model with external colors. It is a rigid, non-self-stabilizing algorithm. The algorithm is known to succeed in LC-atomic ASYNC and to fail in ASYNC [73].

(BLACK, BLACK)	⇝	WHITE, M2H
(BLACK, WHITE)	⇝	–, M2O
(BLACK, RED)	⇝	skip
(WHITE, BLACK)	⇝	skip
(WHITE, WHITE)	⇝	RED, STAY
(WHITE, RED)	⇝	–, M2O
(RED, BLACK)	⇝	–, M2O
(RED, WHITE)	⇝	skip
(RED, RED)	⇝	BLACK, STAY

(a) Vig3Cols
3 colors algorithm for ASYNC [84]

(BLACK, BLACK)	⇝	WHITE, STAY
(BLACK, WHITE)	⇝	skip
(WHITE, BLACK)	⇝	–, M2O
(WHITE, WHITE)	⇝	BLACK, M2H

(b) Vig2Cols
2 colors algorithm for LC-atomic ASYNC [84]

(BLACK, BLACK)	⇝	WHITE, STAY
(BLACK, WHITE)	⇝	skip
gathered	⇝	skip
(WHITE, BLACK)	⇝	–, M2O
(WHITE, WHITE)	⇝	BLACK, M2H

(c) Her2Cols
2 colors algorithm for ASYNC [53]

(*, BLACK)	⇝	WHITE, M2H
(*, WHITE)	⇝	RED, STAY
(*, RED)	⇝	BLACK, M2O

(d) Flo3ColsX
3 colors external for SSYNC [48]

(*, BLACK)	⇝	WHITE, M2H
(*, WHITE)	⇝	RED, STAY
(*, RED)	⇝	YELLOW, M2O
(*, YELLOW)	⇝	GREEN, STAY
(*, GREEN)	⇝	BLACK, STAY

(e) Oku5ColsX
5 colors external for LC-atomic ASYNC [73]

(*, BLACK)	⇝	WHITE, M2H
(*, WHITE)	⇝	RED, STAY
(*, RED)	⇝	YELLOW, M2O
(*, YELLOW)	⇝	BLACK, STAY

(f) Oku4ColsX
4 colors external Quasi-Self-Stabilizing for
LC-atomic ASYNC [73]

(*, BLACK)	⇝	WHITE, M2H
(*, WHITE)	⇝	RED, STAY
(*, RED)	⇝	WHITE, M2O

(g) Oku3ColsX
3 colors external rigid Non-Self-Stabilizing for
LC-atomic ASYNC [73]

Figure 4.5 – **Rendezvous** Algorithms from the Literature
Guards match the (me.color, other.color). Wildcard (*) replaces any value.
Placeholder (–) retains the original value, "skip" means no change, "gathered" holds only when both robots have the same position. Rule precedence is from top to bottom.

4.4.2 Verification by Model Checking

Given a **Rendezvous** algorithm and a verification model, the SPIN model checker essentially verifies that the following liveness property (expressed in LTL) holds in every possible execution:

```
ltl gathering { <> [] (position == SAME) }
```

The formula defines a predicate called `gathering` with the meaning that there is a time after which the position is always SAME. Concretely, to verify the property, the model checker runs an exhaustive search in the transition graph of configurations such that all initial configurations lead to some cycle such that the predicate `gathering` holds or, in other words, that the variable `position` is equal to SAME in every configuration of such cycle(s). The results are detailed in table 4.8

Centralized	FSYNC	SSYNC	LC-atomic ASYNC	Move-atomic ASYNC	ASYNC	
Non-Rigid Self-Stabilizing						
-	-	-	-	-	-	NoMove
-	✓	-	-	-	-	ToHalf
✓	-	-	-	-	-	ToOther
✓	✓	✓	✓	-	-	Vig2Cols
✓	✓	✓	✓	✓	✓	Vig3Cols
✓	✓	✓	✓	✓	✓	Her2Cols
✓	✓	✓	-	-	-	Flo3ColsX
✓	✓	✓	?⋆	-	-	Oku5ColsX
✓	-	-	-	-	-	Oku4ColsX
✓	-	-	-	-	-	Oku3ColsX
Rigid[a] Quasi-Self-Stabilizing						
✓	✓	✓	✓	-	-	Oku4ColsX[a]
Rigid Non-Self-Stabilizing						
✓	✓	✓	✓	-	-	Oku3ColsX

[a] *Oku4ColsX is supposed to be non-rigid. However, proving algorithms that are both non-rigid and non-self-stabilizing cannot be done by the current version of our model checker.*

⋆ Our model checker could not find counter examples for Oku5colsX. However, because it does not follow the Identical Color Condition, this does not prove validity for non-rigid motion

Table 4.8 – Results of Model-Checking Liveness

Two of those results were actually unexpected: Oku3colsX [73] and Oku4colsX [73] are not supposed to be self-stabilizing at all[3], yet are verified to be self-stabilizing under the centralized scheduler by our model checker. Looking in details at the algorithms, it turns out that the key counter-examples to self-stabilization rely on a simultaneous execution of both robots, which explains the result.

This concludes the proof of theorem 3.1.
We include more details of the model checker in chapter 12.3.5 of the appendix.

[3] However, Oku4colsX is quasi-self-stabilizing, that is, both robots always start with the same initial color chosen arbitrarily.

4.4.3 Performance

Using SPIN as a basis for our work, we were able to confirm known results for ten different **Rendezvous** algorithms proposed in the literature (performance results are presented in table 4.9).

	States				Runtime	Memory
	Sorted	Matched	Transitions	Atomic steps	[ms]	[MB]
NoMove	12,080	2,738	17,604	73,027	220	145
ToHalf	6,141	979	10,979	57,152	130	135
ToOther	4,046	57	8,367	34,372	110	134
Vig2Cols	188,010	4,014,448	5,452,656	33,925,728	3,110	151
Vig3Cols	612,209	13,678,976	18,419,016	114×10^6	11,200	190
Her2Cols	395,150	8,589,648	11,652,481	72,971,392	6,840	170
Flo3ColsX	13,053	48,509	80,419	440,286	210	135
Oku5ColsX	414,247	8,981,645	12,126,155	73,027,637	7,870	172
Oku4ColsX	307,795	6,607,778	8,936,310	54,718,251	5,080	162
Oku3ColsX	83,072	1,714,653	2,329,400	14,330,584	1,380	142
Oku4Cols QSS	307,793	6,607,778	8,936,308	54,718,251	5,110	162
Oku3Cols NSS	83,070	1,714,653	2,329,398	14,330,584	1,380	142

Table 4.9 – Model Checker Runtime Performance for the ASYNC Scheduler
(Intel i7-8650U running SPIN 6.4.9 on Arch Linux)

4.5 Investigating Lights with Weaker Consistency Guarantees

While previous sections were motivated for the purpose of verifying whether previously published results were indeed correct, this section is devoted to demonstrating how our tool can be used to explore problem solvability in new models not considered before, in particular, models that have strictly weaker consistency guarantees.

In particular, we concentrate on the current model considered for robots with lights, where lights are *atomically* modified. Similarly to Lamport's safe and regular versions of registers [64], we define safe and regular version of lights, in addition to the current atomic version used in the literature:

1. A light is *atomic* if its change of value can always be reduced to a single point in time in every execution.

2. A light is *regular* if a LOOK occurring during the change of its color can return either the color before the change or the color after the change. In particular, a *new-old* inversion phenomenon may occur: if robot r_1 changes its color from White to Black, then, robot r_2 may LOOK and see color Black, and later robot r_3 may look and see color White (including the case where $r_3 = r_2$.

3. A light is *safe* if a LOOK occurring during the change of its color can return any color in the possible subset of the algorithm.

Obviously, regular and safe lights provide weaker consistency guarantees, yet could result from actual physical phenomenon occurring in practice, such as light flickering (for regular lights) or lights whose color changes in a continuous palette (for safe lights). It is thus important to check algorithms against those weaker consistency guarantees.

We implemented all three lights models in our asynchronous model (obviously, in the semi-synchronous and fully synchronous models, these weaker models are irrelevant as a look never occurs during a color update). Alas, it turns out that none of the published algorithms we considered so far can handle even regular lights: indeed, additional cycles of non-gathered configurations appear for all algorithms, preventing **Rendezvous** from ever occurring.

Then, we tried to design a **Rendezvous** algorithm that could work with weaker consistency requirements with respect to the light category used. The underlying principle is to add supplementary colors between the colors used by previous algorithms (seen as important colors), in order to somewhat increase the chances that those important colors remain in the same sequence. Starting from Viglietta's 3-color algorithm [84], we derived the protocol presented in listing 4.1 using four colors.

Listing 4.1 – A New Algorithm for **Rendezvous** with Weaker Consistency Lights.

```
inline Alg_WeakConsistency(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.me == COL_A_P) -> command.new_color = COL_B
    :: (obs.color.me == COL_A)   ->
        if
        :: (obs.color.other == COL_A)
            -> command.new_color = COL_A_P;
                command.move = TO_HALF;
        :: (obs.color.other == COL_B)
            -> command.move = TO_OTHER
        :: else -> skip
        fi
    :: (obs.color.me == COL_B)   ->
        if
        :: (obs.color.other == COL_B) ->
            command.new_color = COL_C
        :: (obs.color.other == COL_C) -> command.move = TO_OTHER
        :: else -> skip
        fi
    :: (obs.color.me == COL_C)   ->
        if
        :: (obs.color.other == COL_C) ->
            command.new_color = COL_A
        :: (obs.color.other == COL_A) -> command.move = TO_OTHER
        :: else -> skip
        fi
    :: else -> command.new_color = COL_A
    fi
}
```

Note that this protocol does not follow ICC, so our verification only checks rigid, simple-self-stabilization. It turns out that our new protocol, unlike any published algorithm so far, is a valid ASYNC **Rendezvous** protocol for the regular light model. However, it still fails with the safe light model. As the code provided in listing 4.1 indicates, trying new protocols against the model-checker is not difficult as the code is self-explanatory, yet our tool gives an answer in a matter of seconds. Because our implementation is conservative, if the answer is positive, then, the protocol is valid for the considered model. However, a counter-example may or may not be valid and has to be analyzed further.

Chapter 5

Safe and Unbiased Leader Election with Lights

Leader Election is a fundamental problem of distributed computing in general. For mobile robots in particular, **Leader Election** is paramount as it is a requirement for other problems such as arbitrary pattern formation [47] and flocking [18, 49]. Surprisingly it appears no attempt had yet been made to build more powerful **Leader Election** algorithms using lights.

Deterministic **Leader Election** is known to be generally impossible for $\mathcal{OBLOT}$ mobile robots [47]. This is due to possible initial configuration that contain symmetries such that no robot can be singled out. A trivial example is a network of two robots, which is always symmetrical. The exact conditions under which **Leader Election** is deterministically possible have been detailed by Dieudonné *et al.* [37].

To solve this issue, robots are endowed with the ability to make decisions non-deterministically [19]. More specifically, they are able to perform Bernoulli trials of which the scheduler cannot predict or control the outcome. Current state-of-the-art algorithms by Canepa and Gradinariu Potop-Butucaru [19] allow for non-deterministic **Leader Election** for three or more robots using motion under either the SSYNC scheduler or the k-bounded ASYNC scheduler. For three robots, the algorithm relies on angles between robots and uses a Bernoulli trial to break the symmetries in the case where robots form an equilateral triangle. For more than three robots, the **LEADER** is the closest robot to the center of the smallest enclosing circle (SEC) of the network. If two or more robots are closest, a Bernoulli trial is used to reduce the number of candidates by allowing a random number of robots to move towards the center of the SEC.

In this chapter, we present the first **Leader Election** algorithms for $\mathcal{LUMINOUS}$ robots without using motion and positions. Because a network of two $\mathcal{LUMINOUS}$ robot *can* be asymmetrical, as opposed to networks of two $\mathcal{OBLOT}$ robots, which are always symmetrical, the proposed algorithms achieve **Leader Election** for a network of any size. We introduce a 2-color algorithm for the SSYNC scheduler, a 3-color for ASYNC. We also introduce the *safe* and *unbiased* properties for **Leader Election**. Safe **Leader Election** must prevent multiple **LEADER** robots from existing at any point during the execution. Unbiased **Leader Election** ensures all robots have the same chance of getting elected, regardless of scheduling.

We present a 3-color safe SSYNC, a 4-color safe ASYNC, a 4-color unbiased ASYNC and a 5-color safe, unbiased ASYNC **Leader Election** algorithm.

We also propose incremental upgrades to the state-of-the-art algorithms [19], which allow them to solve **Leader Election** in unbounded ASYNC and require less restrictive initial conditions.

5.1 Details of the Model

In this chapter, we consider $\mathcal{LUMINOUS}$ robots as defined in chapter I. Robots are also endowed with the ability to perform Bernoulli trials. At the start of its COMPUTE phase, a robot can choose between two different actions, such as target locations, or colors, with a winning option being given a certain probability, and the losing action the complementary probability.

5.2 Problem Definition

Definition 5.1 (Leader Election)**.**

An algorithm achieves ***Leader Election*** *if there exists a* **LEADER** *state (color, or position for instance), and, for any possible execution, there exists a suffix for which there exists a single robot r_1 such that r_1 is always the only robot in the* **LEADER** *state.*

Looking at this definition, it appears that a valid execution may include configurations that contain multiple leaders, as long as there eventually only exists one. This definition matches the *soft* Leader Election of chapter II.

We now introduce *safe* elections, in which we require the election algorithm to never include more than one **LEADER** during the execution, with the obvious exception of initial arbitrary configurations which may already include multiple **LEADER** robots.

Definition 5.2 (Safe Leader Election)**.**

A ***Leader Election*** *algorithm is* safe *if, for any execution whose initial configuration does not contain multiple robots that are either in a* **LEADER** *state, or carrying pending transitions to a* **LEADER** *state, no configuration contains more than one* **LEADER***, regardless of scheduling.*

Another interesting aspect of Leader Election is that the scheduler may be able to bias the election towards a subset of robot. In practice, this may imply that a faster robot has a greater chance of being elected, and it may be something we wish to avoid. We introduce the *unbiased* elections for this particular purpose.

Definition 5.3 (Unbiased Leader Election)**.** *A* ***Leader Election*** *algorithm is* unbiased *if, starting from a configuration in which robots are in the same state (same color, or same distance from the center of the SEC, for instance), robots have the same probability of being elected* **LEADER***, regardless of scheduling.*

Note that the probability in this definition is different from the probability of the Bernoulli trial. In practical, the probability of a robot r_1 being elected through an *unbiased* algorithm should always be $\frac{1}{n}$ for a network of n robots, regardless of the parameters of the Bernoulli trial.

Trivially, because the FSYNC scheduler has no decision power, any Leader Election algorithm is *unbiased* under the FSYNC scheduler.

5.3 Leader Election Based on Motion

To the best of our knowledge, the only algorithms for motion-based **Leader Election** in the ASYNC model were proposed by Canepa and Gradinariu Potop-Butucaru [19] and are described as algorithms 5.1 and 5.2. Algorithm 5.1 works for a network of three robots, while the algorithm 5.2 requires at least four robots. No motion based algorithm can exist for two robots.

Algorithm 5.1 Original **Leader Election** Algorithm by Canepa and Gradinariu Potop-Butucaru [19] for Three Robots

Compute the angles between two robots
if *my_angle* is the smallest
    Become **LEADER**
    Exit
else if *my_angle* is not the smallest but the other two are identical
    Become **LEADER**
    Exit
else if All angles are identical
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{3}$
    if Trial won
        Move perpendicular to the opposite side of the triangle in opposite direction

Algorithm 5.2 Original **Leader Election** Algorithm by Canepa and Gradinariu Potop-Butucaru [19] for Four or More Robots

Compute the smallest enclosing circle SEC
Compute the distance d_k to the center of SEC, for all robots $1 \leq k \leq n$
if $\forall k \neq myself, d_{myself} < d_k$, where $1 \leq k \leq n$
    Become **LEADER**
    Exit
if $\forall k \neq myself, d_{myself} \leq d_k$, where $1 \leq k \leq n$
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
        move a distance of $d_{myself} \cdot p$ towards the center of the SEC

In the first algorithm, the selected **LEADER** is either the robot which is on vertex with the smallest unique angle or, in the case of an isosceles triangle, the robot on the apex. It relies on Bernoulli trials to sort the case of the equilateral triangle. In the second algorithm, the **LEADER** state is defined as being the closest robot to the center of the SEC. It relies on Bernoulli trials to decide in the case of multiple robots closest to the center of the SEC.

These algorithms rely on the assumption that the scheduler is k-bounded, that is a robot r_2 can only be activated at most k times between each activation of robot r_1.

In the case of algorithm 5.2, we show why the k-bounded hypothesis is necessary by considering the case where two robots r_1 and r_2 are at an equal, smallest distance to the center of the SEC. We show the necessity of the hypothesis in the scheduling presented in figure 5.1. In this scheduling, after robot r_1 wins its Bernoulli trial, the ASYNC scheduler stops it and repeats the LCM cycle of r_2 until it wins its own Bernoulli trial.

Because of the k-bounded hypothesis, the scheduling has a probability of success of $1-(1-p)^k$, which, while being closer and closer to 1 as k grows, is smaller than 1. This implies that this election algorithm eventually works under this hypothesis. Now, removing the

k-bounded assumption is equivalent to k being infinite, and the probability of success of the scheduling being 1, so the algorithm fails. A similar scheduling exists for algorithm 5.1 in the case of robots forming an equilateral triangle.

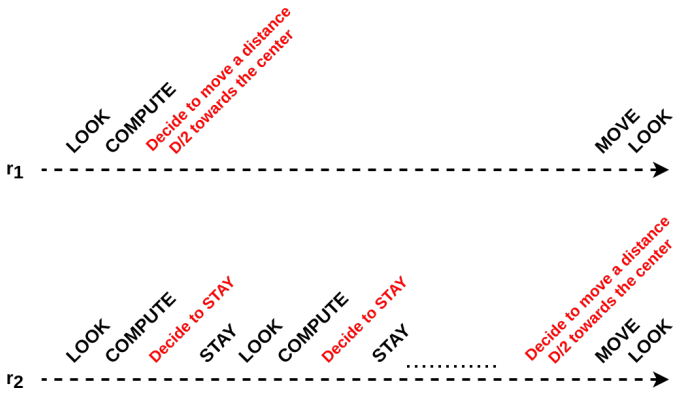


Figure 5.1 – Unbounded Scheduling Which Defeats Algorithm 5.2 in Unbounded ASYNC

Furthermore, it should be noted that these algorithms do not explicitly consider the case where multiple robots might be located at the same position.

Theorem 5.1. *If the adversary scheduler can control the coordinate system of each robot at any point in the execution, a network of two or more disoriented oblivious robots located at the same position (tower) cannot be separated, even using Bernoulli trials, under the unbounded ASYNC scheduler.*

Proof. Let us consider, without loss of generality, a network of three robots forming a tower, *i.e.* located at the same position. Because they are located at the same position, their snapshots are identical.

Without Bernoulli trials, because robots are disoriented, the FSYNC scheduler can prevent separation by manipulating the coordinate systems in such a way that robots always move to the same location.

Using Bernoulli trials, given an unbounded ASYNC scheduler, if a robot r_1 decides, to move out of the tower in a random direction, the scheduler can execute a scheduling identical to figure 5.1 and make all three robots win the trial. This way, all three robots decide to move to the same location. When activated simultaneously, they form a new tower at this new location. □

So, it is impossible to solve **Leader Election** for any number of robots if all robots are located at the same position.

In the case of the three robot algorithm, if a tower of two robots exist, *i.e.* two robots are located at the same position, then the angle is not defined for the robots on the tower. However, since the other robot is differentiated, then it can trivially be elected.

In the case of the algorithm for four or more robots, if multiple robots are located on the center of the SEC, because of theorem 5.1, then they cannot be separated, and so they should be ignored. Thus, in the case of a network of $n \geq 4$, but where only three robots are not located on the center of the SEC, the algorithm for three robots should be used.

We modify the algorithms to work without any additional scheduling hypothesis, by adding a losing action in the algorithm, and ignoring robots located on the center of the SEC. Our new algorithms are described as algorithms 5.3 and 5.4. This version of the algorithms cannot be

defeated in the unbounded ASYNC scheduler, because a robot cannot repeatedly try to win the Bernoulli trial: if a robot loses the Bernoulli trial, it moves in such a way that it is no longer a candidate for the election. These algorithms are also able to deal with any configuration where robots are at least on two separate locations.

On the other hand, the probabilistic proofs of Canepa and Gradinariu Potop-Butucaru [19] still apply. Thus, these enhanced algorithms work under an unbounded scheduler assumption.

Algorithm 5.3 Upgraded Motion-Based **Leader Election** for Three Robots ; Changes are Underlined

Compute the angles between two robots
if Other robots have identical positions, different than mine
    Become **LEADER**
    Exit
if *my_angle* is the smallest
    Become **LEADER**
    Exit
else if *my_angle* is not the smallest but the other two are identical
    Become **LEADER**
    Exit
else if All angles are identical
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{3}$
    if Trial won
        Move perpendicular to the opposite side of the triangle in opposite direction
    else if Trial lost
        Move towards the other robots, perpendicular to the opposite side of the triangle

Algorithm 5.4 Upgraded Motion-Based **Leader Election** for Four or more Robots ; Changes are Underlined

Compute the smallest enclosing circle SEC
Compute the distance d_k to the center of SEC, for all robots $1 \leq k \leq n$
if $d_{myself} > 0$ & $\forall k \neq myself, d_{myself} < d_k$, where $1 \leq k \leq n$ & $d_k > 0$
    Become **LEADER**
    Exit
if $d_{myself} > 0$ & $\forall k \neq myself, d_{myself} \leq d_k$, where $1 \leq k \leq n$ & $d_k > 0$
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
        Move a distance of $d_{myself} \cdot p$ towards the center of the SEC
    else if Trial lost
        Move a distance of at least $d_{myself} \cdot p$ opposite to the center of the SEC

5.4 Leader Election Based on Lights

We now study the case of $\mathcal{LUMINOUS}$ **Leader Election**.

As shown by Dieudonné *et al.* [37], there exists a well defined category of initial configurations in which the robots' positions may be used to deterministically elect a **LEADER**. However, we require algorithms to work regardless of initial configuration, so the position of the robots is not used. In practice this is useful for cases when the position of robots is symmetrical, such as when robots are forming a regular n-gon.

As such, while we previously considered that robots execute a MOVE phase during their cycles, we now make the hypothesis that robots are not moving during the execution, as it would not impact the election. Therefore, no MOVE phase is included.

This model can be summarized as robots being activated to either perform a snapshot, or change their color according to their snapshot.

Therefore, combining this with the hypothesis that robots should not move, the election should only be based on robots color.

In the case of $\mathcal{LUMINOUS}$ robots, the **LEADER** state is a color conveniently named LEADER.

Let us first look at the fairly trivial SSYNC **Leader Election** algorithm 5.5, also described in figure 5.3:

This $\mathcal{LUMINOUS}$ algorithm is essentially identical to the motion-based algorithm :

- There is a DEFAULT color.
- If there is no robot in the LEADER color, robots in the DEFAULT color have a given probability $p = \frac{1}{n}$ of switching to the LEADER color upon activation, with n the size of the network.
- If a robot in the LEADER color sees another LEADER, it switches to the DEFAULT color.

To keep figures readable, some abbreviations are defined:

- "X and Y" is true if both X and Y are true.
- "X or Y" is true if either X or Y is true.
- "$\nexists X$" is true if no robot of color X exist in the snapshot.
- "$\nexists(X,Y)$" is true if no robot of color X or Y exist in the snapshot.
- "$\exists X$" is true if at least one robot of color X exists in the snapshot.
- "$\exists(X,Y)$" is true if at least one robot of color X or Y exists in the snapshot.
- "$\forall X$" is true if the snapshot only contains robots of color X.
- "$\forall(X,Y)$" is true if the snapshot contains no robots of color other than X and Y.

Note that a robot is not included in its own snapshot.

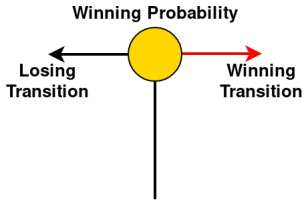


Figure 5.2 – Symbol for the Bernoulli Trial

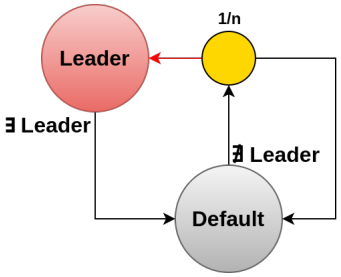


Figure 5.3 – SSYNC **Leader Election**

Algorithm 5.5 SSYNC **Leader Election**

```
if me.color = DEFAULT
    if ∄ other.color = LEADER
        Perform a Bernoulli trial with a probability of winning of p = 1/n
        if Trial won
            me.color ⇐ LEADER
else if me.color = LEADER
    if ∃ other.color = LEADER
        me.color ⇐ DEFAULT
```

Theorem 5.2. *Algorithm 5.5, described in figure 5.3 achieves motionless* ***Leader Election*** *under the SSYNC scheduler, using the optimal number of colors, for any number of robots*[1].

[1] In fact, this algorithm also functions properly for a network of a single robot !

Proof. Let us first consider a configuration containing only robots in the DEFAULT color.
If no robot is in the LEADER color, robots in the DEFAULT color attempt switching to the LEADER color with a probability of $\frac{1}{n}$. This leads to three possible cases :

1. The new configuration contains no robot in the LEADER color. This is identical to the initial configuration.
2. The new configuration contains a single robot in the LEADER color. Then, no robot can change color upon activation, and the election is over.
3. The new configuration contains multiple robots in the LEADER color. Upon activation, any robot LEADER that sees other LEADER robots switches back to DEFAULT. This eventually degrades to either the first of second case.

Because of the Bernoulli trial, the scheduler cannot chose which of the three case occurs after activating a subset of DEFAULT robots. The first and third case lead back to another Bernoulli trial, and the second case has a non-zero probability of happening. Thus the network eventually reaches the second configuration in which the is a single robot in the LEADER color.

Since using one color is trivially identical to using no color, and **Leader Election** is trivially impossible using no color for two robots, using 2 colors is optimal.

□

This algorithm, similarly to the motion based approach, is self-stabilizing, achieves **Leader Election** for the SSYNC scheduler and fails under a unbounded ASYNC scheduler and would require k-boundedness, for the same reasons.

In fact, under the ASYNC scheduler, the proof for theorem 5.2 does not hold because the scheduler can reliably induce the third configuration, where multiple robots are in the LEADER color, with a probability of 1. The scheduler activates a first robot repeatedly until it wins the trial and holds the color change. The Bernoulli trial happens during the COMPUTE phase, before the color change. Unless the two events were atomic, there exists a moment in the execution when the scheduler is aware of the following color change and can choose to stop the COMPUTE phase before the change of color. The scheduler then activates a second robot repeatedly until it also eventually wins, and similarly holds the color change. This is then repeated for every robot in the network. Then, all robots in the network are activated and switch to the LEADER color. Therefore, the scheduler can deterministically have all robots win the Bernoulli trial. This obviously prevents the SSYNC algorithm from being used in ASYNC.

However, it improves upon its motion based by achieving **Leader Election** for any number of robots, where motion-based requires at least three. It also does not require robots to have separate positions.

This algorithm is obviously not safe, as there is a non-zero chance that several robots become LEADER after activating multiple DEFAULT robots simultaneously.

Similarly, this algorithm is obviously not unbiased. The scheduler can activate one robot repeatedly until it is elected. So, the scheduler can choose a robot and ensure with a probability of 1 that it is elected as LEADER.

Note that, using the same reasoning than with our improved motion based algorithm, it is possible to achieve **Leader Election** in unbounded ASYNC, by adding the following new constraints:

- If a robot loses the Bernoulli trial, it switches to the FOLLOWER color.
- A robot in the FOLLOWER color switches to the DEFAULT color if all robots are in the FOLLOWER color.

Algorithm 5.6 includes these modifications and is presented in figure 5.4.

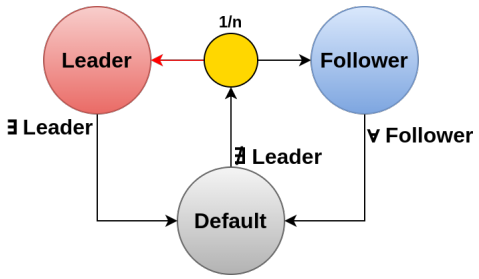


Figure 5.4 – ASYNC **Leader Election** algorithm

Algorithm 5.6 ASYNC **Leader Election**

if me.color = DEFAULT
  if $\nexists$ other.color = LEADER
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
      me.color $\Leftarrow$ LEADER
    else
      me.color $\Leftarrow$ FOLLOWER
else if me.color = FOLLOWER
  if $\forall$ other.color = FOLLOWER
    me.color $\Leftarrow$ DEFAULT
else if me.color = LEADER
  if $\exists$ other.color = LEADER
    me.color $\Leftarrow$ DEFAULT

Theorem 5.3. *Algorithm 5.6, described in figure 5.4 achieves motionless* ***Leader Election*** *under the ASYNC scheduler for any number of robots.*

Proof. First, a rapid analysis of the algorithm shows that the only locked configuration contains a single robot in the LEADER color and all other robots in either the FOLLOWER or DEFAULT color. So, if this configuration has a non-zero probability of happening regardless of scheduling, the network is deadlocked and **Leader Election** is eventually achieved. Let us now look at possible configurations:
If the configuration contains a single robot in the LEADER color with no pending transition to DEFAULT and no robots in the DEFAULT color with pending transitions to LEADER, the election is over.
If the configuration contains multiple robots in the LEADER color, all other robots are stuck and the configuration eventually degrades to one or zero robots in the LEADER color.
If the configuration contains a single robot in the LEADER color and robots in the DEFAULT color with pending transitions to LEADER, it eventually degrades to the previous case of multiple robots in the LEADER color.
If the configuration contains no robot in the LEADER color, given enough activations, at least one robot eventually attempts the Bernoulli trial. Regardless of scheduling, there is a non-zero probability of reaching a configuration where a single robot has a pending transition from DEFAULT to LEADER. At this point, robots in the FOLLOWER color are either stuck, or carrying a pending transition to DEFAULT. Other robots in DEFAULT may either carry no transition, or transitions to FOLLOWER. Because any new snapshot taken by a FOLLOWER robot leads to no transition, all sequences of activations by scheduler have a non-zero probability of leading to

no other robots winning the Bernoulli trial and being stuck in the FOLLOWER color. Then, the robots with a pending transition to LEADER must be elected.
This means that this algorithm prevents the scheduler from reliably getting a second robot to win the Bernoulli trial after a first robot did. Therefore, the scheduler cannot reliably prevent the election from finishing.

□

Note that the provided SSYNC algorithm is actually somewhat 'wait-free', in the sense that as long as at least one robot in the network is not crashed, and no robots are crashed in the LEADER color, then **Leader Election** is eventually achieved. This is not the case of the provided ASYNC algorithm. We provide a more general result:

Definition 5.4 (Self-Stabilizing, Non-Blocking $\mathcal{LUMINOUS}$ **Leader Election**)**.**
Leader Election *is* self-stabilizing *and* non-blocking *[57] if there exists configurations containing no robot in the* LEADER *color, and for any such configuration, there exists at least one robot r_1 such that, after enough activations of only r_1, r_1 reaches the* LEADER *color.*

Theorem 5.4 (Non-Blocking $\mathcal{LUMINOUS}$ **Leader Election**)**.**
There is no $\mathcal{LUMINOUS}$ probabilistic algorithm that achieves self-stabilizing, non-blocking, ***Leader Election*** *under the unbounded ASYNC scheduler.*

Proof. To ensure **Leader Election** for any initial configuration, it is required to use at least one Bernoulli trial [37]. Let us assume a self-stabilizing, non-blocking, algorithm exists. this algorithm ensures that a single non crashed robot can reach the LEADER color if no robots are in the LEADER color. So, if a robot fails the Bernoulli trial, it must be able to attempt it again without requiring any activation of other robots. Otherwise, the algorithm would not be non-blocking.
Then is it possible for the ASYNC scheduler to activate a first robot repeatedly until it wins the trial, hold the color change and activate a second robot repeatedly until it also eventually wins, and hold the color change.
Then, two robots in the network go to the LEADER color. Therefore, the scheduler can deterministically have multiple robots win the Bernoulli trial and reach the LEADER color. Let us now consider the algorithm can try to eliminate all LEADER robots but one. If the process is deterministic, then activating both robots simultaneously results in no robot being singled out, so the process must use a Bernoulli trial. If one outcome of the Bernoulli trial leads a robot to stay in the LEADER color, then the same reasoning applies, and both robot eventually leave the LEADER color to the same color. If no outcome leads to the LEADER color, then the colors reached by robots both allow for trying the initial trial again, as otherwise the algorithm would not be non-blocking. So, the scheduling can be repeated infinitely, and no robot can ever be singled out. Therefore, **Leader Election** is impossible.

□

It should be noted that nothing in this proof explicitly relies on the $\mathcal{LUMINOUS}$ model, but only on the fact that the scheduler can predict and hold the change of state, and multiple , so the same reasoning also applies to some $\mathcal{OBLOT}$ algorithms:

Definition 5.5 (Self-Stabilizing, Non-Blocking **Leader Election**)**.**
Leader Election *is* self-stabilizing *and* non-blocking *[57] if there exists configurations containing no robot in the* **LEADER** *state, and for any such configuration, there exists at least one robot r_1 such that, after enough activations of only r_1, r_1 reaches the* **LEADER** *state.*

Theorem 5.5 (Non-Blocking **Leader Election**)**.**
There is no probabilistic algorithm that achieves self-stabilizing, non-blocking ***Leader Election*** *under the unbounded ASYNC scheduler.*

Proof. To ensure **Leader Election** for any initial configuration, it is required to use at least one Bernoulli trial [37]. Let us assume a non-blocking algorithm exists. this algorithm ensures that a single non crashed robot can reach the **LEADER** state (color, position, *etc.*). So, if a robot fails the Bernoulli trial, it must be able to attempt it again without requiring any activation of other robots. Otherwise, the algorithm would not be non-blocking.
Then is it possible for the ASYNC scheduler to activate a first robot repeatedly until it wins the trial. Since all changes of state for mobile robots happen after the Bernoulli trial, the scheduler can hold the state change and activate a second robot repeatedly until it also eventually wins, and hold the state change.
Then, two robots in the network go to the **LEADER** state. Therefore, the scheduler can deterministically have multiple robots win the Bernoulli trial and reach the **LEADER** state. Let us now consider the algorithm can try to eliminate all **LEADER** robots but one. If the process is deterministic, then activating both robots simultaneously results in no robot being singled out, so the process must use a Bernoulli trial. If one outcome of the Bernoulli trial leads a robot to stay in the **LEADER** state, then the same reasoning applies, and both robot eventually leave the **LEADER** state to a new, identical, state. If no outcome lead to the **LEADER** state, then the state reached by robots both allow for trying the initial trial again, as otherwise the algorithm would not be non-blocking. So, the scheduling can be repeated infinitely, and no robot can ever be singled out. Therefore, **Leader Election** is impossible. □

As an example, let us confront this proof to algorithm 5.2, for four or more robots by Canepa and Gradinariu Potop-Butucaru [19].
If the **LEADER** state is defined as being the only robot closest to the center of the SEC, then the algorithm is indeed self-stabilizing and non-blocking. In a configuration with no robot in the **LEADER** state, at least two robots are closest to the center of the SEC and, as shown in figure 5.1, the unbounded scheduler can ensure that both deterministically move towards the center at the same distance. So, no robot actually ever reaches the **LEADER** state.
If the **LEADER** state is defined as being closest to the center of the SEC, then all configuration contain at least one robot in the **LEADER** state.

5.5 Safe Leader Election

In order to achieve safe **Leader Election**, we define a new color, WIN, and redefine the properties of the algorithm accordingly. Changes for the original are underlined.

- There is a DEFAULT color.
- If they see no robot neither in the LEADER color nor the WIN color, robots in the DEFAULT color have a given probability $p = \frac{1}{n}$ of switching to the WIN color upon activation, with n the size of the network.
- If a robot in the WIN color sees another WIN robot or a LEADER robot, it switches to DEFAULT. Otherwise it switches to the LEADER color.
- If a robot in the LEADER color sees another LEADER, it switches to the DEFAULT color.

In general terms, the purpose of the WIN color is to act as a buffer which ensures only one robot can reach the LEADER color.

Lemma 5.1. *Under the ASYNC scheduler, unless the initial configuration contains two or more robots which are in the* LEADER *color, or have pending transitions to the* LEADER *color, it is impossible for more than one robot to reach the* LEADER *color for a* ***Leader Election*** *algorithm using the* WIN *color.*

Proof. Without loss of generality, let us prove this property for two robots trying to both reach the LEADER color.
For the purpose of contradiction, let us assume there exists an execution that includes a configuration of two LEADER robots, but for which the initial configuration C_{init} does not include two or more robots in the LEADER color or with pending transitions to the LEADER color.
It is possible to rewind this execution, starting from the the configuration containing two robots in the LEADER color. Then, among the configurations that can be reached through this process, we should find the configuration C_{init}.
Starting from a configuration that contains two robot in the LEADER color, let us rewind the execution by one activation. Two different configurations can be reached:

- C_1^- : robot r_1 is in the LEADER color, and r_2 in the WIN color with a pending transition to the LEADER color.
- C_2^- : both robots are in the WIN color with a pending transition to the LEADER color.

Both these configurations can lead to a configuration with two robots in the LEADER color with the right activations. However, both contain two robots in the LEADER color or with pending transitions to the LEADER color. Therefore, neither are C_{init}, so we need to rewind at least one more activation.

Rewinding the configuration C_1^- by one activation leads to the following reachable configurations:

- C_{1-1}^- : both robots are in the WIN color with a pending transition to the LEADER color.
- C_{1-2}^- : robot r_1 is in the LEADER color, and r_2 in the WIN color with no pending transitions.
- C_{1-3}^- : both robots are in the WIN color. r_1 has a pending transition to the LEADER color and r_2 has no pending transitions.

Configuration C_{1-1}^- is identical to C_2^- and is analyzed below.
Configuration C_{1-2}^- does not lead to two robots in the LEADER color, as the activation of r_2 leads to a pending transition to the DEFAULT color since a LEADER robot is detected. Therefore, it cannot be part of the execution.
Similarly, activating r_2 in configuration C_{1-3}^- leads to a pending transition to the DEFAULT color, while activating r_1 leads to configuration C_{1-2}^-. So it cannot be part of the execution either.

We now rewind the second configuration C_2^- by one activation :

- C_{2-1}^- : both robots are in the winning color with no pending transitions.
- C_{2-2}^- : both robots are in the winning color. r_1 has a pending transition to the LEADER color and r_2 has no pending transitions.

In configuration C_{2-1}^-, the activation of either robot leads to a pending transition to the DEFAULT color, as another winning robot can be detected. Therefore, this cannot be part of the execution.
Configuration C_{2-2}^- is identical to C_{1-3}^- and has already been proven to not be part of the execution.
Therefore, there is no configuration that does not contain two robots in the LEADER color or with pending transitions to the LEADER color that leads to two robots in the LEADER color.
Following the same principle, this proof can easily be expanded to any number of robots in the LEADER color.

□

Note that since this lemma is true for the general ASYNC, it is also true for SSYNC. Therefore, ensuring the above properties of the WIN color to either the SSYNC algorithm or the ASYNC algorithm ensure that it becomes *safe*, with the cost of an additional color.

Theorem 5.6. *Algorithms 5.7 and 5.8, presented in figures 5.5 and 5.6 allow for safe* ***Leader Election*** *under the SSYNC and ASYNC schedulers, respectively.*

Proof. According to theorems 5.2 and 5.3 both these algorithms ensure that, eventually, there is a single robot in the WIN color, as it replaces the former LEADER color. All other robots are then stuck. Following lemma 5.1, these algorithms ensure the safety property, as the WIN color ensures that only one robot can transition to the LEADER color. So the WIN robot switches to LEADER and the election is achieved.
Hence, both algorithms allow for safe **Leader Election** under their respective scheduler.
□

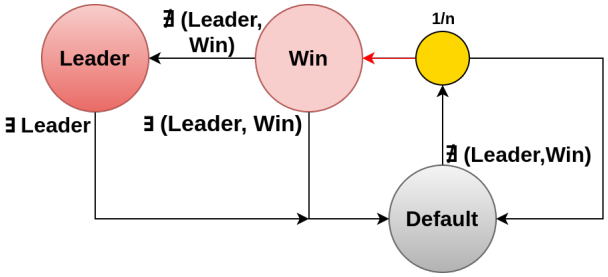


Figure 5.5 – Safe SSYNC **Leader Election**

Algorithm 5.7 Safe SSYNC **Leader Election**

if me.color = DEFAULT
  if ∄ other.color = WIN **and** and ∄ other.color = LEADER
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
      me.color ⇐ WIN
else if me.color = WIN
  if ∃ other.color = WIN **or** ∃ other.color = LEADER
    me.color ⇐ DEFAULT
  else
    me.color ⇐ LEADER
else if me.color = LEADER
  if ∃ other.color = LEADER
    me.color ⇐ DEFAULT

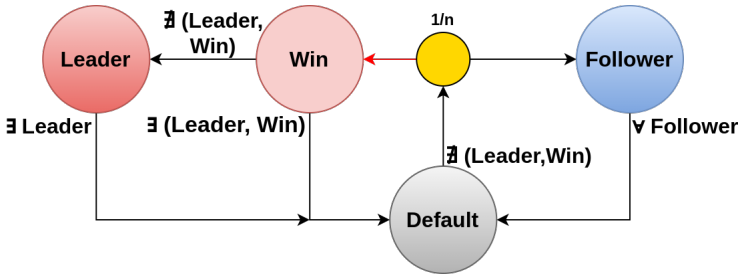


Figure 5.6 – Safe ASYNC **Leader Election**

Algorithm 5.8 Safe ASYNC **Leader Election**

if me.color = DEFAULT
  if $\nexists$ other.color = WIN **and** and $\nexists$ other.color = LEADER
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
      me.color $\Leftarrow$ WIN
    else
      me.color $\Leftarrow$ FOLLOWER
else if me.color = FOLLOWER
  if $\forall$ other.color = FOLLOWER
    me.color $\Leftarrow$ DEFAULT
else if me.color = WIN
  if $\exists$ other.color = WIN **or** $\exists$ other.color = LEADER
    me.color $\Leftarrow$ DEFAULT
  else
    me.color $\Leftarrow$ LEADER
else if me.color = LEADER
  if $\exists$ other.color = LEADER
    me.color $\Leftarrow$ DEFAULT

5.6 Unbiased Leader Election

We introduce an algorithm that ensures unbiased ASYNC **Leader Election**. It is based around the principle that robots should all attempt the Bernoulli trial and, unless the election is successful, all switch back to DEFAULT, and start again. This is done through the addition of a RESET color. Unfortunately, the use of the FOLLOWER color appears mandatory, even in SSYNC.

- There is a DEFAULT color.
- If there is no robot in the RESET color, robots in the DEFAULT color have a given probability $p = \frac{1}{n}$ of switching to the LEADER color upon activation.
- If a robot loses the Bernoulli trial, it goes to a FOLLOWER color.
- If a robot in the LEADER color sees another LEADER and no DEFAULT, or sees a RESET, it switches to the RESET color.
- A robot in the FOLLOWER color switches to the RESET color if all robots are in the FOLLOWER color, or if there is a RESET robot.
- A robot in the RESET color switches to the DEFAULT color if all robots are either in the RESET or DEFAULT color.

Theorem 5.7. *Algorithm 5.9, described in figure 5.7 allows for unbiased* ***Leader Election*** *under the ASYNC scheduler.*

Proof. First, a rapid analysis of the algorithm shows that the only locked configuration contains a single robot in the LEADER color and all other robots in the FOLLOWER color. In all other configurations, there exists at least one robot that can change color upon activation.
If the initial configuration contains no, or more than one LEADER, then all robots should gather in DEFAULT or RESET within finite time.
If a RESET robot exists alongside LEADER or FOLLOWER robots and no DEFAULT, it is stuck and the other robots must switch to RESET. Then, all robots are in the RESET color.
If a DEFAULT robot exists alongside the RESET robots, it must wait for all other robots to switch to RESET, and then for RESET robots to switch to DEFAULT. Then all robots are in the DEFAULT color.
If no RESET robot exist, and there are some robots in the DEFAULT color, alongside robots in the LEADER or FOLLOWER colors, then the DEFAULT robots must switch to either color before these robots can change color. Ignoring the obvious case of a single LEADER robot and the rest of the network in the FOLLOWER color, once the network is parted in these two colors, the first activation switches a robot to the RESET color and all robots must switch to the RESET color.
We have shown that, unless the election is finished, robots eventually all gather in a single color.
We now assume all robots are in a single color.
If all robots are in the LEADER color, then the first activated robot goes to the RESET color and is stuck until all other robots have reached RESET.
If all robots are in the FOLLOWER color, then the first activated robot goes to the RESET color and is stuck until all other robots have reached RESET.
When all robots are in the RESET color, they switch to the DEFAULT color and are stuck until there are no more robot in the RESET color.
When all robots are in the DEFAULT color, they non-deterministically switch to the LEADER and FOLLOWER. Since there are no more robots in the RESET color, they are stuck in their respective color until there are no more DEFAULT robots left.

Three cases then arise :

1. There are no robot in the LEADER color.
2. There is only one robot in the LEADER color.
3. There are multiple robots in the LEADER color.

In the first case, there are only FOLLOWER robots and the first activated robot switches to RESET and is stuck until all robots switch to RESET.
In the second case, the election is over and the configuration is locked.
In the third case, FOLLOWER robots are stuck and the first LEADER robot to be activated switches to RESET. Then all robots must switch to RESET.
This ensures that the only way for a robot to attempt a Bernoulli trial is for all other robots also attempt it, with the same parameters. Therefore, all robots have a $\frac{1}{n}\left(\frac{n-1}{n}\right)^n$ probability of being elected each time they cycle through the algorithm. □

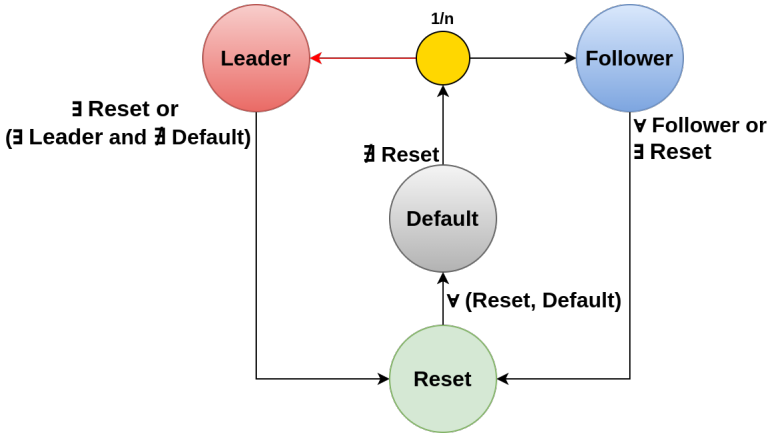


Figure 5.7 – Unbiased ASYNC **Leader Election**

Algorithm 5.9 Unbiased ASYNC **Leader Election**

if me.color = DEFAULT
  if ∄ other.color = RESET
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
      me.color ⇐ LEADER
    else
      me.color ⇐ FOLLOWER
else if me.color = FOLLOWER
  if ∀ other.color = FOLLOWER **or** ∃ other.color = RESET
    me.color ⇐ RESET
else if me.color = RESET
  if ∀ other.color = {RESET,DEFAULT}
    me.color ⇐ DEFAULT
else if me.color = LEADER
  if ∃ other.color = RESET **or** (∃ other.color = LEADER **and** ∄ other.color = DEFAULT)
    me.color ⇐ RESET

5.7 Safe Unbiased Leader Election

This algorithm is built as the inclusion of the WIN color in the unbiased **Leader Election** algorithm described in figure 5.7. This algorithm is described in figure 5.8.

Proving this algorithm is combination of the proofs for the safe and unbiased algorithms, and is not detailed.

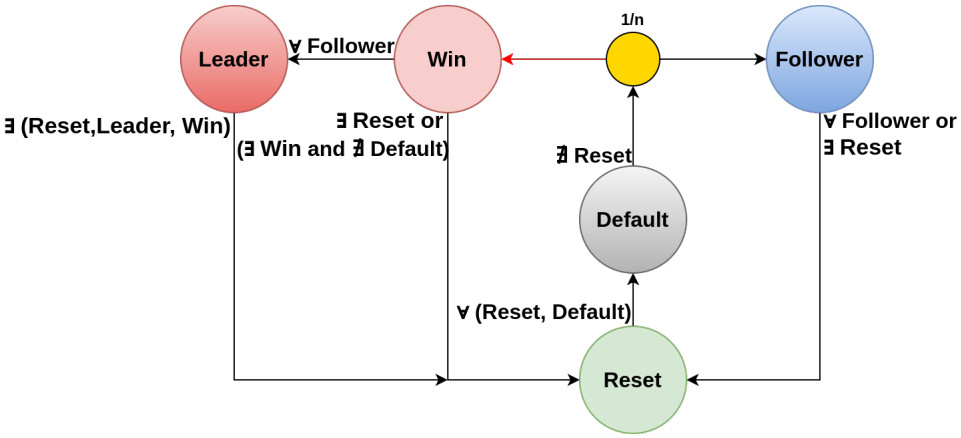


Figure 5.8 – Safe Unbiased ASYNC **Leader Election**

Algorithm 5.10 Unbiased Safe ASYNC **Leader Election**

if me.color = DEFAULT
  if ∄ other.color = RESET
    Perform a Bernoulli trial with a probability of winning of $p = \frac{1}{n}$
    if Trial won
      me.color ⇐ WIN
    else
      me.color ⇐ FOLLOWER
else if me.color = FOLLOWER
  if ∀ other.color = FOLLOWER **or** ∃ other.color = RESET
    me.color ⇐ RESET
else if me.color = RESET
  if ∀ other.color = {RESET,DEFAULT}
    me.color ⇐ DEFAULT
else if me.color = WIN
  if ∃ other.color = RESET **or** (∃ other.color = WIN **and** ∄ other.color = DEFAULT)
    me.color ⇐ RESET
  else if ∀ other.color = FOLLOWER
    me.color ⇐ LEADER
else if me.color = LEADER
  if ∃ other.color = RESET **or** ∃ other.color = LEADER **or** ∃ other.color = WIN)
    me.color ⇐ RESET

Conclusion: The Power of Lights

We introduced the $\mathcal{LUMINOUS}$ model, which allows robots to store and transmit limited amounts of information with no cost to the realism of the model. This allows for reliable communication and helps dealing with the issue of unrealistic scheduling.

In this context, we proved a new result: $\mathcal{LUMINOUS}$ robots under the SSYNC scheduler are not more powerful than oblivious robots under the FSYNC scheduler. We then designed an algorithm which allows for two $\mathcal{LUMINOUS}$ robots to gather under the ASYNC scheduler, using only two colors, *i.e.* a single bit information.

Due to the difficulty of proving this algorithm, we created and proved a model checking system for two-robot **Gathering** under most schedulers. We used this system to verify the proofs of several known algorithms, and to prove an algorithm to solve **Rendezvous** in the case of non-atomic lights. While proving the model checker, we also demonstrated the counter intuitive result that, under the ASYNC scheduler, a system of oblivious robots is not necessarily self-stabilizing.

We demonstrated an upgraded version of the state-of-the-art, motion based algorithm for **Leader Election**, and used the $\mathcal{LUMINOUS}$ model to introduce new solutions, with the added properties of safe and unbiased elections. Combined with the possibility of electing a **LEADER** in a network of two robots, these algorithms allow *safe* and *unbiased* elections under the ASYNC scheduler.

Part II

Unreliable Vision

Chapter 6

Uncertain Visibility

Ever since the $\mathcal{OBLOT}$ model was introduced, its full visibility sensor was considered unrealistic by practitioners: since robot visual sensors have physical limitations (*e.g.* limited resolution for omnidirectional 3D cameras [58, 71]), this intrinsically yields a limitation of the visibility range. As a result, in the literature, three models have been used for visibility sensors: the full visibility model, where all robots can see all other robots, the limited visibility model [3, 46, 50, 51, 82], where there exists a limit $\lambda > 0$ such that all robots closer than λ are seen and all robots further than λ are *not seen*, and the obstructed visibility model as described in chapter 6.5.

However, limited visibility only partly addresses this issue, as further robots that are at the same distance (greater than λ) may be seen or not seen due to unpredictable reasons. Two incorrect outputs could be obtained by an observer about a particular robot:

1. False positives: no robot exists at a position, but one is output by the vision sensor.
2. False negatives: a robot exists at a position, but none is output by the vision sensor.

False positives can be dealt with using known techniques (such as marker-based detection [63]), so we do not consider them in this work.

Figure 6.1 – Example of markers used for preventing false positives

False negatives, that are not addressed by the limited visibility model, are the focus of this chapter. We define *uncertain* visibility sensors for mobile robots as sensors that satisfy the two following properties:

1. Every robot closer than λ is output by the sensor.
2. A subset of the robots further than λ is output by the sensor.

Note that a subset including all such robots represents the full visibility model, while the empty subset represents the limited visibility model.

Since we are interested in characterizing the exact limits of models for the computability of tasks in the $\mathcal{OBLOT}$ model, we consider that the subset of robots beyond λ that remain visible is decided by an adversary. Also, a robot r_3 that is at the same distance from two distinct robots r_1 and r_2 may be output by r_1's visibility sensor, but not by r_2's.

Our work is inspired by the idea introduced by Santoro and Widmayer [80], describing the consequences of transmission faults that are controlled by an adversary in a synchronous system

for the agreement problem in a distributed system.

They considered:

- Omission faults *i.e.* messages that are sent but not received.
- Corruption faults *i.e.* messages that are different from when they were sent.
- Addition faults *i.e.* messages that are received but that were not sent.

In our context, omission faults by visibility sensors correspond to false negatives. Addition faults can be considered as false positives in our context. Corruption faults are tantamount to erroneous positions returned by the sensors.

In this chapter, we consider omission faults in the case of visibility sensors. Similarly to the paper of Santoro and Widmayer [80], we retain the synchronous scheduling of individual entities. Our model differs from the work of Santoro and Widmayer [80] with our usage of the λ parameter from the $\mathcal{OBLOT}$ model. The reason being to integrate well with previous abstraction for mobile robotic entities sensors, in order to enable comparison with previous results obtained in this model.

6.1 Model Definition and Basic Results

With the notable exception of restricted visibility sensors, our model matches the classical $\mathcal{OBLOT}$ model [43]. Robots are modeled as points in a bidimensional Euclidean space, are anonymous and uniform (that is, they execute the same code and have no identifiers), and unless specified otherwise, cannot communicate explicitly (but can observe other robots positions in their ego-centered coordinate system) and are oblivious (that is, they cannot remember their past actions).

The scope of this section is limited to the FSYNC scheduler, as it more closely matches the synchronous setting of the paper by Santoro and Widmayer [80]. So, in every synchronous step, every robot is scheduled for execution, and performs a complete LOOK-COMPUTE-MOVE cycle. Movements are rigid, meaning a robot always reaches its target at the end of a LCM cycle.

We introduce a new visibility model that is an extension of the already existing *full visibility* and *limited visibility*. We refer to this model as the *uncertain visibility* model. In *limited visibility*, there exists a distance λ, which may or may not be known to robots, so that if the distance between two robots r_1 and r_2 is greater than λ, r_1 and r_2 cannot see each other. Now, if r_1 and r_2 are closer than λ, then they can see each other. In the *full visibility* model, this distance λ is infinite.

In our new visibility model, we define the distance λ such that if the distance between two robots r_1 and r_2 is greater than λ, depending on the adversary, it is uncertain whether r_1 and r_2 can see each other. Yet, if r_1 and r_2 are closer than λ, then they always see each other. Under this model, full visibility corresponds to the case where $\lambda = \infty$, while the limited visibility corresponds to the case where the adversary prevents visibility whenever r_1 and r_2 are further away than λ.

In this chapter, our focus is on the *uncertain* part of the visibility model. As in the $\mathcal{OBLOT}$ model, we consider that $\lambda > 0$ is unknown to the robots and that no two robots are closer than λ in the initial configuration. To introduce *selective* vision among robots, we consider that correctly viewing a robot r_1 is similar to correctly receiving a "visibility message" from r_1. Then, the adversary may simply block a subset of the visibility messages among robots when they are further than λ away from one another. We introduce two classes of visibility messages adversaries:

Definition 6.1 (k-random)**.**
k-random adversaries can make up to *k visibility messages disappear in each synchronous round. The number of messages is chosen for each round by the vision adversary, but the messages are chosen uniformly at random. A k-random adversary corresponds to* ***probabilistic visibility*** *in the sequel.*

Definition 6.2 (k-enemy)**.**
k-enemy adversaries can make up to *k visibility messages disappear in each synchronous round. The number of messages is chosen for each round by the vision adversary, and those messages are also chosen by the adversary. A k-enemy adversary corresponds to* ***adversarial visibility*** *in the sequel.*

From these definitions, we make the following observations, considering a FSYNC network of n robots and $n \cdot (n-1)$ visibility messages sent each round.

Observations.

1. *0-random and 0-enemy adversaries are identical, and equivalent to full visibility.*
2. *$n \cdot (n-1)$-random and $n \cdot (n-1)$-enemy both encompass limited visibility.*
3. *Any task that can be solved against the k-enemy adversary can be solved against the k-random adversary.*
4. *Because vision adversaries can block* up to *a maximum number of visibility messages, if a task can be solved against the k-random (respectively, the k-enemy), it can also be solved against any i-random (respectively, i-enemy) adversary where $0 \leq i \leq k$.*

Definition 6.3 (Necessary and Sufficient)**.**
If a task $\mathcal{T}$ cannot be solved against the $(k+1)$-random (respectively, the $(k+1)$-enemy) adversary but can be solved against the k-random (respectively, the k-enemy) adversary, we say the k-random (respectively, the k-enemy) adversary is necessary and sufficient for $\mathcal{T}$.

A key difference between this uncertain model and the limited visibility model is the **lack of symmetry** in vision: in the limited visibility model: if a robot r_1 sees another robot r_2, then it is certain that r_2 is also able to see r_1. This is not the case in the uncertain visibility model.

Our notion of uncertain visibility also relates to the notion of obstructed visibility [66], where a robot r_1 cannot see a robot r_3 if there exists a third robot r_2 inside the line of sight between r_1 and r_3. In a network of n robots, there are at most $n \cdot (n-1) - 2 - 2 \cdot (n-2) = n^2 - 3 \cdot n + 2$ obstructions (when all robots are collinear). When all robots are more than λ away from each other, any obstructed configuration can be represented by an identical transparent configuration and a particular $(n^2 - 3 \cdot n + 2)$-enemy adversary. So, if an algorithm can perform a task against the $(n^2 - 3 \cdot n + 2)$-enemy adversary, it can perform the task under obstructed visibility. It should be noted that obstructed visibility actually allows more information to be sensed by robots, as the obstructing robots know they are causing obstructions, while in the case of the $(n^2 - 3 \cdot n + 2)$-enemy vision adversary, no robot is aware of which messages are dropped. So the reverse, *i.e.* if an algorithm can perform a task under obstructed visibility, it can perform a task against the $(n^2 - 3 \cdot n + 2)$-enemy adversary, is not true.

6.2 FSYNC *n* robots Gathering

In this section, we consider the benchmarking problem of *eventually* **Gathering** n robots at the same location, not known beforehand, starting from any initial configuration. As seen in

chapter 2.1, in general, the problem is impossible to solve deterministically in the SSYNC $\mathcal{OBLOT}$ model [77, 83], but remains solvable in the FSYNC $\mathcal{OBLOT}$ model [8, 24, 83] when robots execute the "center of gravity" algorithm: each robot simply moves to the center of gravity of every observed robot [24] (if robots are endowed with multiplicity detection), or to the center of gravity of occupied positions [8] (if robots cannot detect multiplicity points). In the SSYNC model (and hence the FSYNC model), the center of gravity algorithm is known to solve the weaker problem of **Convergence**: in any execution, for any $\varepsilon > 0$, all robots are *eventually* within ε of one another.

We study the behavior of this *move to center of gravity* algorithm in the case of probabilistic and adversarial visibility.

Theorem 6.1. *In FSYNC, if n robots can achieve* ***Convergence*** *against a k-enemy adversary using the* move to center of gravity *algorithm, then they also achieve* ***Gathering****.*

Proof. In each FSYNC round, robots may or may not see other robots. However, we know that the visibility parameter λ is positive. Now, if robots can achieve **Convergence**, there exists ε ($0 < \varepsilon < \lambda$) such that in any execution, all robots are within ε of one another (by definition of **Convergence**). Therefore, after a finite number of steps, the smallest enclosing circle (SEC) for the entire network has a diameter smaller than λ. At this point, all robots can reliably see one another and the same result as in previous works [8, 24] applies. So, **Gathering** is eventually achieved. □

Theorem 6.2. *In FSYNC, deterministic* ***Gathering*** *can be achieved for any k-random adversary such that* $k \leq n \cdot (n-1) - 1$.

Proof. Let us consider the smallest enclosing circle (SEC) of the *n* robots network.

Definition 6.4 (Smallest Enclosing Circle)**.** *Given a set* $\mathbb{S}$ *of positions on the plane, the smallest enclosing circle (SEC) of* $\mathbb{S}$*, named SEC* $(\mathbb{S})$ *is the unique smallest circle enclosing all positions of* $\mathbb{S}$*.*

Definition 6.5 (Robot Position on the SEC)**.**
Given a set $\mathbb{S}$ *of positions p on the plane:*

1. *A point p is* inside *the SEC if p is in the disc bounded by SEC or the SEC itself.*
2. *A point p is* strictly inside *the SEC if p is in the disc bounded by the SEC but not on the SEC itself.*
3. *A point p is* critical *iff SEC* $(\mathbb{S}) \neq$ *SEC* $(\mathbb{S} \backslash \{p\})$.

Note that critical points can obviously not be strictly inside the SEC.
To prove theorem 6.2, we prove that the diameter of the SEC decreases towards zero and *eventually* becomes smaller than λ. In the remaining of this proof, we call r_1 and r_2 the robots that receive and send a visibility message, respectively. We note the three following cases:

1. If r_1 is strictly inside the SEC, and because r_2 is, by definition, inside the SEC, when r_1 targets the center of gravity, it targets the midpoint between r_1 and r_2, which is *strictly inside* the SEC. Therefore, the movement of r_1 does not change the diameter of the SEC.
2. Now, if r_1 is on the SEC itself, and r_1 is not critical, then because r_2 is inside the SEC, the target of r_1 is strictly inside the SEC. Therefore, the number of robots on the SEC itself decreases.
3. Last, if r_1 is critical. As r_1 moves strictly inside the SEC, the new SEC diameter decreases.

If the adversary is $(n \cdot (n-1)-1)$-random, then as long as the SEC has a positive diameter $d > 0$, then case 2 has a positive probability of happening, and leads to case 3. So, the following events occur with positive probability: the number of robots on the SEC decreases towards 2, and the diameter of the SEC decreases to diameter $d' < d$.
For the purpose of contradiction, let us assume that the diameter of the SEC continuously decreases towards a diameter z that is greater than λ and never becomes smaller than z. Then for any $\varepsilon > 0$, if the diameter d of the current SEC is $z+\varepsilon$, then the diameter of the next SEC, $d' < d$, is still greater than z.
Let us consider the third case, where r_1 is critical.

We consider the worst possible case, where a number k of other robots are closer than λ from r_1, and r_2 is located infinitely close the the boundary of the SEC, at distance larger than λ. Then, r_1 chooses a target T_1 that is located on the $[r_1\ r_2]$ chord, at a distance of at least $\frac{\lambda}{k+2}$.

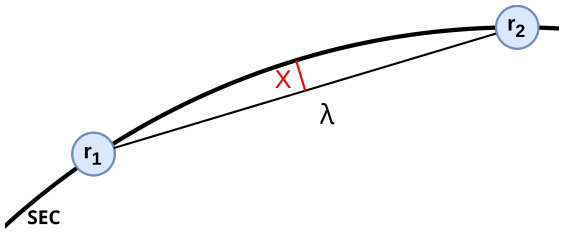


Figure 6.2 – Minimum Traveled Distance Towards the Center of the SEC for $k = 0$.

Then, from the intersecting chords theorem, T_1 is chooses a target at a distance of at least $\frac{d}{2} - \frac{\sqrt{d^2 - 4 \cdot \frac{\lambda^2 \cdot (k+1)}{(k+2)^2}}}{2}$ closer to the center of the SEC. From the Taylor series, this distance is strictly greater than $X = \frac{1}{d} \cdot \frac{\lambda^2 \cdot (k+1)}{(k+2)^2}$. The event that all critical robots sequentially target a robot further than λ has a positive probability of occurring, so it eventually does. Critical robots then each move of at least X towards the center of the original SEC. So, for $\varepsilon < X$, this contradicts the hypothesis, therefore the diameter of the SEC decreases towards zero.
Because of these properties, the diameter of the SEC has a non-zero probability of decreasing for every FSYNC cycle and cannot increase. So it *eventually* decreases towards zero with probability 1, and n-robot **Convergence** is eventually achieved against the $(n \cdot (n-1)-1)$-random adversary. Because of theorem 6.1, **Gathering** can also be achieved.

□

Theorem 6.3. *In FSYNC, it is impossible to solve* ***Gathering*** *deterministically against a* $(n \cdot (n-1))$*-random adversary.*

Proof. Indeed, against the $(n \cdot (n-1))$-random adversary, robots can be initially all blind (if they are located more than λ away from one another). Then, either they never move (preventing **Convergence**, and so, **Gathering**), or they move deterministically, but the adversary can provide them a symmetric coordinate system so they always move away from every other robot. □

Since the $(n \cdot (n-1)-1)$-random adversary is necessary and sufficient to solve deterministic **Gathering**, we now consider the more powerful k-enemy adversary.

Theorem 6.4. *In FSYNC,* ***Gathering*** *can be achieved against any k-enemy adversary with* $k \leq 2 \cdot n - 3$.

Proof. For each round, let us consider two cases :

- The $(2 \cdot n - 3)$-enemy adversary prevents at most one robot from receiving any message.
- The $(2 \cdot n - 3)$-enemy adversary prevents two or more robots from receiving any message.

The second case requires dropping $2 \cdot (n-1) > (2 \cdot n - 3)$ messages per round, so it is impossible. For the first case, we consider that robots run the move to center of gravity algorithm.
If all robots receive at least one message from a robot at a distance greater than λ, following a similar reasoning to the proof of theorem 6.2, all critical robots target points strictly inside the current SEC, and the diameter of the SEC eventually decreases towards zero. For the adversary to prevent **Convergence**, it should prevent this case from happening.
Therefore, to prevent **Convergence**, the adversary should ensure one robot receives no messages indefinitely. Let us assume this robot, r_1, sees no other robots and does not move. If r_1 is always strictly inside the SEC, then for each sequence where all critical robots target points strictly inside the SEC and, following the reasoning from the proof of theorem 6.2, the diameter of the SEC decreases towards zero. Thus, this is also a case which the adversary should prevent.
If r_1 is on the SEC itself, let us look at a robot r_2 such that r_2 is also on the SEC itself. If r_2 sees any robot inside the SEC, its target is strictly inside the SEC and, following the reasoning from the proof of theorem 6.2, once all critical robots except r_1 have been activated sequentially, the diameter of the SEC decreases towards zero, albeit with the center of the SEC drifting towards r_1. For r_2 not to move strictly inside the SEC, this requires r_2 not to see any robot at another location.

There are now two new cases:

1. r_2 is the only robot at its location.
2. there are M $(1 < M \leq n-1)$ robots at the same location as r_2.

In the first case, preventing r_2 from seeing another robot requires dropping another $n-1$ messages for a total $2 \cdot (n-1) > (2 \cdot n - 3)$ messages per round, so this is impossible.
In the second case, the adversary must prevent all present robots from moving. Otherwise, this second case *eventually* degrades into the first case, with robots *eventually* leaving the location of r_2.
This requires dropping a total of $M \cdot (n - M)$ messages. For $M = 1$ and $M = n-1$, the total of messages to be dropped is $n-1$. This number is trivially greater for any M between these two values. Therefore, it is impossible to prevent all these robots from moving, as it would require dropping a minimum of $n-1$ more messages (than the already $n-1$ messages dropped from r_1), so this is impossible for the $(2 \cdot n - 3)$-enemy adversary. The same reasoning holds if there are other robots at the same location as r_1. It is impossible to prevent r_2 from choosing a target strictly inside the SEC and the diameter of the SEC strictly decreases following the same reasoning as the proof of theorem 6.2.

Therefore, **Convergence** is possible for the $(2 \cdot n - 3)$-enemy adversary. Because of theorem 6.1, **Gathering** is also possible. □

Theorem 6.5. *In FSYNC, it is impossible to solve* ***Gathering*** *deterministically against a* $(2 \cdot n - 2)$*-enemy adversary.*

Proof. In the case of the $(2 \cdot n - 2)$-enemy adversary, it is possible to prevent a robot r_1 from both sending and receiving any visibility messages. Let us consider that the starting configuration is such that the starting location of r_1 is more than λ away from the initial positions and trajectories of the other robots. First, none of the remaining robots ever sees r_1. As a result, their **Gathering** location does not depend on the location of r_1. Also, since r_1 never sees other robots, it either

stays still (and thus never gather with the remaining of the robots since it remains invisible), or deterministically moves (in the latter case, its coordinate system can be set up so that it moves away from the other robots trajectories). Therefore **Gathering** is never achieved. □

Overall, a $(2 \cdot n - 3)$-enemy adversary is necessary and sufficient to solve deterministic **Gathering**. Note than neither adversaries required multiplicity detection to be defeated.

6.3 FSYNC Uniform Circle Formation

In this section, we demonstrate the difficulty of deterministic **Uniform Circle Formation** with uncertain visibility, under the FSYNC scheduler.
Uniform Circle Formation requires a network of n robots to *eventually* form a regular n-gon while ensuring no collision happens during the execution.
This necessary condition for **Uniform Circle Formation** is that a robot should never move to the same location as another robot. If this event, called a collision, occurs, then the pattern formation fails, as there is now no way to deterministically separate these robots ; see theorem 5.1.
We first consider that the pattern must *eventually always* be formed. We call this problem **Strong Uniform Circle Formation**.
Let us first observe that for networks of $n = 1$ and $n = 2$ robots, **Strong Uniform Circle Formation** is always trivially solved whenever robots start from distinct positions. In order to obtain a terminating algorithm, when a robot sees one or two robots in distinct positions, it must remain still.

Theorem 6.6. *In FSYNC, for a network of $n = 3$ robots, **Strong Uniform Circle Formation** is possible against the 4-random adversary, and impossible against the 5-random adversary.*

Proof. For $n = 3$ robots, the pattern corresponds to an equilateral triangle. If it is not already formed, there is at least one robot that must move in order to form it. In the case where this robot does not see one of the other two robots, since robots have no knowledge of n, this robot believes $n = 2$ and does not move.
In the case of the 5-random adversary, it is possible that a single visibility message is transmitted for the entire network at each round, so no robot ever has a complete vision of the network. So no robot ever moves.
In the case of the 4-random adversary, at least two messages are transmitted each round. This implies each robot can be in one of the three following situation:

- The robot receives no message.
- The robot receives a single message.
- The robot receives two messages.

In a FSYNC round, the probability that one particular robot r_1 receives two messages is strictly positive. We consider the algorithm by Flocchini *et al.* [44] for 3 robots. Then, a robot r_1 receiving two messages performs its movement as planned by Flocchini *et al.* properly. Once every robot has been in the third situation, all robots have moved according to Flocchini *et al.* once, and remained still the rest of the time. So, *eventually*, the pattern is formed with probability one. □

Theorem 6.7. *In FSYNC, for a network of $n \geq 4$ oblivious robots, no self-stabilizing algorithm can solve **Strong Uniform Circle Formation** against the 1-random adversary.*

Proof. Let us assume the regular n-gon is formed. If robot r is prevented by the adversary from seeing the entire network, it perceives the current network as the wrong pattern, *i.e.* a regular n-gon with a missing robot on one vertex instead of the assumed required regular $n-1$-gon. There is a positive probability that the dropped message forces the robot to move out of the pattern to attempt forming the $n-1$-gon instead. So the pattern cannot be kept infinitely and strong **Uniform Circle Formation** is impossible. □

Let us now consider **Weak Uniform Circle Formation**, for which we only require the pattern to be formed *eventually* at least once.

Theorem 6.8. *In FSYNC, a network of n robots using state-of-the-art algorithms by Mamino and Viglietta [67] for $n = 4$ and by Flocchini* et al. *[44] for $n \neq 4$ does not solve* ***Weak Uniform Circle Formation*** *against the* 1*-random adversary for $n \geq 4$.*

Proof. For $n = 4$ we show a configuration that has a non-zero probability of triggering a collision in figure 6.3. In this configuration, robots form a convex shape, and each robot r_k should move to the target T_k to form a square. However, this configuration is also such that $T_4 r_3 r_4$ form an equilateral triangle, and $|r_3 r_1| = |r_4 r_1|$. If the dropped message is such that r_1 does not see r_2, then r_1 mistakenly tries to create an equilateral triangle following the algorithm from Flocchini *et al.* [44], by moving to T_4. r_4 and r_1 then collide. This has a $\frac{1}{12}$ chance of happening and cannot be recovered from.

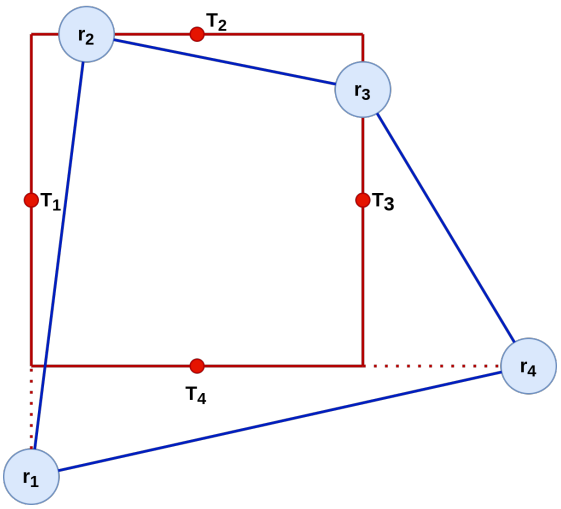


Figure 6.3 – Collision Inducing Configuration for $n = 4$.

For $n = 5$ we show a configuration that has a non-zero probability of triggering a collision in figure 6.4. In this configuration, using the algorithm from Flocchini *et al.* [44], robots try to move to the smallest enclosing circle, defined by its diameter $[r_1, r_5]$. However, this configuration is also such that, if r_1 is removed from the network, other robots form a convex, and T_4, the target of r_4 on the supporting square is located on the intersection of the SEC and the radius passing through r_2. So, by dropping the message from r_1 to r_4, r_4 executes the algorithm from Mamino and Viglietta [67] and moves to T_4 while r_2 executes the algorithm from Flocchini *et al.* [44], and also moves to T_4, hence a collision. This has a $\frac{1}{20}$ chance of happening and cannot be recovered from.

For $n \geq 6$ robots, a necessary step is to have all robots reach the smallest enclosing circle (SEC) without collision. According to the algorithm by Flocchini *et al.* [44], if multiple robots are located on the same half line starting from the center of the SEC, the furthest robot from the center moves to the SEC while the others perform steps to move to other locations of the SEC.

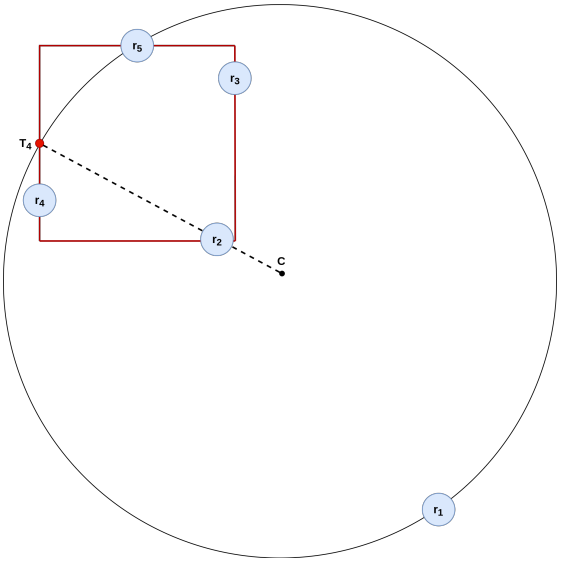


Figure 6.4 – Collision Inducing Configuration for $n = 5$.

In the case of a 1-random adversary, if the second furthest robot from the center does not see the furthest robot, they both move to the same point of the SEC, and a collision occurs. □

Overall, the 1-random adversary defeats state-of-the-art solutions [44, 67] for $n \geq 4$, even in FSYNC. However, it may be possible to design algorithms where all robots verify against collisions considering all possible missing visibility messages in the network (assuming their own view is complete). Since only one robot has an incomplete view of the system, it is the sole possible cause of collision, and other robots with complete views could predict the possibility of the collision and avoid it.

We conjecture that this is indeed the case.

Conjecture 6.1. *In FSYNC, for a network of $n \geq 4$ robots, it is possible to solve* ***Weak Uniform Circle Formation*** *against a 1-random adversary.*

Additionally, without making any assumption on the algorithms used, it seems that a fundamental limit of collision prevention would be the 2-random adversary. Against this adversary, there is a non-zero probability of two robots picking identical targets without being aware of the existence of the other robot. While it remains to be formally proven, in this case, it seems no strategy could prevent the collision.

Conjecture 6.2. *In FSYNC, for a network of $n \geq 4$ robots, it is impossible to solve* ***Weak Uniform Circle Formation*** *against a 2-random adversary.*

We also conjecture that the 1-enemy adversary precludes any solution for **Weak Uniform Circle Formation**.

Conjecture 6.3. *In FSYNC, for a network of $n \geq 3$ robots, no algorithm can solve* ***Weak Uniform Circle Formation*** *against the 1-enemy adversary.*

6.4 FSYNC Leader Election

Let us assume a network of n robots attempting to use their respective positions to single out one robot and elect it as a **LEADER**. Within the mobile robot literature, there are several proposed definitions of **Leader Election** specification. In the more general Distributed Computing

context, several specifications for **Leader Election** have also been used. Gupta *et al.* [52] proposed the notion of *agreement* that can be translated in the context of a robot network as "In any execution, *eventually* all non-faulty robots permanently know which robot is the **LEADER**". We use this condition to define **Strict Leader Election** and prove that it is trivially impossible to satisfy this specification against a 1-random adversary.

Definition 6.6 (**Strict Leader Election**)**.**
*A **Leader Election** process is* strict *if at any given time in the execution after **Leader Election**, every robot in the network knows which robot is the* **LEADER**.

Theorem 6.9. *In FSYNC, strict **Leader Election** is impossible against the* 1*-random adversary.*

Proof. Among the $n \cdot (n-1)$ visibility messages that are sent in each round, $n-1$ are sent by the **LEADER**. Therefore, for each round, there is a $\frac{1}{n}$ probability that a 1-random adversary drops a visibility message sent by the **LEADER**. Then, another robot is, at least temporarily, not aware of its existence. □

A less restrictive specification for **Leader Election** is defined by Attiya and Welch [5]. States are partitioned into **ELECTED** states and **NON-ELECTED** states. **Leader Election** is complete once one robot always remains in an **ELECTED** state, while all other robots remain in a **NON-ELECTED** state. We define **Soft Leader Election** following this principle:

Definition 6.7 (**Soft Leader Election**)**.**
Soft ***Leader Election*** *is achieved if, eventually, there is always exactly one robot in the* **LEADER** *state.*

Definition 6.8 (Self-Stabilizing, Non-Blocking **Leader Election**)**.**
Leader Election *is* self-stabilizing *and* non-blocking *[57] if for any configuration containing no* **LEADER** *robot, there exists at least one robot r_1 such that, after enough activations of only r_1, r_1 becomes* **LEADER**.

Theorem 6.10. *In FSYNC, no self-stabilizing, non-blocking algorithm can solve **Soft Leader Election** against the* 1*-random adversary.*

Proof. Let us assume a configuration such that a robot r_1 is the only robot in the **LEADER** state. Because the algorithm is both self-stabilizing and non-blocking, there exists another robot r_2 such that removing r_1 from the network and activating r_2 enough times makes r_2 eventually leave the **NON-ELECTED** state and reach the **LEADER** state.
Let us then consider the case where the visibility message from r_1 to r_2 is blocked by the 1-random adversary. This has a $\frac{1}{n \cdot (n-1)}$ probability of happening in each round.
Then, robot r_1 is the rightful **LEADER** of the network, and is fully aware of it. On the other hand, robot r_2 now mistakenly believes, that there is no **LEADER** in the network. Therefore, it starts the **Leader Election** process and, eventually reaches the **LEADER** state. At this point, two **LEADER** robots appear in this network. By repeating this sequence infinitely often, **Soft Leader Election** cannot be achieved.
Note that this true regardless of whether or not the algorithm is deterministic.

□

Note that, according to this definition, both the algorithm from Canepa and Gradinariu Potop-Butucaru [19] and our improved algorithm for four or more robots are both self-stabilizing and non-blocking. And, these algorithms indeed fail against the 1-random adversary when assuming the following configuration: r_1 is the single robot closest to the center of the smallest enclosing circle, and r_2 is the single robot second closest to the center of the smallest enclosing circle. If the visibility message from r_1 to r_2 is blocked, then the snapshot of r_2 mistakenly results in it being **LEADER**.

Theorem 6.11. *In SSYNC (hence, in FSYNC), it is possible to solve strict (hence, soft)* **Leader Election** *non-deterministically against a* 0*-random adversary.*

Proof. A 0-random adversary is tantamount to a full visibility adversary. Then the algorithms by Canepa and Gradinariu Potop-Butucaru [19] solve **Strict Leader Election** in SSYNC. □

For the more general case, without making any assumptions on the algorithm we give an upper bound for the vision adversary:

Theorem 6.12. *In FSYNC, for a network of n robots, no self-stabilizing algorithm can solve* **Soft Leader Election** *against the* $(n-1)$*-random adversary.*

Proof. The $(n-1)$-random adversary allows for an arbitrary long sequence of configurations to exist, with non-zero probability, in which no other robot in the network sees the **LEADER**. Because the algorithm is self-stabilizing, robots must attempt an election and, given enough activations, a second robot eventually reaches the **LEADER** state.

□

Conjecture 6.4. *In FSYNC, for a network of n robots, it is possible to solve self-stabilizing* **Soft Leader Election** *using* $\mathcal{LUMINOUS}$ *robots against the* $(n-2)$*-enemy adversary.*

6.5 FSYNC $\mathcal{LUMINOUS}$ Rendezvous

As explained in chapter I, a recent, fundamental development in mobile robots has been the addition of lights [31]. $\mathcal{LUMINOUS}$ robots are able to broadcast a light that can emit a single color among a fixed set. This color is then perceived in the snapshot of other robots which can perform computations according to their own color and the color of other robots. Because most $\mathcal{LUMINOUS}$ algorithms rely on this new capability to create what is functionally a distributed state machine, this is an obvious candidate for uncertain visibility.
While a comprehensive study of the effect of uncertain visibility on $\mathcal{LUMINOUS}$ algorithms is desirable, we focused on the specific problem of **Rendezvous**.
Under the $\mathcal{LUMINOUS}$ model, uncertain visibility allows the adversary to drop visibility message and block one robot from seeing another robot. This should also hide the color broadcast by this second robot.
Rendezvous is a special case of **Gathering** with exactly two robots. We recall that at least three solutions are known to work in the full visibility model under the ASYNC scheduler:

- Das4: The 4-color algorithm from Das *et al.* [31] described in figure 6.5.
- Viglietta3: The 3-color algorithm from Viglietta [84] described in figure 6.6.
- Heriban2: The 2-color algorithm from Heriban *et al.* [53] described in figure 6.7.

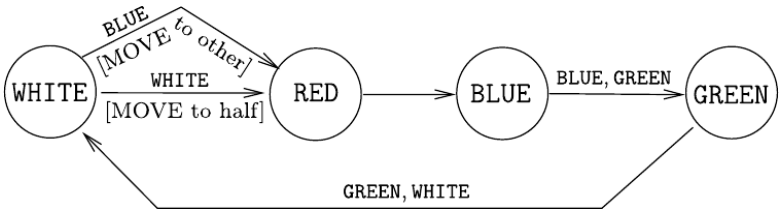


Figure 6.5 – Das4: 4-color Algorithm from Das *et al.* [31]

Each figure describes the algorithm through the state machine that is followed by each robot. The content of each circle matches the current color of the computing robot and the guard of each arrow matches the color of the other robot. In figure 6.6, movements are written as 0 for stay, 1 for move to other and $1/2$ for move to midpoint.

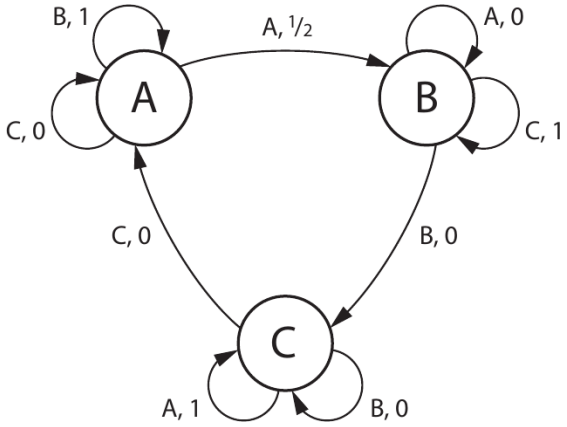


Figure 6.6 – Viglietta3: 3-color Algorithm from Viglietta [84]

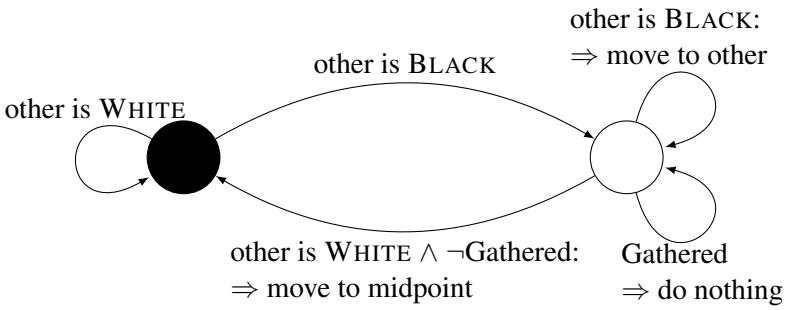


Figure 6.7 – Heriban2: 2-color algorithm from Heriban *et al.* [53]

Theorem 6.13. *In FSYNC, no deterministic* **Rendezvous** *algorithm can tolerate a* 2*-random adversary (hence, a* 2*-enemy adversary).*

Proof. With two robots, a 2-random adversary is tantamount to the $n \cdot (n-1)$-random adversaries, so the starting configuration could be that all robots are blind, and remain blind thereafter. If they do not move, **Rendezvous** is never achieved. If they move, they do so blindly so the adversary can set them away. □

Theorem 6.14. *In FSYNC, the* 1*-random adversary is the strongest possible against which aforementioned* **Rendezvous** *algorithms perform correctly.*

Proof. The 1-random adversary implies that, in each round, each robot has a $1/2$ probability of not seeing the other robot, *i.e.*, being blind. If a robot is blind, it is convinced that **Gathering** is achieved, and does not move, regardless of its color. Because the robot does not see a second robot, and all color transitions are decided according to the color (or position) of the second robot in every aforementioned algorithm, the robot cannot change its color. This is tantamount to the robot not being activated.
Note that there is an exception if the robot has color RED in the Das4 algorithm [31]. We also assume that if a robot is RED and sees no other robot, is does not change its color[1].
Therefore, when a robot is blind, it behaves as if it had not been activated. This implies that the 1-random adversary can be seen as a particular case of the SSYNC model where a single robot is left out from the set of executed robots (each such SSYNC execution is still fair with probability 1). As aforementioned algorithms are correct in ASYNC, they are also correct in SSYNC, hence in FSYNC against a 1-random adversary. Because no algorithm can solve deterministic

[1] Assuming the robot actually changes its color to BLUE does not impact the behavior of the algorithm, as it only adds a blind behavior for the robot.

Rendezvous against a 2-random adversary (theorem 6.13), 1-random is the strongest possible adversary against which the algorithms can succeed. □

Theorem 6.15. *In FSYNC, $\mathcal{LUMINOUS}$ **Rendezvous** using the aforementioned algorithms is impossible using the* 1*-enemy adversary.*

We actually prove a stronger theorem on this issue :

Theorem 6.16. *Any self-stabilizing algorithm that solves $\mathcal{LUMINOUS}$ **Rendezvous** in the SSYNC full visibility model fails against the FSYNC* 1*-enemy adversary.*

Proof. We reuse proposition 4.3 from Viglietta [84]:

"For any algorithm solving **Rendezvous** in rigid SSYNC, there exists two colors X and Y and a distance $d > 0$ such that any robot set to X that sees the other robot at distance d and set to Y does not move."

Note that this is true for all three aforementioned algorithms.

Now, in FSYNC, if the 1-enemy adversary drops the message from the robot having color X (say, robot r_1) to the robot having color Y (say, robot r_2), then r_2 never moves or changes its color, because it does not see a second robot. Because of this, robot r_1 never moves either, and **Rendezvous** is impossible. □

The statement of theorem 6.16 may seem contradictory with results obtained in the previous sections. Using the **Gathering** proof for n robots, we obtain that FSYNC **Gathering** is possible for two robots in 1-enemy with the classical (*i.e.*, not $\mathcal{LUMINOUS}$) $\mathcal{OBLOT}$ model. However, no $\mathcal{LUMINOUS}$ **Rendezvous** algorithm seems to exist in FSYNC against the same 1-enemy adversary. The difference is that in the second case, the algorithm *must* also be correct in SSYNC with full visibility, which is not required for oblivious FSYNC **Gathering**. This observation shows that $\mathcal{LUMINOUS}$ algorithms are *not* sufficient to obtain *universal* solutions (a **Gathering** solution is universal if it works both in the SSYNC full visibility model, and in the FSYNC 1-enemy model). As a result, uncertain visibility is orthogonal to asynchrony.

Overall, with respect to visibility adversaries, it seems **Leader Election** and **Uniform Circle Formation** are the most challenging tasks, as a 0-random adversary may be necessary and sufficient to solve the task in the more general settings, while **Rendezvous** and **Gathering** are more tolerant to uncertain visibility. This makes this model ideal for our goal of realistic robots, as it allows imperfection in sensors while still allowing multiple benchmark problems to be solved.

Chapter 7

Obstructed Visibility

7.1 Model and Problem Definition

In this chapter, let us consider the punctual opaque robot model [3]. Robots are dimensionless, and if three robots r_1, r_2 and r_3 are collinear so that r_2 is on the $[r_1, r_3]$ segment, then r_1 can see r_2 robot but cannot see r_3. Similarly, r_3 can see r_2 but cannot see r_1 and r_2 can see both r_1 and r_3. In other words, we consider unlimited vision range that can only be interrupted by another robot. Such an arrangement of three robots is called an obstruction and a configuration including obstructions is obstructed.

The most common way to manage obstructed robot networks is to convert the obstructed network into an unobstructed network so that further tasks can be performed under an unobstructed hypothesis.

To achieve this, robots must solved the **Mutual Visibility** problem, introduced by Di Luna *et al.* [66]: for a set of autonomous robots occupying distinct positions in the two dimensional plane, robots must coordinate their movement to form a configuration, in which no three robots are collinear. Other properties, such as avoiding collisions, or finishing within finite time can be required.

A solution for **Mutual Visibility** in the oblivious SSYNC model was published by Di Luna *et al.* [66]. Solutions have then been introduced in the ASYNC model by Bhagat *et al.* [13], but require one common axis and the knowledge of the total number of robots in the network.

After the introduction of the light model for mobile robots [30, 31, 76], solutions using lights have been proposed. Di Luna *et al.* [65] presented a solution using three colors in SSYNC with no additional assumption, and three colors in ASYNC assuming one common axis. This was later improved by Sharma *et al.* [81] to two colors, with similar assumptions.

However, in general, solving **Mutual Visibility** requires the robots to move.

In difficult environments where obstructed visibility is only a secondary hindrance for solving the main problem, this means that time and resources are expanded for a task that might not be mission critical. In those difficult environments, careful and predetermined motion planning might also be critical, and changing it might not be a viable option.

Furthermore, this is not resilient to the fact that some robots might not be able to move freely, either because of damaged motors or difficult terrain.

Because of this, in this section, we discuss a new problem, called the **Obstruction Detection** problem, which does not require robots to move and is defined as follows:

Definition 7.1 (Obstruction)**.**
For two robots r_1 and r_2 so that there are no other robots on the $[r_1, r_2]$ segment, if there is one or more robots other than r_1 and r_2 on the $[r_1, r_2)$ ray, we say that r_2 is **OBSTRUCTING** *r_1 or that r_2 is* **OBS** *for r_1. If there are no robots other than r_1 and r_2 on the $[r_1, r_2)$ ray, then robot r_2 is* **NOT OBSTRUCTING** *robot r_1, or r_2 is* **NOBS** *for r_1.*

Definition 7.2 (Visibility)**.**
The visibility of a robot r is the set of robots that only contains robots visible by r.

Definition 7.3 (Obstruction Detection)**.**
Let us consider a set of robots occupying distinct positions in the two dimensional plane. The ***Obstruction Detection*** *problem is solved if, for each robot r_i in this set, r_i eventually determines, without moving, for every robot r_j in its visibility, whether r_j is* **OBS** *for r_i or* **NOBS** *for r_i.*

In other words, the problem is solved if, eventually, for every robot r in the network, every robot in the visibility of r is classified as either **OBS** or **NOBS** with absolute certainty.
Robots considered in this section are endowed with lights as defined in chapter I

This problem sounds fairly trivial at first since it requires robots to only transmit a single bit of information. However, this single bit may need to only be transmitted to a single robot in the visibility. So the problem is, in a sense, similar to directional transmission.

7.2 Simplifying the Problem: Line Theorem

Let us first simplify the problem by considering two cases: whether or not robots form a single line.

Theorem 7.1. *Unless all robots in the network are collinear, for any line $\mathcal{L}$ formed by robots positions in the network, there exists at least one robot r such that all robots on $\mathcal{L}$ are in the visibility of r.*

Proof. If not all robots are collinear, then for any line $\mathcal{L}$ formed by robot positions, there is at least one robot outside $\mathcal{L}$. Among those robots, the robot which is the closest to $\mathcal{L}$ without being part of it can see the entire line. Let us call this robot r.
By contradiction, let us assume that this is not true. This means that there is a robot $r_{\mathcal{L}}$ on $\mathcal{L}$ that r cannot see. Since the only way that a robot can be hidden from another robot is to have a third robot between the two, this implies that there is a robot between r and $r_{\mathcal{L}}$. Therefore, r is not the closest robot to L. This is a contradiction. □

On the other hand, if all robots are collinear, then there is a unique line $\mathcal{L}$ where all robots are located. Therefore, there are no robots outside $\mathcal{L}$ and, if the size of the network is greater than 3, no robot can see both robots at each end of the line.

Definition 7.4 (Proximity)**.**
Let s be an ordered sequence of robots starting at r_1 and ending at r_2 such that every robot in s can see both the previous and following robot in the sequence. The proximity of two robots r_1 and r_2 is defined as the number of robots (excluding r_1 and r_2) in the smallest such sequence, starting from r_1 and ending with r_2.

As an example, if r_1 can see r_2, the proximity of r_1 and r_2 is zero. If a robot r_3 is obstructing r_1 and r_2, such that r_1 can see r_3 and r_3 can see r_2, but r_1 cannot see r_2, then the proximity of r_1 and r_2 is one, and so on.
Using theorem 7.1, we can see that either:

- The network is a single line. Therefore the proximity between two robots can be anywhere between 0 and the size of the network minus two.
- The network is not a line. Then, since any two robots can be seen at the same time by a third robot, this means that no proximity greater than one can exist.

7.3 Obstruction Detection for the Line Configuration

We managed to divide the configurations into two distinct subsets of configurations, that can be handled by our algorithm in different manners.

Theorem 7.2. *Every robot in the network can determine using a single snapshot whether the network is a single line.*

Proof. Any robot can determine if the network is a line during its compute phase if one of the three following conditions are met :

1. The snapshot contains exactly one robot, including the computing robot itself.
2. The snapshot contains exactly two robot, including the computing robot itself.
3. The snapshot contains exactly three robots, including the computing robot itself, and all three robots are collinear.

If the snapshot contains more than three distinct robots, then the network cannot be a single line, as four or more robots in a line would require at least one obstruction and no robot would be able to see all four. If the snapshot contains only three robots that are not collinear, then, by definition, this cannot be a single line. □

In the first case, the problem is trivially solved: there can be no obstructions. We also notice that in the second case, the computing robot is **NOBS** for the visible robot. Finally, the computing robot is **OBS** for the two visible robots in the third case. Therefore, we use the following subroutine when the observed network is a line, called the line solving subroutine:

Theorem 7.3. *Let us define the following two-color subroutine:*

- *In the case of two visible robots (including myself), I am* **NOBS** *and I change my color to* WHITE.
- *In the case of three visible robots (including myself), and the three robots are collinear, I am* **OBS** *for those two robots, and I change my color to* BLACK.
- *If a robot broadcasts* WHITE*, I classify it as* **NOBS**.
- *If a robot broadcasts* BLACK*, I classify it as* **OBS**.

This subroutine solves ***Obstruction Detection*** *for any line configuration.*

Proof. Since the network is a line, a robot can either be **NOBS** for everyone, or **OBS** for every visible robot. No robot can be both **NOBS** and **OBS** for different robots at the same time. So broadcasting the **NOBS/OBS** status is sufficient for every robot to eventually solve the **Obstruction Detection** Problem. □

7.4 Non-Line Obstruction Detection: a Simple Approach

Let us now construct a simple deterministic approach for the **Obstruction Detection** problem in ASYNC, based on the following principles:

- The visibility of a robot r_1, for k visible robots is defined as two vectors $\vec{\mathbf{p}}$ and $\vec{\mathbf{c}}$ that contain respectively the positions and colors for each robot from 1 to k, 1 being the robot itself.

- The color of r_1 should only be a function of the position vector $(p_1, p_2, ..., p_k)$.
- To solve the problem of **Obstruction Detection**, the robot should use the two vectors as inputs.

By restricting the robot's color to be solely a function of visible positions, we ensure that the color of a robot does not change after having executed a full cycle. This allows the algorithm to be both self-stabilizing and silent for any fair scheduler. The most obvious candidate algorithm following these principles is the subroutine used to solve the line case.

Theorem 7.4. *The subroutine used for the single-line network does not solve the **Obstruction Detection** Problem in the case of a non-line network.*

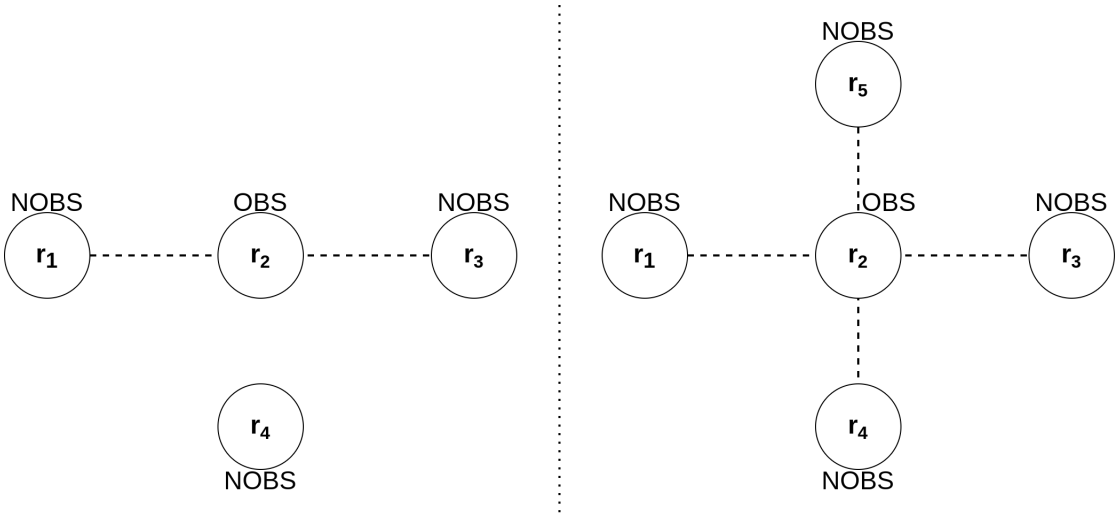


Figure 7.1 – Example: Insufficient First Fit

Proof. In the example presented in figure 7.1, in the left example, r_2 prevents r_1 from seeing r_3, and r_3 from seeing r_1. Therefore, r_2 is **OBS** for r_1 and r_3. However, r_2 is **NOBS** for r_4, as it does not prevent it from seeing any robot. Since they cause no obstructions at all, r_1, r_3 and r_4 are obviously **NOBS** for everyone.
So the fact that r_2 is broadcasting **OBS** allows r_1 and r_3 to know for sure that r_2 is **OBS** for them. This is because they do not see r_2 as part of any obstruction, so the fact that r_2 is **OBS** for someone implies that r_2 is necessarily **OBS** for them.
On the other hand, this information is insufficient for r_4. Indeed, r_4 sees r_2 as part of an obstruction, so r_4 does not know whether this is the only obstruction r_2 is a part of, or if r_2 is also hiding another robot from r_4. This is demonstrated in the right example where the visibility of r_4 has not changed, but, because of r_5, r_2 is now **OBS** for r_4. □

Since broadcasting the obstruction status is insufficient, a sensible approach would be to add more bits of information in order to solve the problem. One possible way is for each visible robot to broadcast some information about the number of obstructions it is part of, and then for each robot to check the consistency of this information with its position vector.

To achieve this, one possible approach would be to broadcast the parity of the number of obstructions a robot is causing.

If robot r_1 sees robot r_2 broadcasting that it is **OBS** and part of an odd number of obstructions, it can then check whether it can see robot r_2 being part of an odd number of obstructions. If the information is consistent, then r_1 determines r_2 is actually **NOBS** for r_1. If r_1 notices a discrepancy, however, then r_1 determines r_2 is part of an obstruction it cannot see. Therefore r_2

is **OBS** for r_1.

Figure 7.2 shows the same configuration where this approach solves the problem.

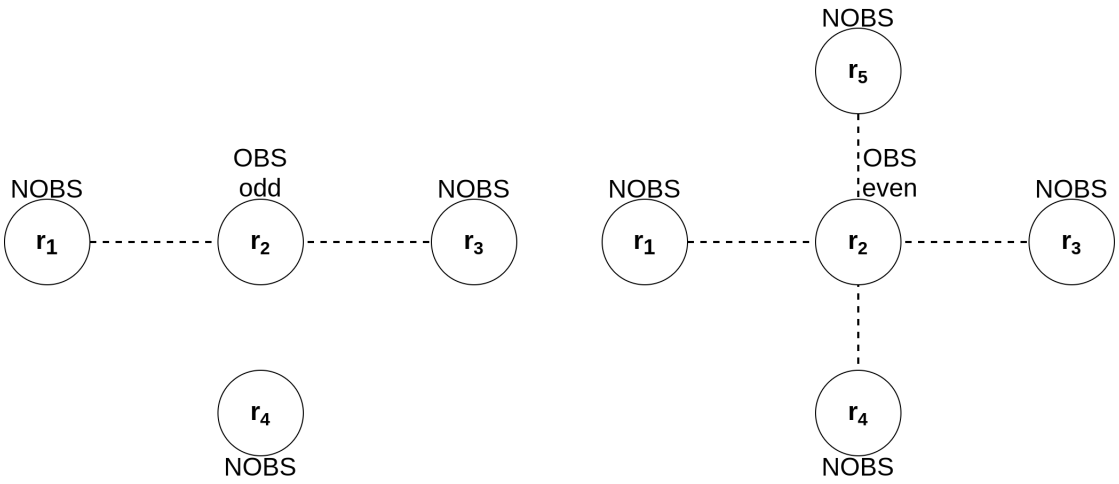


Figure 7.2 – Example: Sufficient Second Bit ?

Note that it would be possible to consider zero obstructions to be an even number of obstructions and remove the first bit entirely.

This second bit is, unfortunately, also insufficient.

We now prove why the current logic behind the algorithm cannot work.

Theorem 7.5 (Properly using the snapshot)**.** *Using only the position vector, and the color of a robot r is insufficient to determine whether r is* **OBS**.

Proof. We now prove that the algorithms we previously used (that uniquely deduce the obstruction status of an observed robot from its color) are fundamentally flawed: Let us assume an infinite number of available colors in a way that any two different snapshots imply two different colors. Note that as a consequence of the second design principle, if two snapshots appear identical when looking only at positions, then they must imply the same color.

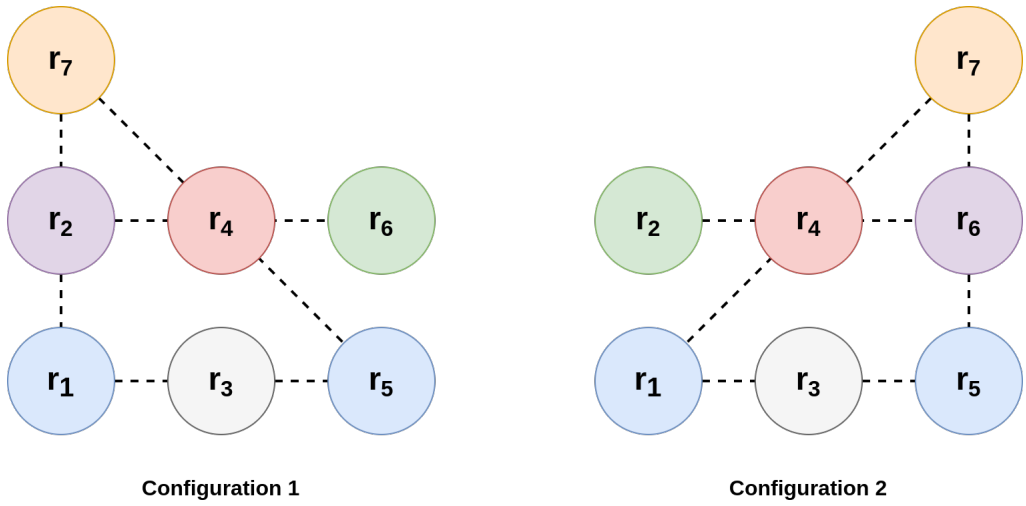


Figure 7.3 – Counter-Example to Prove Theorem 7.5

In the first configuration of figure 7.3, robots r_1 and r_5 must have identical colors since their position vectors are identical. On the other hand, every other robot has a unique position vector,

so they may all broadcast a different color.
The second configuration is created by changing the position of robot r_7. Note that, in that case, the position vectors of r_1 and r_5 have not changed, since they cannot see r_7. Also, because robots do not use chirality, the position vectors of r_3, r_4 and r_7 cannot be distinguished, and the position vectors of r_2 and r_6 have been swapped. This implies that the only difference, in terms of color, is r_2 and r_6 swapping their colors.
However, in the first configuration, r_4 is **OBS** for r_5 and **NOBS** for r_1, and **OBS** for r_1 and **NOBS** for r_5 in the second configuration.
Because neither the position snapshots of r_1 and r_5 nor the color of r_4 have changed between the two configurations, it is not possible to solve the Obstruction Detection problem if the algorithm only uses a robot's color to determine its obstruction status. □

Because of this, the algorithm should explicitly use the entire color vector. It should be noted that, given an infinite (similar to $\mathbb{R}^{2n}$) number of colors, robots could broadcast their entire position vector, which would trivially solve the Obstruction Detection problem.

We have currently been unable to find and prove an algorithm following the design principles that reliably solves Obstruction Detection using a finite number of colors.

7.5 Non-Line Obstruction Detection: Using a Token

One can notice that if one robot is differentiated from the others, for instance, if it carries a token, then the problem can be trivially solved for this particular robot. Each robot that can see the token holder would only need to broadcast one bit of information: *"Am I* **OBS** *for the token holder ?"* Solving the problem for all robots would then only require that, eventually, every robot has held the token at least once.

We previously defined robots as having only a finite number of colors. However, to solve the problem in such a way, this require robots to either use an internal memory of n bits, with n the size of the network, so that a robot can store the obstruction information after passing the token.

Once a network contains a single token holder, if we chose to not transmit the token and have the robot keep it, we could effectively consider that this robot is now the **LEADER** of the network. As such, such a token creation algorithm can also be used to elect a **LEADER**.

7.5.1 Difficulty of Creating a Token with Obstructed Visibility

For a network with no obstructions, it is easy to detect whether a token exists in the entire network, and thus decide to create one. This task becomes more complex in the case of a network with obstructed visibility, as some robots may not know whether a token already exists.

The more intuitive approach is to require token creation to be managed in such a way that each robot sees exactly one token at any given point in time. This is, however, not possible for arbitrary configurations. An easy counter-example is shown in figure 7.4. For this particular configuration, it is not possible to assign tokens in a way that each robot sees exactly one token. Another approach is to have exactly one token for the entire network.
We use theorem 7.1 again, as we know that every two robots in the network are within a proximity of one or zero. This implies that if a token exists, every robot in the network either sees the token, or sees a robot that sees the token. Remember that this is only true in a network configuration that does not consist of a single line.

For this observation, we can define a three layers network architecture: The token holder is the first layer, the robots that see the token holder, and transmit their obstruction status to the token holder as the second layer, and a third layer of robots that only see second and third layer robots.

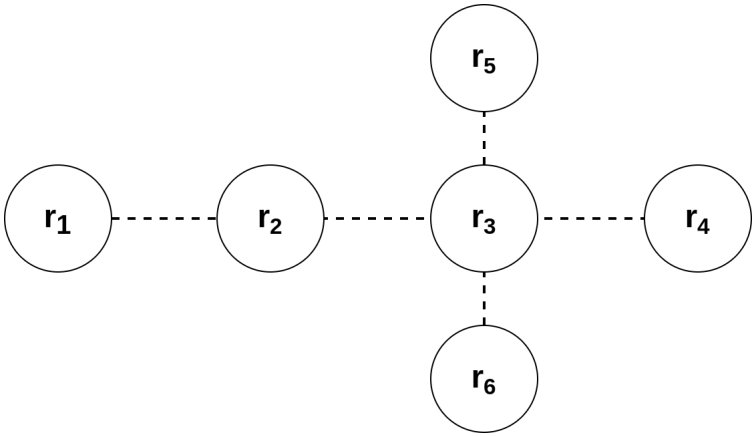


Figure 7.4 – Counter Example
No configuration where each robot sees exactly one token can exist.

To build a procedure despite incomplete visibility, we observe that there are two cases of conflict to take into account: two tokens should not see one another, and second layer robots should only see one token holder.

7.5.2 Algorithm Architecture

We first define the global architecture the network needs to self-stabilize to. We first define three layers: **THOLDER** (Token holder), **PRIMARY** and **DEFAULT**. These layers obey the following principles:

- If a robot is in the **THOLDER** layer, its visibility should only include robots in the **PRIMARY** layer. If it sees another **THOLDER**, it resets to the **DEFAULT** layer.
- Once a **THOLDER** robot has received its obstruction information, it transmits its token to a randomly chosen **PRIMARY** robot and goes to the **PRIMARY** layer.
- If a robot is in the **PRIMARY** layer, its visibility should include exactly one **THOLDER** robot, otherwise it goes to the **DEFAULT** layer.
- If a robot is in the **DEFAULT** layer and sees no **PRIMARY**, then it should attempt to create a token.

However, because of conflicts, some robots might lose the token they are trying to create and become **PRIMARY**. Because of that, we also define layers **SECONDARY** and **CANDIDATE** that are transitions layers between **DEFAULT** and **PRIMARY/THOLDER**, respectively. This is summed up in figure 7.5.
It should be noted that deterministically creating a single token is tantamount to electing a **LEADER**. Because of possible symmetries in the network, no deterministic algorithm can solve this problem [37, 47]. Therefore, robots are endowed with the ability to perform Bernoulli trials: a robot can choose between two different actions, such as target locations, or colors, with a winning option being given a certain probability, and the losing action the complementary probability. This ability is also necessary to chose which robot the token should be transmitted to.

7.5.3 A Possible Solution

To solve Obstruction Detection in ASYNC, we require two separate algorithms. First an Obstructed Token Creation algorithm, which ensure that eventually, a single **THOLDER** exists and

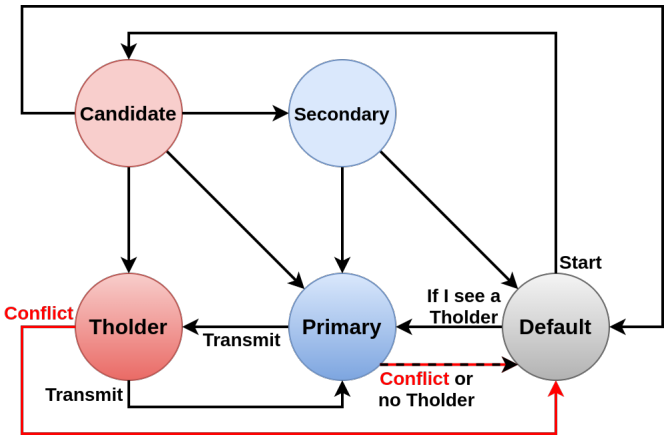


Figure 7.5 – Architecture of the Algorithm
Red indicates cases that should only happen for self-stabilization.

the network follows the aforementioned structure. Second, an Obstructed Token Transmission algorithm which ensures **PRIMARY** robots transmit their obstruction status to the **THOLDER**, and that the token is transmitted from the **THOLDER** to a randomly chosen **PRIMARY**.

So, when performing **Obstruction Detection**, robots do the following:

- Check whether or not the network is a line.
- If it is a line, perform the 2 color Line Solving algorithm.
- If it is not a line, performs the Obstructed Token Creation once and Obstructed Token Transmission algorithm indefinitely.

This ensures that the **Obstruction Detection** problem is eventually solved for the whole network.

7.5.4 Gathering Information and Transmitting the Token

Let us first consider the configuration in which the token has been successfully created and all the network follows the proper structure. This implies a single **THOLDER** robot, its visibility only including **PRIMARY** robots, and all other robots in the **DEFAULT** state.

The **THOLDER** robot is in the TOK color, while **PRIMARY** robots are in the RTT (Ready To Transmit) color. The next step is to send the payload. Once this is done, we just need to send the token to a **PRIMARY** robot and start again. We use a Bernoulli trial for robots to decide whether or not they want to receive the token, and send it once only one robot is willing to receive it. Our transmission algorithm is described in figure 7.7.

To keep figure 7.7 readable, some abbreviations similar to the ones used in section 5.4 are defined:

- "X and Y" is true if both X and Y are true.
- "X or Y" is true if either X or Y is true.
- "$\nexists X$" is true if no robot of color X exist in the snapshot.
- "$\nexists(X,Y)$" is true if no robot of color X or Y exist in the snapshot.
- "$\exists X$" is true if at least one robot of color X exists in the snapshot.
- "$\exists(X,Y)$" is true if at least one robot of color X or Y exists in the snapshot.
- "$\forall X$" is true if the snapshot only contains robots of color X.
- "$\forall(X,Y)$" is true if the snapshot contains no robots of color other than X and Y.

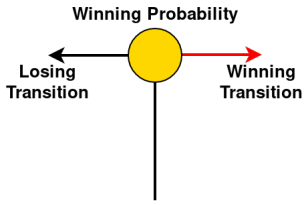


Figure 7.6 – Symbol for the Bernoulli Trial

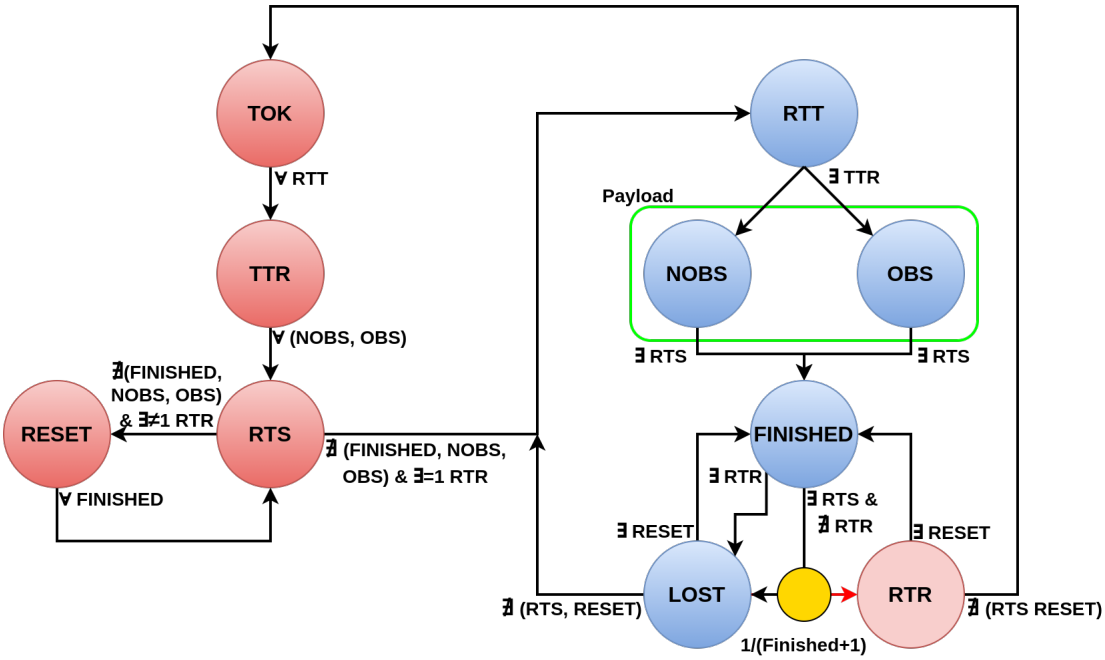


Figure 7.7 – Obstructed Token Transmission Algorithm

Algorithm 7.1 Obstructed Token Transmission Algorithm: **THOLDER**

```
if me.color = TOK
    if ∀ other.color = RTT
        me.color ⇐ TTR
else if me.color = TTR
    if ∀ other.color = {NOBS,OBS}
        me.color ⇐ RTS
else if me.color = RTS
    if ∄ other.color = {FINISHED,NOBS,OBS}
        if ∃! other.color = RTR
            me.color ⇐ RTT
        else
            me.color ⇐ RESET
else if me.color = RESET
    if ∀ other.color = FINISHED
        me.color ⇐ RTS
```

Algorithm 7.2 Obstructed Token Transmission Algorithm: **PRIMARY**

```
if me.color = RTT
    if ∃ other.color = TTR
        if I see a robot other than the THOLDER on the (me THOLDER) line
            me.color ⇐ OBS
        else
            me.color ⇐ NOBS
else if me.color = NOBS or me.color = OBS
    if ∃ other.color = RTS
        me.color ⇐ FINISHED
else if me.color = FINISHED
    if ∃ other.color = RTS and ∄ other.color = RTR
        Perform a Bernoulli trial with a probability of winning of p = 1/(FINISHED + 1)
        if Trial won
            me.color ⇐ RTR
        else
            me.color ⇐ LOST
    else if ∃ other.color = RTR
        me.color ⇐ LOST
else if me.color = LOST
    if ∃ other.color = RESET
        me.color ⇐ FINISHED
    else if ∄ other.color = {RESET,RTS}
        me.color ⇐ RTT
else if me.color = RTR
    if ∃ other.color = RESET
        me.color ⇐ FINISHED
    else if ∄ other.color = {RESET,RTS}
        me.color ⇐ TOK
```

The algorithm functions as follows:

1. One robot is in the TOK color and all visible robots in the RTT color.

2. The TOK robot switches to TTR (Token TRansmit).
3. All robots now broadcast their payload of one bit (NOBS or OBS).
4. Once every **PRIMARY** has broadcast its payload, the token holder switches to RTS (Ready To Send).
5. **PRIMARY** robots move to FINISHED and then perform a Bernoulli trial.
   (a) They can become RTR (Ready To Receive).
   (b) Otherwise, They become LOST.
   (c) If a FINISHED robot is activated after a RTR has appeared, it directly becomes LOST.
6. If there is more than one RTR, or no RTR at all, the RTS robot, which can see all **PRIMARY** robots, by definition, switches to RESET until every **PRIMARY** robot is back in FINISHED and cannot change color.
7. The RESET robot switches back to RTS.
8. If only one RTR has appeared, then RTS switches to RTT and the LOST switch to RTT and RTB to TOK.
9. This leads to **DEFAULT** robot seeing a **THOLDER**. They must then switch to RTT.
10. This also leads to **PRIMARY** robots seeing no **THOLDER**, they must then go to **DEFAULT**
11. Once a TOK exists and can only see RTT, the transmission process can start again.

Note that the winning probability is $1/(\text{FINISHED}+1)$, and not $1/\text{FINISHED}$. Indeed, if two **PRIMARY** robots were being obstructed by either the **THOLDER** and another **PRIMARY**, and all other robots would have failed the trial, then both robots would have a winning probability of 1 and could not be sorted. Choosing FINISHED+1 ensures that the winning probability is always smaller than 1.

As a sidenote, in the algorithm described in figure 7.7, the size of the payload is only one bit, which is sufficient for our problem. However, it can be easily extended by adding more two-choices steps to a given algorithm. For example, sending a 10 bits message would require adding 18 colors to the **PRIMARY** part and 9 to the **THOLDER** part.

Conjecture 7.1. *The Obstructed Token Transmission algorithm described in figure 7.7 functions under the ASYNC scheduler and allows for the* **THOLDER** *to know the obstruction status of all visible robots. Eventually, a* **PRIMARY** *robot is chosen to be the next* **THOLDER**. *The adversary scheduler cannot reliably decide which* **PRIMARY** *robot is chosen to be the next* **THOLDER**

Proof. For reasons explained in section 7.5.5, we only prove the OTT algorithm for the FSYNC scheduler.
Let us first assume this algorithm is started following a successful Token Creation. So the network follows the structure described in figure 7.5: there is a single **THOLDER** robot, in the TOK color, which can only see **PRIMARY** robots in the RTT color. All other robots are stuck in the **DEFAULT** state.

1. After the first cycle, the **THOLDER** robot is TOK and changes color to TTR, as it saw RTT robots. **PRIMARY** robots are RTT and cannot change color as they do not see a TTR robot.

2. After the second cycle, the **THOLDER** robot is TTR and cannot change color, as it sees RTT robots. **PRIMARY** robots are RTT and change to either OBS or NOBS depending on their obstruction status.
3. After the third cycle, the **THOLDER** robot is TTR and changes color to RTS, as it sees only OBS and NOBS robots. **PRIMARY** robots are OBS or NOBS and cannot change color as they see no RTS robot. At this point, the **THOLDER** has found all its obstructions.
4. After the fourth cycle, the **THOLDER** robot is RTS and cannot change color, as it sees OBS and NOBS robots. **PRIMARY** robots are OBS or NOBS and change to FINISHED as they see a RTS robot.
5. After the fifth cycle, the **THOLDER** robot is RTS and cannot change color, as it sees FINISHED robots. **PRIMARY** robots are FINISHED and change to either LOST or RTR depending on the Bernoulli trial.
6. After the sixth cycle, if there were no RTR robots:
   - The **THOLDER** robot is RTS and changes color to RESET. **PRIMARY** robots are LOST and cannot change color.
   - After the seventh cycle, the **THOLDER** is RESET and cannot change color. **PRIMARY** robots are LOST and change color to FINISHED.
   - After the eighth cycle, the **THOLDER** is RESET and changes color to RTS. **PRIMARY** robots are FINISHED and cannot change color.
   - Robots are now back to the configuration at the end of the fourth cycle.
7. After the sixth cycle, if there were multiple RTR robots:
   - The **THOLDER** robot is RTS and changes color to RESET. **PRIMARY** robots are LOST or RTR and cannot change color.
   - After the seventh cycle, the **THOLDER** is RESET and cannot change color. **PRIMARY** robots are LOST or RTR and change color to FINISHED.
   - After the eighth cycle, the **THOLDER** is RESET and changes color to RTS. **PRIMARY** robots are FINISHED and cannot change color.
   - Robots are now back to the configuration at the end of the fourth cycle.
8. After the sixth cycle, if there was a single RTR robot:
   - The **THOLDER** robot is RTS and changes color to RTT. It is now a **PRIMARY** robot. **PRIMARY** robots are LOST or RTR and cannot change color.
   - After the seventh cycle, LOST robots change color to RTT and the RTR robot changes color to TOK. It is now a **THOLDER** robot.
   - After the eighth cycle, **DEFAULT** robots which can see the **THOLDER** change color to RTT. They are now **PRIMARY** robots.
   - Robots are now back to the a configuration similar to before the first cycle.

During the third cycle, the **THOLDER** robot becomes aware of all visible obstructions.
There is a positive probability of a single RTR robot appearing after the fifth cycle. Since other cases reset the network, there is eventually a single RTR robot, and the token transmission is successful.
Since the FSYNC scheduler cannot chose which robots change color to RTR, it cannot reliably chose the next **THOLDER**.
So conjecture 7.1 is true for this scheduler.

□

7.5.5 The Issue of Proving Obstructed Algorithms

While we described what the Obstructed Token Creation algorithm *should* do, we did not describe a proven algorithm.

During our work on **Obstruction Detection**, we quickly noted that any proof in this setting is orders of magnitude harder than proofs in the regular transparent setting.

The key issue surrounding the obstructed visibility model is the fact that two different robots may have two different snapshots. In fact, in a line configuration, two robots may share no robots in their snapshot. This implies that any proof must be done considering not a set of configurations for the network, but a set of snapshots for each robot in the network. The size of this set increases even more when proving self-stabilizing algorithms.

As we have shown for the Obstructed Token Transmission algorithm it may still be possible to prove complex algorithms in the obstructed FSYNC setting. However, this issue of different snapshots is magnified by stronger schedulers, such as SSYNC or ASYNC, as snapshots are not only different, but may now evolve over time differently, or become outdated.

Overall, we believe a key reason why **Mutual Visibility** has been relied on for using the obstructed visibility model is simply the sheer difficulty of proving any other algorithm in this setting. **Mutual Visibility** provides a global variable – the number of obstructions – that can be proven to be eventually decreasing.

Problems which do not provide such variables, such as **Obstruction Detection**, require different strategies for proving.

As a matter of fact, to the best of our knowledge, the proven algorithm which uses the most colors is a 7-color **Mutual Visibility** algorithm [14], while our Obstructed Token Transmission already uses 10 colors. Most attempts made to build a ASYNC Obstructed Token Creation algorithm resulted in 10 additional colors, which is practically not provable in an already difficult setting.

So, we believe that it is not currently possible to reliably prove such complex algorithms in the ASYNC obstructed visibility setting. It seems necessary to either develop new techniques, or use weaker proofs, such as the simulations we introduce in part III.

7.5.6 Sidenote: Ensuring Token Unicity for a Line

All previous results rely on the fact that the network should not be a single line and on theorem 7.1. However, it is interesting to look at how such algorithms would behave in the case of a network with a single line topology. In this setting, since the hypothesis that no two robots can have a proximity greater than one does not exist any longer, the key concept of a three layer structure also falls apart.

Indeed, the proximity can now be as large as $(n-2)$, with n the size of the network. We now show an algorithm designed to guarantees that, if the number of tokens in the network is different from one, at least one robot can detect the anomaly.

We build algorithm 7.3 using the following principles:

- Robots use colors 1, 2 and 3 to create a direction using a 1,2,3,1,2,3 pattern.
- Colors are ordered such that $1 > 2$, $2 > 3$ and $3 > 1$, with greater colors pointing towards the token.
- The token should only see the color 1.
- Robots on the edge of the line should ensure the color they see is greater than their own.
- Each robot should ensure the colors it sees are properly ordered.

- In case of violation, a reset is triggered.

Algorithm 7.3 ASYNC Token Anomaly Detection

```
if me.color is visible or two identical colors are visible
    Reset
if me.color = TOKEN and 2 or 3 is visible
    Reset
if I see a single robot
    if me.color = 1 and 2 is visible
        Reset
    if me.color = 2 and 3 is visible
        Reset
    if me.color = 3 and 1 is visible
        Reset
```

An example of the pattern is shown in figure 7.8.

Figure 7.8 – Properly Ordered Pattern Pointing Towards the Token

Theorem 7.6. *If the line configuration contains either no token, or more than one token, then the reset is eventually triggered.*

Proof. Let us first consider the case when there is no token in the network.
If colors are not properly ordered, *i.e.* in a 1,2,3,1,2,3 pattern, at least one robot can detect the error in the pattern and trigger the reset.
If colors are properly ordered, then one of the two edge robots sees a color that is smaller than its own, which should not happen as colors should increase towards the token, and triggers the reset. An example of this behavior is shown in figure 7.9.

Figure 7.9 – The Right Edge Robot Can Detect an Anomaly

Let us now consider the case where two or more tokens exist.
Let us consider two tokens T_1 and T_2 so that there are no other tokens between T_1 and T_2. Since colors decrease both from T_1 and T_2, there is a local minimum between the two tokens where colors break the ordered pattern. This discrepancy is visible by at least one robot which triggers the reset. An example of this behavior is shown in figure 7.10.

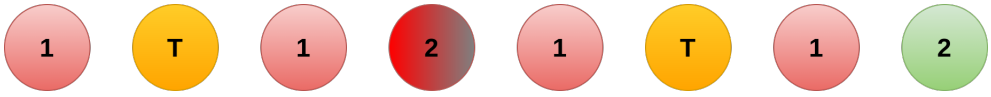


Figure 7.10 – One Robot Can Detect an Anomaly

□

Conclusion: Unreliable Vision

In this chapter, we studied two different, more realistic models of vision for mobile robots.

First, we introduced a new model, called uncertain visibility. This model is based on the idea of robots being able to 'miss' a robot during their snapshot. This behavior is either random, or governed by an adversary, and can be seen as a generalization of limited visibility. We showed tight bounds for several problems under this new model and the FSYNC scheduler. We also proved that uncertain visibility is orthogonal to synchrony, as $\mathcal{LUMINOUS}$ algorithms capable of solving two-robot **Gathering** in SSYNC are inherently unable to solve two-robot **Gathering** against adversarial visibility.

More specifically, for specifications that must maintain a global invariant (such as leader election or uniform circle formation), state-of-the-art solutions fail even against the weakest non-trivial adversary (1-random). For specifications with eventual safety properties (such as gathering and rendezvous), the results are contrasted: when the algorithm allows all robots to move at every round, strong adversaries can be handled, when there exists a synchronization mechanism (as in rendezvous algorithms for luminous robots), even the weakest non-probabilistic adversary (1-enemy) precludes any deterministic solution.

In fact, in this context, the 1-enemy adversary can be seen as a crash fault for the blind robot r, as r behaves identically as if it is never activated. For this problem, it is fairly intuitive to realize that luminous algorithms are more sensitive to crash faults than oblivious algorithms. Now, wait-free algorithms [16, 17] are able to withstand the crash of $N-1$ robots, still solving the problem for the subset of correct robots, so they should be able to handle stronger uncertain visibility adversaries. However, to our knowledge, there exists no wait-free rendezvous algorithm in the SSYNC full visibility model (only the case of $N > 2$ is handled by previous work). As highlighted by our results, if such a solution exists, it may not use lights.

The second model, obstructed visibility, is not a new model, but all research on this model seemed limited to the single problem of **Mutual Visibility**. We devised a new problem, named **Obstruction Detection**, and attempted to solve it. First, through a single cycle approach, which would allow for a trivial proof under the ASYNC scheduler. We proved a criteria that any algorithm must follow to solve **Obstruction Detection** in this single-cycle specification, but were not able to design such an algorithm.

We then attempted to solve the problem through a token creation and transmission method. While our approach initially appeared sound, the complexity of both the model and the candidate algorithms is such that current proving techniques may be unusable in practice. In the end, neither approaches yielded definitive results, as the obstructed visibility models massively increases the difficulty of designing and proving algorithms.

Part III

Real World Performance

Chapter 8

Monte-Carlo Simulation of Mobile Robots

8.1 Motivation

As we have seen, several attempts have been made to make the $\mathcal{OBLOT}$ model more realistic, *e.g.* by limiting the range of sensors through the limited visibility model [3, 50, 51], by allowing the sensors to miss other robots [55], by using inaccurate compasses [25, 50, 51, 68, 85] (so, the orientation of the robots is not arbitrary), or by discarding the hypothesis that robots are transparent [66].

However, most attempts are hindered by increased complexity due to manually proving the algorithms in those more complex settings. For instance, to the best of our knowledge, the consequences of error-prone vision have only been studied through the very simple problems of **Gathering** and **Convergence** [25, 50, 51, 68, 85].

To allow more complex problems to be studied under more realistic settings, it appears necessary to favor an automated approach. We have already discussed these approaches in chapter 3.2.
However, model-checking based approach for continuous **Gathering** [33, 34] requires proving several abstractions beforehand to simplify the continuous setting to a binary gathered / non-gathered, and is thus currently only applicable for **Rendezvous** algorithms.

At this moment, using formal methods for problems such as **Obstruction Detection** does not seem a feasible solution.

We investigate another approach: since our goal is to bridge the gap between theoretical mobile robots, and actual robotics, we move one step towards robotics and use a very common tool: simulation. First, robot simulators, such as Gazebo, are industry standard tools for designing physical robots. Then, simulating mobile robots is not necessarily a new idea, and has been done since the very beginning of mobile robots [3].

Our goal is to design and implement a practical simulator for networks of mobile robots that is focused on finding counter-examples and monitoring network behavior, rather than proving algorithms or providing visual representation. Our vision is that this tool is especially useful in the early stages of algorithm design to eliminate obviously wrong paths, and detect anomalies. It should not be seen as a replacement for formal tools, but as a replacement for researcher intuition when working on a network model or algorithm.

As such, it should be easy to use, understand and modify to include any algorithm or model.

It should also be capable of monitoring network behavior and output quantitative data points to compare real world performance, according to a given set of metrics, of proven algorithms in a given setting.

We first focus on the known limitations of this approach and highlight the difficulty of

encoding victory and defeat conditions for the computed executions and how it impacts our ability to reliably detect counter-examples, as well as the expected consequences of working in a discretized euclidean space, such as the impossibility of distinguishing **Convergence** and **Gathering**.
We show the limits of this framework through the problems of $\mathcal{OBLOT}$ FSYNC **Convergence**, and **Geoleader Election**.

8.2 Overview of the Framework

Our simulation framework is written from scratch in Python 3. Our design goal is to remain as close as possible to the theoretical model of Suzuki and Yamashita [83], in order to maximize readability and usability by the mobile robot distributed computing community.

Each mobile entity is thus encapsulated as an instance of the `Robot` class. In the case of the basic $\mathcal{OBLOT}$ model, robots have the following properties:

- A unique `name`.
- `x` and `y`, representing coordinates in the Euclidean plane.
- A list of `Robots` named `snapshot` that contains visible `Robots`.
- A `target`, which is a tuple of the `x` and `y` coordinates of the target.

The `Robot` class also contains three methods:

- The `LOOK` method uses the network as an input. It creates a list of the other `Robots` in network and assigns it to `self.snapshot`.
- The `COMPUTE` method uses `self.snapshot` to compute and assign `self.target`, according to the algorithm we want to check.
- The `MOVE` method updates `self.x` and `self.y` according to `self.target`.

This is summarized in figure 8.1

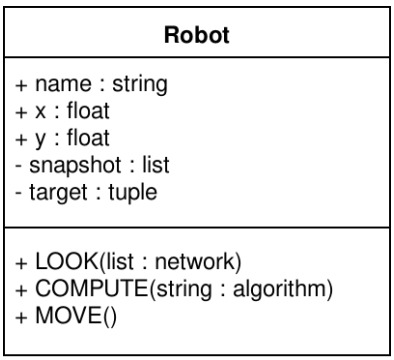


Figure 8.1 – Robot Class

Because robots are anonymous, `name` cannot be used for computing purposes, and is simply a way for the scheduler to reliably monitor the robots in the network. Similarly, robots cannot use `x` and `y` directly as they are disoriented.
The simulation consists of two parts: an initializing sequence and a loop.

The initializing sequence creates a `network` list, which contains all robots, according to simulation parameters. To circumvent the problem of the infinite number of initial positions, our simulation framework is based on the Monte-Carlo method for choosing initial configurations [69]. So, unless otherwise specified, the initial location of each robot is chosen uniformly at random within the bounds of the type used to represent positions. Using the Monte-Carlo method allows us to both minimize biases in the initial parameters, and arbitrarily increase the precision of the simulation by simply increasing the number of simulations. For each iteration of the main loop, a scheduling function is executed once. In the case of FSYNC, for each loop iteration, all robots in the network simultaneously perform a `LOOK`, then simultaneously perform a `COMPUTE`, and then simultaneously perform a `MOVE`. Using different schedulers, such as SSYNC or ASYNC only requires changing the scheduling function: SSYNC creates a non empty list of robots to be activated for a whole cycle, and ASYNC picks a single robot to be activated for a single phase. The loop terminates whenever a *victory* condition holds, which confirms the algorithm completed its intended task. In the case where an algorithm may fail, a *defeat* condition can also be used. For practical reasons, the loop has a maximum number of iterations. However, reaching this maximum should not be interpreted as either a failure or a success.

8.3 Scheduling

Modeling the FSYNC scheduler can be trivially done by performing all LOOK operations, then all COMPUTE operations, then all MOVE operations.

For the ASYNC and SSYNC schedulers, we rely on randomness to test as many executions as possible. To model the SSYNC scheduler, for each time step, we chose a non-empty subset of the network uniformly at random and perform a full cycle.

To model the ASYNC scheduler, we chose one robot uniformly at random and perform its next operation[1].

In the case of the ASYNC scheduler, we must also consider what happens if a robot performs a LOOK operation while another robot is moving. the $\mathcal{OBLOT}$ model usually considers that an adversary can chose the perceived location of the second robot to be anywhere between its initial position and its destination (on a straight line). Modeling this behavior could be easily done by changing the perceived coordinates in the LOOK operation uniformly at random between the location and the target of the perceived robot (on a straight line). However, existing literature about the ASYNC model shows that the most problematic scenarios appear when the outdated position perceived for a robot is its initial location. With our simulation framework, we also observed that always choosing the initial location when observing a given robot while it is in its MOVE phase yielded the most adversarial results, so, while our framework is able to simulate both perceptions, we assume this adversarial behavior in the sequel.

For all schedulers, our simulation framework supports both the rigid and the non-rigid settings. The rigid setting mandates a robot that selected a distinct target in the COMPUTE phase to always reach it in the MOVE phase. The non-rigid setting partially removes this condition: the robot may be stopped by the scheduler before it reaches the target, but not before it traverses a distance of at least δ, for some $\delta > 0$.

[1]Note that this model does not explicitly include simultaneous operations: we consider that the output of two simultaneous events $E1$ and $E2$ can be either the output of $E1$ then $E2$ or the output of $E2$ then $E1$, as proven in theorem 4.9.

8.4 Simulation Conditions

Our framework uses Monte-Carlo simulation for both the initial conditions and the scheduling. This means we can perform an arbitrarily large number of simulations, which in turn induces an arbitrarily more precise simulation. Therefore any criterion on either time, number of iterations, or precision is equivalent. Unless specified otherwise, 4 identical simulations are run in parallel, for one hour, on a modern quad-core CPU. We use the PyPy3 JIT compiler instead of the CPython interpreter, for better performance. Results of the 4 simulations are then compiled and analyzed.

8.5 Existing Simulators

There are two notable existing simulators for distributed agents: Sycamore and JBotSim.

JBotSim is a Java library for simulating distributed networks in general. While it appears to be able to simulate $\mathcal{OBLOT}$ robots, it is not designed to.

Sycamore is a Java program focused explicitly on mobile robots. However, it appears to be far more complex to build, use and modify than our proposal. Moreover, the latest version we could find seems to date back from 2016 and require unsupported versions of Java.

We found a third Java-based simulator, named oblot-sim[2]. We are, however, unsure of its provenance and design goals. All three simulators emphasize ease of use through a complete graphical interface. Our proposal focuses on extreme simplicity: In its current version, a complete instance of the simulator requires five separate files for a total of less than 30KB. We also believe that using Python instead of Java greatly improves portability and ease of understanding, which in turns allows researchers to more easily implement and test unusual settings.

Finally, our goal is not to visualize executions, but to simulate as many as possible to process the data from the executions.

8.6 Limitations of the Simulation

While the initial approach described in the previous section initially might seem sound and simple to work with, it results in two distinct problems. As stated previously, our objective with robot simulation is to reliably provide counter-examples whenever they should occur. This requires reliably detecting problematic executions, which is difficult for two reasons. First, success and defeat conditions for most mobile robot algorithms are written in a way that might not be directly usable in a computer simulation. Then, we show that issues predictably arise due to the nature of discretized floating point numbers compared to true real numbers used in mathematical models.

8.6.1 Halting the Simulation: *Victory* and *Defeat* Conditions

One of the goals of this simulation framework is to find counter-examples for a given algorithm and setting. To do so, we need to simulate the evolution of the network until one of two things happen:

- A sufficient condition has been met. This implies that the current execution is successful and a new simulation with a different initial configuration should begin. This is called a *victory condition*.
- A necessary condition has been violated. This implies that the current execution constitutes a counter-example. This is called a *defeat condition*.

[2] `https://github.com/werner291/oblot-sim`

We illustrate the difficulty of using such conditions in practice through the scope of one of the most common problems of mobile robots: **Gathering**.

The common victory condition for **Gathering** is the following, for two robots r_1 and r_2:

Condition 8.1 (Theoretical **Gathering** Victory)**.**
Gathering *is achieved if and only if, for any pair of robots in the network, the distance between the two robots is eventually always zero.*
This can also be written as $\exists t_0 \in \mathbb{R}_{\geq 0} : \forall t_1 \geq t_0, \forall (r_1, r_2) |r_1 r_2|_{t_1} = 0$

Where $|r_1 r_2|_t$ is the distance between r_1 and r_2 at time t of the execution.

However, this particular condition would require the ability for the simulator to infinitely simulate the future of the network, which is obviously impossible.

Moreover, the matching defeat condition would simply be

$$\nexists t_0 \in \mathbb{R}_{\geq 0} : \forall t_1 \geq t_0, \forall (r_1, r_2) |r_1 r_2|_{t_1} = 0$$

or

$$\forall t_0 \in \mathbb{R}_{\geq 0} : \exists t_1 \geq t_0, \exists (r_1, r_2) |r_1 r_2|_{t_1} \neq 0$$

which is unusable for the same reasons.
We instead define a more practical defeat condition:

Condition 8.2 (Practical **Gathering** Defeat)**.**
$\exists (t_0, t_1) \in (\mathbb{R}_{\geq 0})^2 : t_1 > t_0, inputs(t_0) = inputs(t_1), \exists t \in [t_0, t_1] / \exists (r_1, r_2) |r_1 r_2|_t \neq 0$

Where $inputs(t)$ is the set of all input parameters relevant to the algorithm. This is different from the configuration, which would contain *all* parameters of the network at a given point of the execution.

This input set is used as a practical way to detect cycles in the execution. For a deterministic algorithm, if all inputs of the algorithm are identical to a previously encountered set of inputs, then a cycle has been found.

The input set we use must be chosen such that for two sets S_1 and S_2, $S_1(t) = S_2(t) \implies \forall S_1(t+1), \exists S_2(t+1) : S_1(t+1) = S_2(t+1)$. In other words, regardless of the scheduling, two identical sets should not be able to generate different sets.

Theorem 8.1. *For two robots executing a deterministic algorithm, if condition 8.2 is true then condition 8.1 is false.*

Proof. For a deterministic algorithm, if condition 8.2 is true, there exists a scheduling starting from the initial configuration which reaches $inputs(t_0)$ and $inputs(t_1)$. Because $inputs(t_0) = inputs(t_1)$, there exists a cycle containing non-gathered configurations. Then the adversary scheduler can repeat this cycle infinitely and condition 8.1 is false. □

Theorem 8.2. *If the number of input sets is finite, then for two robots executing a deterministic algorithm, if condition 8.1 is false, then condition 8.2 is true.*

Proof. Any scheduling is infinite. So, if the total number of input sets is finite, then all schedulings each contain at least one cycle. If condition 8.1 is false, then there are no non-gathered cycles, so there is at least one gathered cycle which must be repeated, and condition 8.2 is true. □

One may naively use a similar reasoning to define a sufficient victory condition:

Condition 8.3 (Naive **Gathering** Victory)**.**
$\exists(t_0,t_1) \in \mathbb{R}^2_{\geq 0} : t_1 > t_0, inputs(t_0) = inputs(t_1), \forall t \in [t_0,t_1], \forall (r_1,r_2) |r_1r_2|_t = 0$

However, this condition ignores the fact that the scheduler may be able to not repeat this cycle by carefully choosing the activation order of the robots.
A proper condition that could be usable regardless of the scheduler is the following:

Condition 8.4 (Practical **Gathering** Victory)**.**
$\forall (r_1,r_2) \exists t_0 \in \mathbb{R}_{\geq 0} : |r_1r_2|_{t_0} = 0 \wedge \forall \mathcal{S}, \exists t_1 > t_0 : inputs(t_0) = inputs(t_1), \forall t \in [t_0,t_1], |r_1r_2|_t = 0$
With $\mathcal{S}$ a scheduling.
In other words, there exists a time after which all robots are stuck in gathered cycles.

Analyzing configurations and finding cycles in the execution is not an issue for our simulator. The main difficulty here lies in our ability to properly model the configuration using the input set. If the set is too restrictive and omits relevant parameters, then we find cycles that do not actually exist. Similarly, a set that is not restrictive enough may hide actual cycles. This depends on both the robot model and the algorithm used to solve the problem.
In the case of **Rendezvous** or **Gathering** for two robots, the standard algorithm for the FSYNC scheduler targets the midpoint between the two robots and is described in algorithm 8.1.

Algorithm 8.1 Basic FSYNC **Rendezvous**

```
self.target[0] = (self.x + snapshot[0].x)/2
self.target[1] = (self.y + snapshot[0].y)/2
```

In the euclidean space, the number of configurations appears to be infinite. However, as we showed in chapter 3.2, because robots are disoriented, the algorithm uses no information on distance, or coordinate systems, all configurations are identical. Then, the input set is actually empty. This implies that an algorithm succeeds if and only if the network is gathered after the first activation of both robots. Otherwise, the defeat condition is immediately true for rigid movement.

For the sake of providing a second example, let us consider that robots are endowed with weak local multiplicity detection, meaning can distinguish a non-gathered configuration from a gathered configuration. This allows us to modify the algorithm to algorithm 8.2.

Algorithm 8.2 FSYNC **Rendezvous** with Multiplicity

```
if ¬gathered
    self.target[0] = (self.x + snapshot[0].x)/2
    self.target[1] = (self.y + snapshot[0].y)/2
```

In this case, the gathered state is a relevant input parameter, and should be included in the input set. Now, all gathered configurations are considered identical and all non-gathered configurations are considered identical. This means that the robots must still gather after the first activation. However, while this was already considered a cycle with the empty set, if robots are now gathered, the input set is different and no cycle has yet been reached. The first cycle is reached after the second activation. If the robots stay gathered, then this is a gathered cycle and should not trigger the defeat condition. However, if for some reason the robots were to separate after the second activation, this would constitute a non-gathered cycle with the first input set, and the defeat condition would be triggered.

Using this reasoning, we check our simulator against our 2-color ASYNC algorithm [53] and the 2-color SSYNC algorithm from Viglietta [84]. For Heriban 2-color, we accurately find no counter-example and all executions lead to the victory condition in ASYNC, SSYNC and FSYNC. For Viglietta 2-color, we accurately find no counter-example and all executions lead to

the victory condition in, SSYNC and FSYNC, and we find counter-examples which trigger the defeat condition in ASYNC.

We perform a similar study for a weaker version of **Gathering**, called **Convergence**. The common condition for **Convergence** is the following:

Condition 8.5 (Theoretical **Convergence** Victory)**.**
Convergence *is achieved if and only if, for any distance ε greater than zero, the distance between any pair of robots is eventually always smaller than ε.*
This can also be written as $\forall \varepsilon \in \mathbb{R}_{>0}, \exists t_0 \in \mathbb{R}_{\geq 0} : \forall t_1 \geq t_0, \forall (r_1, r_2) |r_1 r_2|_{t_1} \leq \varepsilon$

Note that, as we expect, **Gathering** implies **Convergence**, but **Convergence** does not imply **Gathering**. In this case, the distance between the two robots is a relevant parameter to check whether or not the problem is solved. However, since it does not change the behavior of the algorithm, it is still not part of the input set.

We define the following defeat condition:

Condition 8.6 (Practical **Convergence** Defeat)**.**
$\exists (r_1, r_2) : \exists (t_0, t_1) \in (\mathbb{R}_{\geq 0})^2 : t_1 > t_0 \wedge inputs(t_0) = inputs(t_1) \wedge |r_1 r_2|_{t_0} \leq |r_1 r_2|_{t_1}$

Theorem 8.3. *For a deterministic algorithm, if condition 8.6 is true, then condition 8.5 is false.*

Proof. Similarly to **Gathering**, this condition implies a cycle where distance does not decrease, so the adversary scheduler can repeat it infinitely and prevent **Convergence**. □

This does *not* imply that the distance between the two robots must always be strictly decreasing in the general case, as this would neither be a sufficient nor a necessary condition.

Because ε can be infinitely small, we cannot chose the 'right' ε to properly define a victory condition.

8.6.2 The Consequences of the Discretized Euclidean Plane

While we could be tempted to define a similar victory condition than for **Gathering**, the question of ε remains. Floating point numbers are obviously incapable of infinite precision. So, because any number greater that zero is a valid choice, if ε is smaller than the minimum positive number that can be represented in the chosen floating point precision, it cannot be distinguished from a true zero. This means that small enough distances between two robots cannot be distinguished from a gathered state.

So it is intrinsically impossible to distinguish **Convergence** from proper **Gathering**.

Let us modify algorithm 8.1 so that both robot move towards the midpoint, but only move a distance of $\frac{|r_1 r_2|}{2} - \frac{\delta}{2}$ instead of $\frac{|r_1 r_2|}{2}$. In theory, this algorithm does not lead to **Rendezvous**, as robots reach a distance of δ after their first activation. However, if δ is small enough, the precision of floating point numbers is such that $\frac{|r_1 r_2|}{2} - \frac{\delta}{2}$ and $\frac{|r_1 r_2|}{2}$ appear identical, and the distance $|r_1 r_2|$ appears to be zero. This is essentially a **Convergence** algorithm that is fast enough to be mistaken for a **Rendezvous** algorithm. In practice, there is very little that can be done against this sort of behavior and conditions for **Gathering** should not be considered reliable.

On the other hand, under different circumstances, the discrete nature of the simulation can instead lead theoretically good executions to fail in practice. Let us consider a network of two robots r_1 and r_2 such that r_2 does not move and r_1 moves to the midpoint. This should trivially lead to **Convergence**. Let us now assume that $r_1.y = r_2.y$ and $r_1.x$ and $r_2.x$ are such that $r_2.x$ is

the smallest float greater than $r_1.x$. This possibly leads to $\frac{r_1.x + r_2.x}{2} = r_1.x$, so r_1 stops moving and the defeat condition for **Convergence** is wrongly activated.

We test this by setting $r_1.y = r_2.y = 0$, picking $r_1.x$ at random in $[0,1]$ and picking $r_2.x$ at random in $[2,3]$ so that $r_1.x < r_2.x$.

In the first case, r_1 moves to the midpoint and r_2 does not move. This results in approximately 37.5% of one million attempts wrongly failing **Convergence**.

In the second case, r_2 moves to the midpoint and r_1 does not move. This results in approximately 25.0% of one million attempts wrongly failing **Convergence**.

This asymmetry may be explained by biases in the binary64 approximation. Regardless, this is a real, hard to predict problem with a non-negligible chance of happening and requires careful analysis of found counter-examples.

Problems with limited float precision also appear when simulating **Geoleader Election**.

Geoleader Election is successful if given a set of robots, each with their own coordinate system, robots can all deterministically agree on a same robot, called the **GEOLEADER**.

Geoleader Election is known to be impossible in the general case [38] because of possible symmetries in the network. As we have shown in chapter 4.5, in practice, this is solved by using randomized algorithms to break such symmetries.
Let us consider the state-of-the-art algorithm 5.1 by Canepa and Gradinariu Potop-Butucaru [19] for three robots.

For this particular algorithm, there are three cases:

1. The common case, where one angle is greater than the two others.
2. A rare case where two angles are identical and the third one is smaller.
3. The rarest case where all angles are identical. In that case, a Bernoulli trial is required to degrade to the other cases.

Let us assume a network of three robots, $[r_1, r_2, r_3]$ such that r_1 is placed at coordinates $(-0.5, 0)$ and r_2 at $(0.5, 0)$.

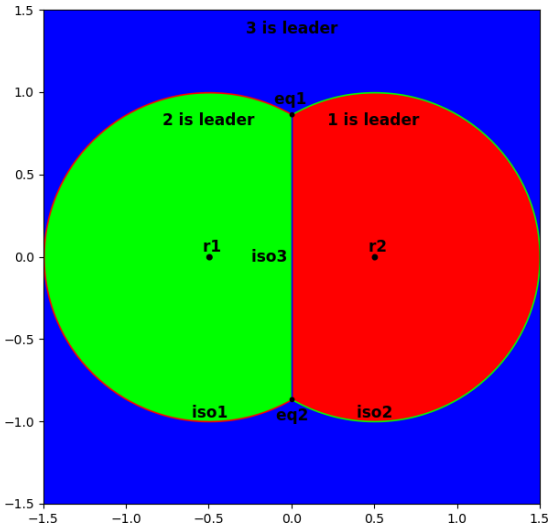
We show where each case appears in figure 8.2a). The third case occurs if r_3 is at $(0, \pm\frac{\sqrt{3}}{2})$ which are noted as points $eq1$ and $eq2$. Positions of r_3 which lead to the second case are noted as $iso1$, $iso2$ and $iso3$.

However, it is *not* possible, using floating point numbers, to have x such that $x^2 = 3$. It is then impossible, regardless of the quality of the simulation, to place r_3 on $eq1$ or $eq2$, despite being possible in theory.

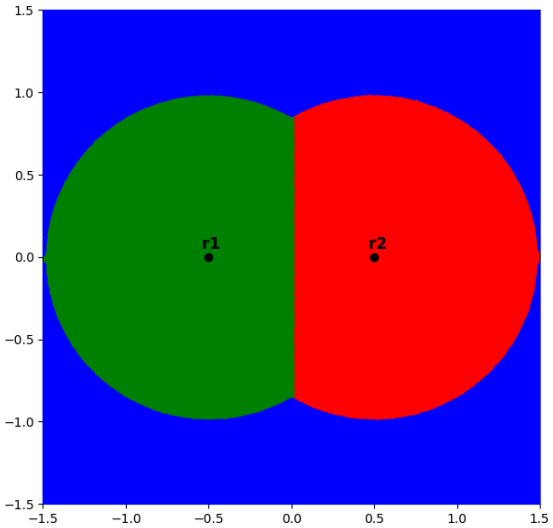
Similarly, an infinitely large number of point mathematically located on the circular arcs of the second case cannot be represented properly using floating point numbers.

To test this, each robot is given a new property 'Leader', which is a string containing the name of the **LEADER** robot. We perform the simulation and display the results in figure 8.2b.
As we predicted, the fact that real numbers cannot be properly represented in our discrete, floating point space prevents the simulator from finding the known counter-example in the case of 3-robots **Leader Election**. Furthermore, the three circular arcs on which the second case occurs have a combined surface theoretically equal to zero. Therefore, they are statistically impossible to find using our Monte-Carlo simulation.

However, it should be noted that, even in a world of perfect sensors, building an equilateral triangle would require placing the third robot with physically impossible precision. So, while this counter-example exists from a mathematical standpoint, it could never occur in a more realistic setting. So when considering practical robot, this could be considered a minor issue.



(a) **LEADER** Depending on the Location of r_3. Red, green and blue represent r_1, r_2 and r_3, respectively



(b) Simulation for 3-robot **Leader Election** with Perfect Vision Sensors. No isosceles or equilateral point was found

On the contrary, the use of a discretized euclidean space could be viewed as massive advantage compared to the regular, continuous model. We recall chapter I, where we discussed the inherent unrealistic hypothesis of robots being able to store and process snapshots of infinite precision. In this approximated context, snapshots have a known, maximum size, depending on the chosen precision for the coordinates of other robots.

So, in this context, storing a snapshot for a full cycle becomes a trivial matter, and using algorithm **SyncSim** described by Das *et al.* [30, 31], to simulate an FSYNC scheduling under an ASYNC scheduler, becomes possible without additional unrealistic hypotheses.

As a result, we believe designing algorithms that function in this context should be a priority, as it would allow mobile robots to only need to function using the FSYNC scheduler, and would remove of the hypothesis of infinite precision. One such algorithm is shown in chapter 10.5.

Chapter 9

Fuel Efficiency in the Usual Settings

An overwhelming majority of the research on mobile robots has been focused on proving, under a given set of conditions, whether there exists a counter example to a given problem. On the other hand, the practical efficiency of a given algorithm (with respect to real-world criteria such as fuel consumption) was rarely studied by the distributed computing community, albeit commanded by the robotics community [4, 87].

Fuel-constrained robots have been considered in the discrete graph context, for both exploration [41] and distributed package delivery [20], but, to our knowledge, no study considered the two-dimensional Euclidean space model that was promoted by Suzuki and Yamashita [83]. A possible explanation for this situation is that the more complex the algorithm (or the system setting), the more difficult it becomes to rigorously find the worst possible execution.

9.1 Rendezvous Algorithms

We first quantify the maximum traveled distance and the average traveled distance for several known **Rendezvous** algorithms. We chose the *Center Of Gravity algorithm* [83], our two-color ASYNC algorithm (*Her2*) [53], the two-color algorithm (*Vig2*) by Viglietta [84], which is known to solve **Rendezvous** in SSYNC and **Convergence** in ASYNC, the three-color algorithm (*Vig3*) by Viglietta [84], the four-color algorithm (*Das4*) by Das *et al.* [30, 31]. We also investigate the algorithms for unreliable compasses by Izumi *et al.* [61]: the SSYNC static-error compass algorithm (*Stat SSYNC*), which, despite its name, works in ASYNC, the SSYNC dynamic-error compass algorithm (*Dyn SSYNC*), which does not work in ASYNC, and the ASYNC static-error compass algorithm (*Dyn ASYNC*).

We take advantage of the modularity of the simulator. The robot class now carries several new properties: `color`, which is the color a robot presently broadcasts ; `compass`, which is the type of compass and error, *i.e.* 'none', 'static' or 'dynamic' ; `compass_error`, which is the maximum error allowed for the compass ; and `compass_offset`, which is the current compass error. The color is changed at the end of the `COMPUTE` method. Depending on the value of `compass`, `compass_offset` is either chosen during the initialization, or at the beginning of every `LOOK` method.

Each algorithm is first carefully analyzed on paper to find the worst possible execution. Simulations are then run according to the aforementioned protocols. Due to limitations described in section 8.6.2, we actually assess those protocols for a degraded notion of **Convergence** rather than **Gathering**. The distance traveled is expressed relative to the initial distance between the two robots. In practice, the first robot is always located at $\{0,0\}$ and the second robot is placed at random on the circle of radius 1 centered on $\{0,0\}$. Algorithms are only tested using simple initial configuration[1], as complete self-stabilization implies existing arbitrary pending moves and renders fuel efficiency mostly pointless.

[1] see section 4.2.1

Results are summed up in table 9.1. Red denotes cases where the simulation was stuck in non-gathered cycles and had to be manually unstuck. Details as to why this happened are provided below.

For scale, running 4 instances of *Vig3* for one hour under the ASYNC scheduler resulted in $\simeq$ 14 million total executions.

	Rigid FSYNC	Rigid SSYNC	Rigid ASYNC	Non-Rigid FSYNC	Non-Rigid SSYNC	Non-Rigid ASYNC
CoG	1 / 1	1 / 1	2.66 / 3	1 / 1	1 / 1	2.36 / 3
Das4	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Vig3	1 / 1	1 / 1	**2 / 1**	1 / 1	1 / 1	**2 / 1**
Vig2	1 / 1	1 / 1	2.94 / 3	1 / 1	1 / 1	2.75 / 3
Her2	1 / 1	1 / 1	2 / 2	1 / 1	1 / 1	2.54 / 3
Stat SSYNC	Stuck	Stuck	Stuck	Stuck	Stuck	Stuck
Dyn SSYNC	1	1	N/A	1.41	1.41	N/A
Dyn ASYNC	6.2	6.2	6.2	49.7	43.4	44.4
Stat SSYNC ε	3	3	3	3.95	4	4
Dyn SSYNC ε	1	1	N/A	1.41	1.41	N/A
Dyn ASYNC ε	6.2	6.2	6.2	49.4	50.9	41

(a) Maximum Traveled Distance
Found / Predicted

	Rigid FSYNC	Rigid SSYNC	Rigid ASYNC	Non-Rigid FSYNC	Non-Rigid SSYNC	Non-Rigid ASYNC
CoG	1	1	1.35	1	1	1.08
Das4	1	1	1	1	1	1
Vig3	1	1	1.06	1	1	1.01
Vig2	1	1	1.12	1	1	1.04
Her2	1	1	1.15	1	1	1.03
Stat SSYNC	Stuck	Stuck	Stuck	Stuck	Stuck	Stuck
Dyn SSYNC	1	1	N/A	1.03	1.03	N/A
Dyn ASYNC	2.18	2.00	2.07	1.99	1.88	2.00
Stat SSYNC ε	1.95	1.59	1.80	1.43	1.34	1.46
Dyn SSYNC ε	1	1	N/A	1.03	1.03	N/A
Dyn ASYNC ε	2.21	1.93	2.09	1.96	1.91	2.04

(b) Average Traveled Distance

Table 9.1 – Maximum and Average Traveled Distances

While most results match the predictions, our pen and paper analysis missed a worst case execution for ASYNC *Vig3*, which was found by the simulator (highlighted in bold in figure 9.1). This highlights the difficulty of manually finding the maximum distance even with simple algorithms and settings. It should be noted that rigid motion yields worst results than non-rigid. This is normal because increasing the traveled distance relies on picking a target outside of the $[r_1, r_2]$ segment, and when this is the case, performing the full motion increases the traveled distance more than performing it partially. Thus, unless stated otherwise, all further simulations assume rigid motion.

The difference between SSYNC and ASYNC with respect to efficiency becomes apparent, as under the ASYNC scheduler, optimal fuel consumption mandates using four colors, while a simple oblivious algorithm is sufficient in SSYNC.

The algorithms using compasses yield the most interesting results. First, numerous simulations of the SSYNC static algorithm became stuck.

These failures were due to the fact that the sine and cosine operations used in the algorithms tend to sum errors, and there is a possibility that a robot moves in a way that results in an angle of exactly 0, which actually randomly yields an angle of either $0-\varepsilon$ or $0+\varepsilon$, where ε is a very small positive number. This in turn results in unsolvable cycles that prevent **Convergence**. As ε was never larger than 10^{-9}, we chose to prevent this behavior by slightly enlarging the interval of the condition that should be triggered on an angle of zero to an angle in $[-10^{-6}, 10^{-6}]$. We do the same for all conditions for consistency. So any condition that should be true for angles in

$[A,B[$ are now true for angles in $[A-10^{-6},B-10^{-6}[$, in $[A,B]$ now in $[A-10^{-6},B+10^{-6}]$, in $]A,B]$ now in $]A+10^{-6},B+10^{-6}]$ and in $]A,B[$ now in $]A+10^{-6},B-10^{-6}[$.

Interestingly, this new condition only had notable impact on the static error algorithm. Indeed, these errors could be seen as small dynamic random angle errors. Since the static error algorithm is not designed to be resilient against dynamic errors, it fails whenever they appeared. This also demonstrate the resilience of the dynamic error algorithms.

9.2 Convergence For *n* Robots

Cohen and Peleg [24] proved the Center of Gravity (CoG) algorithm solves **Convergence** for n robots under the ASYNC scheduler. We analyze the fuel consumption of the algorithm under both the SSYNC and ASYNC schedulers. Results for the minimum, maximum, and average distance traveled are show in table 9.2. We use the sum of the distances to the CoG in the initial configuration as a baseline unit of distance, *i.e.* the distance traveled in FSYNC.

	Minimum	Maximum	Average
ASYNC			
n = 2	1.00	2.74	1.34
n = 3	0.771	2.96	1.39
n = 4	0.751	2.82	1.37
n = 5	0.818	2.69	1.35
n = 10	1.00	1.97	1.26
n = 25	NC	NC	NC
n = 50	NC	NC	NC
SSYNC			
n = 2	1.00	1.00	1.00
n = 3	0.752	1.50	1.08
n = 4	0.689	1.91	1.10
n = 5	0.711	2.18	1.11
n = 10	0.883	2.15	1.12
n = 25	0.977	1.57	1.09
n = 50	0.995	1.31	1.07

Table 9.2 – Traveled Distances for CoG

It should be noted that, while previous results are based on at least hundreds of thousands of simulations, due to the increase in simulation complexity, in ASYNC, for $n = 25$, only 31 simulations could be computed under an hour. So they were discarded. Similarly, for $n = 50$, no simulation could be finished under an hour.

Looking at the results, one element immediately jumps out: for $n \geq 3$, the CoG algorithm wastes movements. This is easy to understand: robots move towards the center of gravity, which for 3 or more robots is different from the geometric median (*a.k.a.* the Weber point), which would actually minimize movement. Our tests seem to indicate that aiming for the median instead of the CoG can reduce traveled distance by up to 30%. However, it is a known result that no explicit formula for the geometric median exists.

In practice, when trying to minimize traveled distance, **Convergence** for n robots should rely on an approximation of the geometric median rather than the center of gravity.

Chapter 10

Analyzing Algorithms in Realistic Settings

In chapter 8.6.2, the simulation of inaccurate compasses yielded extremely interesting results. To follow this track, we now focus in this chapter on the setting where visual sensors are inaccurate. In more details, we analyze the Center of Gravity (CoG) algorithm for **Rendezvous** in this setting, as well as the **Geoleader Election** algorithm by Canepa and Gradinariu Potop-Butucaru [19].

We then analyze $\mathcal{LUMINOUS}$ **Rendezvous** algorithms against a new type for error where robots may read the wrong color in their snapshot.

10.1 Visibility Sensor Errors

To study the impact of inaccurate sensors, we consider three different models for vision error. For a robot r_1 looking at a robot r_2 located in (x,y) in the Cartesian coordinate system centered at r_1, and located at (r,θ) in the polar coordinate system centered at r_1, we define:

- The *absolute* error model [68] uses a constant value err. A first number R_{err} is picked uniformly at random in $[0,err]$, and a second θ_{err} in $[0,2\pi]$. The perceived position of r_2 is then $(x+R_{err}\cdot cos(\theta_{err}), y+R_{err}\cdot sin(\theta_{err}))$.

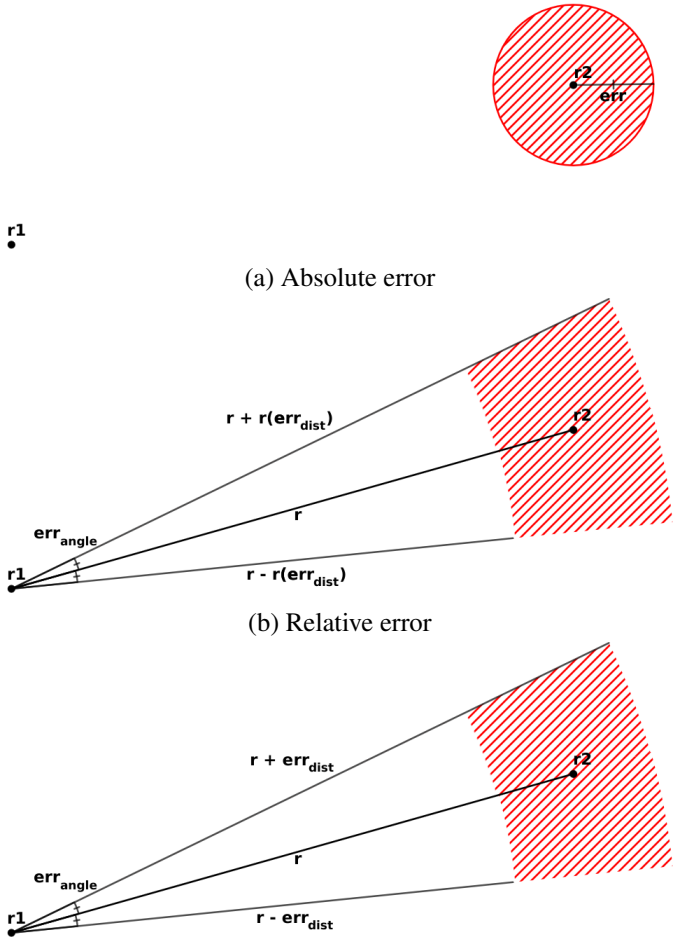
- The *relative* error model [25] uses two constants err_{dist} and err_{angle}. Two numbers R_{err} and θ_{err} are picked at uniformly at random in $[-err_{dist}, err_{dist}]$ and $[-err_{angle}, err_{angle}]$. The polar coordinates of r_2 are then perceived to be $(r+r\cdot R_{err}, \theta+\theta_{err})$

- The *absolute-relative* error model is similar to relative error, but the perceived polar coordinates are $(r+R_{err}, \theta+\theta_{err})$

These error models are depicted in figure 10.1.

It should be noted that each model could be used to accurately model errors in different types of sensors.

The absolute error model is interesting because it is simple to compute, requires no change of coordinate system, uses a single parameter, and closely matches the behavior of robots where the LOOK phase is an abstraction of GPS-type coordinates exchanges [86]. The two relative models are more complex from a computing perspective, but closely match the use of either computer vision or telemetry sensors. Both carry an angular error matched with either proportional or absolute distance error. Which type of distance error is more appropriate would depend on the exact type of sensor. This requires adding three properties to the `Robot` class:



(a) Absolute error

(b) Relative error

(c) Absolute-relative error

Figure 10.1 – Types of Errors
The r_2 point is the actual location of robot r_2, and the red hashed area is the set of possible detected positions by robot r_1.

- `LOOK_error_type`, which is a string that defines the type of error and can be either `'none'`, `'relative'`, `'absolute'`, or `'abs-rel'`.
- `LOOK_distance_error`, which is a float and matches either err or err_{dist}, depending on the type of error.
- `LOOK_angle_error`, which is a float and matches err_{angle}.

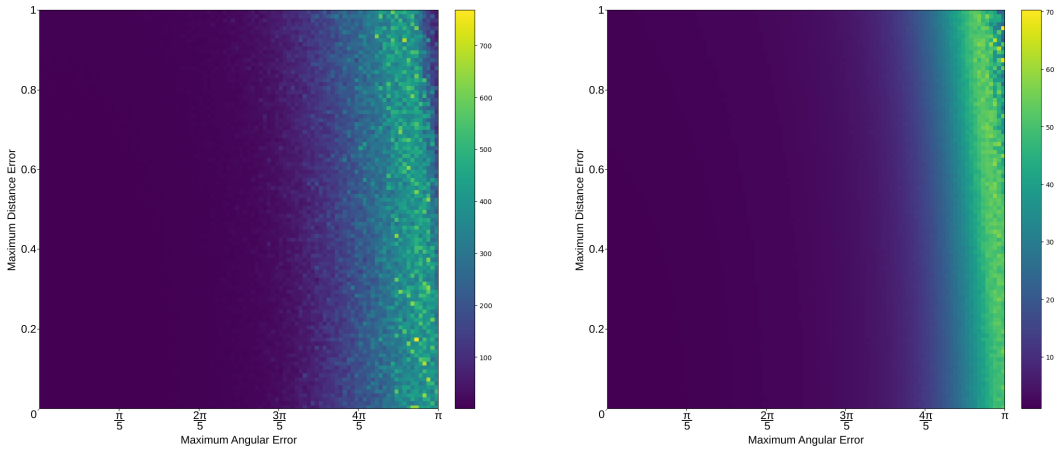
Robots then chose the corresponding error (with parameters chosen uniformly at random) when performing their LOOK operation.

10.2 Convergence for n=2

Convergence with vision error using the CoG algorithm has already been studied by Cohen and Peleg [25]. The error model they considered is identical to our relative error model. The paper states that **Convergence** with distance error using the CoG algorithm is impossible in the general case. This is, however, only true for $n \geq 3$, which the authors omit to mention. In the case $n = 2$, it appears to be theoretically impossible to make the algorithm diverge for a distance

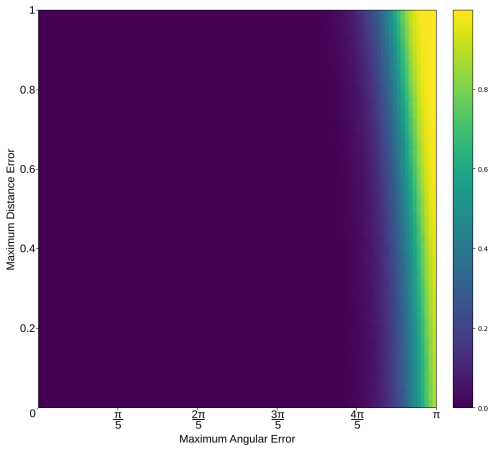
error smaller than a 100%, or $err = 1$. We can reasonably ignore the case of an error greater than 100%, as it would allow for a robot to perceive another one directly behind itself.

To our knowledge, no formal result exists regarding the angle error. In theory, the maximum angle error is π. We simulate **Convergence** for $n = 2$ robots using the CoG algorithm for the relative error model. The error for each robot is chosen uniformly at random at the beginning of the execution.



(a) Maximum Traveled Distance

(b) Average Traveled Distance



(c) Proportion of Diverging Executions

Figure 10.2 – Movement and Divergence of the CoG Algorithm for Two Robots with Inaccurate Visibility Sensors

We must also consider the now possible case of a diverging algorithm. Since the execution is random, any setting should *eventually* converge. However, we must put a reasonable stopping condition in case the execution is clearly diverging. We chose to activate the defeat condition if the distance between the two robots becomes ten times larger than the distance in the initial configuration.

Note that the apparent decrease in maximum and average traveled distance for higher angle error is most likely due to the increase of diverging executions (fewer executions converge, but the traveled distance for those is shorter).

It appears clearly that the angular error has a much greater potential for both preventing **Convergence**, and making robots waste fuel. Indeed, when the angular error remains below $3\pi/5$, a distance error up to 100% can be tolerated with no performance loss.
To give some perspective, the realistic setting of a 10% vision error with a 1° angle error yields a maximum traveled distance of 1.221 and an average of 1.036, with no divergent executions out of more than 500 million data points.

10.3 Compass Errors

In the particular case of compass based algorithms of Izumi *et al.* [61], rendezvous is possible when the compasses are inaccurate. More specifically, the maximum tolerated errors are $\frac{\pi}{2}$, $\frac{\pi}{4}$ and $\frac{\pi}{6}$ for the static SSYNC, dynamic SSYNC, and dynamic ASYNC algorithms, respectively. In our simulation we chose static errors, for consistency, with values up to $\frac{49\cdot\pi}{100}$, $\frac{24\cdot\pi}{100}$ and $\frac{16\cdot\pi}{100}$, to avoid possible edge cases.

Results of maximum and average traveled distances for these algorithms are detailed in table 10.1.

	Rigid FSYNC	Rigid SSYNC	Rigid ASYNC	Non-Rigid FSYNC	Non-Rigid SSYNC	Non-Rigid ASYNC
Stat SSYNC	194	5.02E+03	4.73E+03	1.16E+06	5.33E+05	2.69E+07
Dyn SSYNC	5.88E+24	1.63E+13	N/A	4.69E+13	2.77E+11	N/A
Dyn ASYNC	6.21E+29	6.21E+29	6.21E+29	6.06E+127	5.54E+135	1.90E+123

(a) Maximum Traveled Distance

	Rigid FSYNC	Rigid SSYNC	Rigid ASYNC	Non-Rigid FSYNC	Non-Rigid SSYNC	Non-Rigid ASYNC
Stat SSYNC	6.14	6.34	10.7	8.35	7.99	54.3
Dyn SSYNC	1.61E+17	71.2	N/A	49.8	13.5	N/A
Dyn ASYNC	7.26E+23	7.75E+21	4.21E+25	1.54E+51	9.61E+38	8.60E+43

(b) Average Traveled Distance

Table 10.1 – Maximum and Average Traveled Distances for **Rendezvous** with Inaccurate Compasses

We observe that the unreliable compasses are used in a way that makes robots rotate around each other until they are oriented in such a way that one robot moves while the other stays, regardless of the error. However, there are no provisions in these algorithms to limit distance increases during the rotating phases, which explains the results. Detailed observation shows the distance between the two robots can gradually diverge towards infinity during rotation and then converge to zero in a single cycle. This also proved a challenge to our **Convergence** criterion: robots could converge at rather large coordinates such that the coordinates of robots are in succession, but, since the accuracy of floating point numbers decreases as the number increase, the distance between the two robots was greater than 10^{-10}. We changed the criterion to $|r_1r_2| < max(10^{-10}, |Or_1| \cdot 10^{-10})$, with O the point of coordinates $\{0,0\}$.

10.4 Geoleader Election

Let us consider the state-of-the-art algorithm 5.1 by Canepa and Gradinariu Potop-Butucaru [19] for $n = 3$.

Looking at the previous results from section 8.6.2, we notice that the borders between each zone should be an issue for imperfect sensors, as different errors for different robots may lead to robots electing different **LEADER** robots.
We demonstrate this phenomenon in figure 10.3 for the case of absolute vision error. On top is the actual configuration, where angles $\widehat{r_1r_2r_3}$ and $\widehat{r_2r_1r_3}$ are equal[1], and angle $\widehat{r_1r_3r_2}$ is smaller than both, so r_3 should be elected. The red circle shows the possible perceived position of r_3 by r_1 and r_2 due to vision error. In the bottom left case, we show a possible perception by r_1 where r_1 should be elected **LEADER**, as $\widehat{r_2r_1r_3}$ is now greater than $\widehat{r_1r_2r_3}$. On the lower right, r_2 similarly thinks it should be elected. Now, two different robots consider themselves **LEADER** and the election process fails.

We now use the absolute model to simulate **Geoleader Election** with $err = 0.001$, for $n = 3$.

[1] Because robots have no chirality, angles cannot reliably be distinguished from their opposite. So, two opposite angles may always be considered equal.

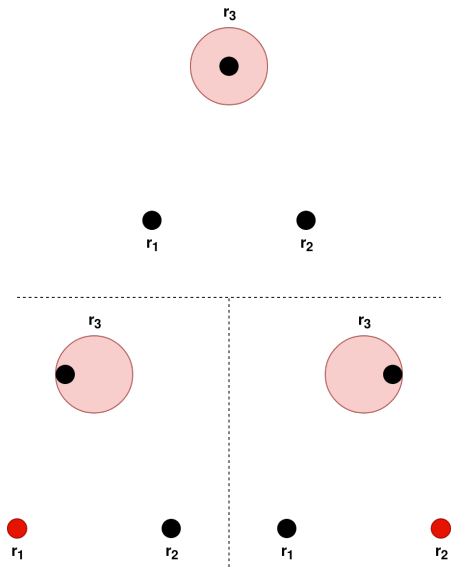


Figure 10.3 – Example of **Leader Election** Failure Due to Imperfect Vision

This simulation yields $\simeq 0.1\%$ of errors in total, where two robots compute different **LEADER** robots, and is shown in figure 10.4.

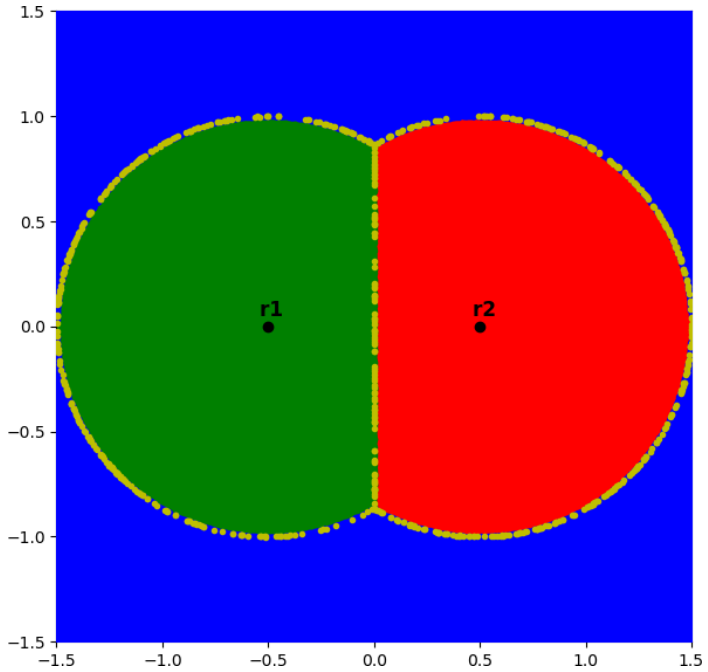


Figure 10.4 – Simulation for 3-robot **Leader Election** with Absolute Vision Error
Yellow points represent configurations where the error generates two different **LEADER** robots.

10.5 Errors in Color Perception

Because our simulation framework can be easily modified to accommodate different functions for the LOOK phase, we decide to verify the robustness of the Das4 [31], Vig3 [84], Vig2 [84] and Her2 [53] algorithms against vision errors. More specifically, we define a new model for $\mathcal{LUMINOUS}$ robots where robot can perceive the wrong color for the other robot when performing their LOOK phase.

Our error model is the following:

- The parameter for the error model is called D and is defined with regards to the initial distance between the two robots, *i.e.* $D = 2$ means that D is equal to twice the initial distance between robots.
- If the distance X between the two robots is greater than D, then robots have a probability of 0.5 of perceiving a wrong color at the end of their LOOK phase.
- If the distance X between the two robots is smaller than D, then robots have a probability of $\frac{X}{2 \cdot D}$ of perceiving a wrong color at the end of their LOOK phase.

This model is summed up in figure 10.5.

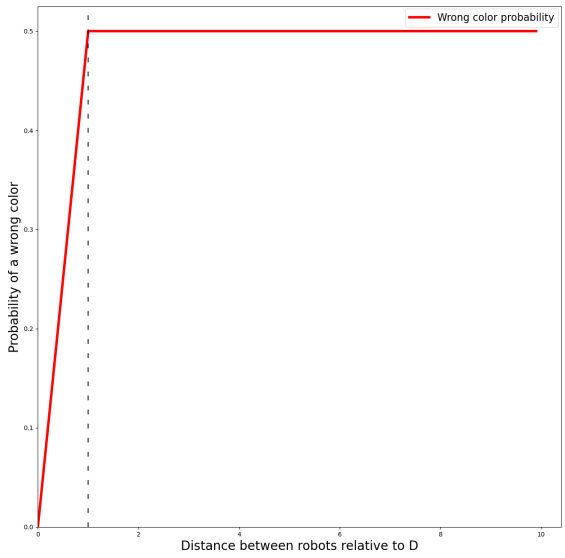


Figure 10.5 – Probability of perceiving a wrong color depending on the distance between robots.

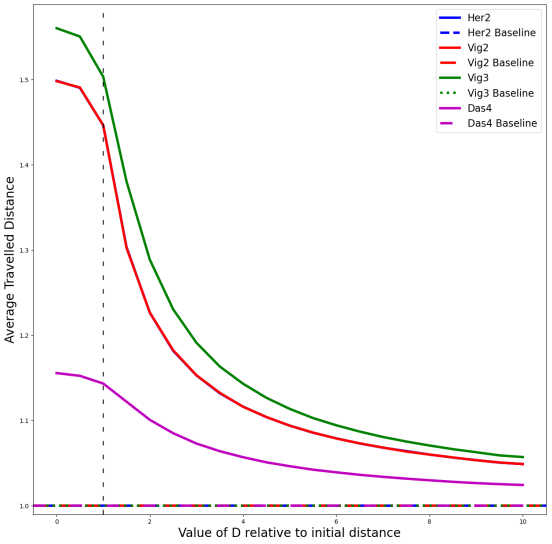
Before performing simulation under this vision model, we made the following assumptions on the results :

1. We assumed Her2 and Vig2 behave identically. In the FSYNC and SSYNC cases, algorithms are identical, and the differentiating ASYNC case should not be statistically significant.
2. We assumed that all algorithms should be similarly impacted by the vision errors, so the hierarchy of fuel efficiency established in chapter 8.6.2 should hold.
3. We assumed the average traveled distance and average activations with errors are always greater than with no errors.

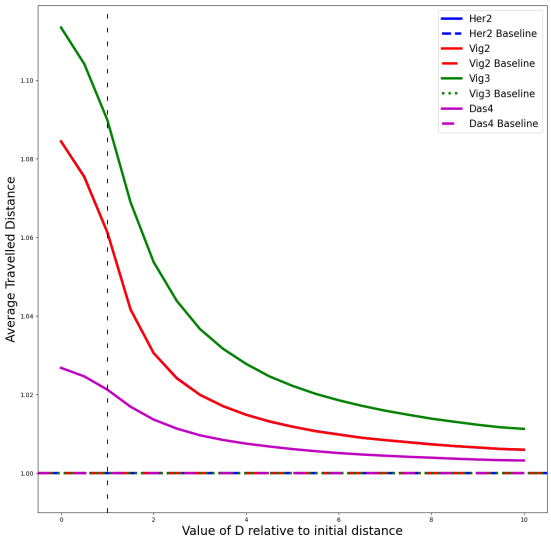
4. We assumed the sharp transition at $D = 1$ in the error model should be visible in the results.

Any deviation from these hypotheses we made as researchers should be investigated to figure out the root cause, and better our understanding of this model.
We perform the same number of 100,000,000 simulations per scheduler and algorithm. For each simulation, D is chosen at random between 0 and 10. For each graph, we also show the baseline average in the case with no vision error.

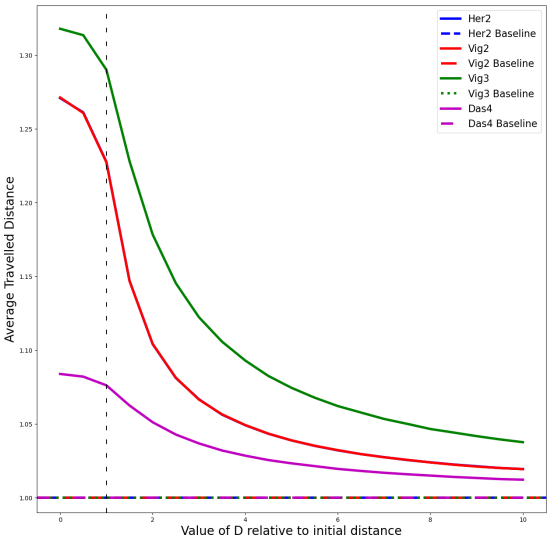
Using the same simulator, we also plot the average number of activations required to achieve **Convergence**. Under the ASYNC scheduler, this is the number of time a robot performed a phase. Under FSYNC, this is the number of FSYNC cycles. Under the SSYNC scheduler, this is the number of cycles per robot. In other words, a cycle where one robot is activated counts as one activation while a cycle where both robots are activated counts as two. Detailed figures with confidence intervals are provided in the appendix.



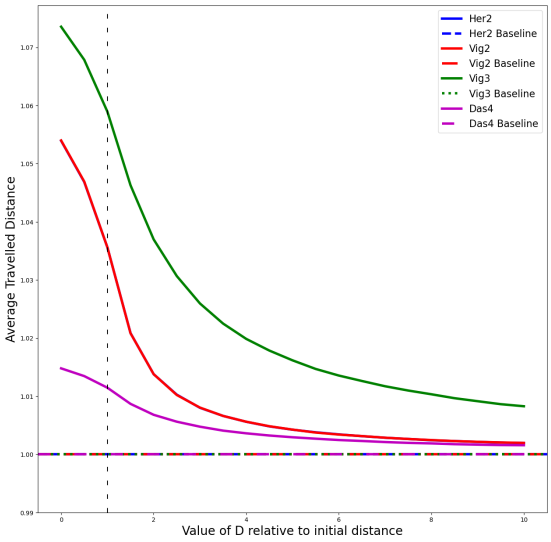
(a) Average traveled distance under the FSYNC rigid scheduler.



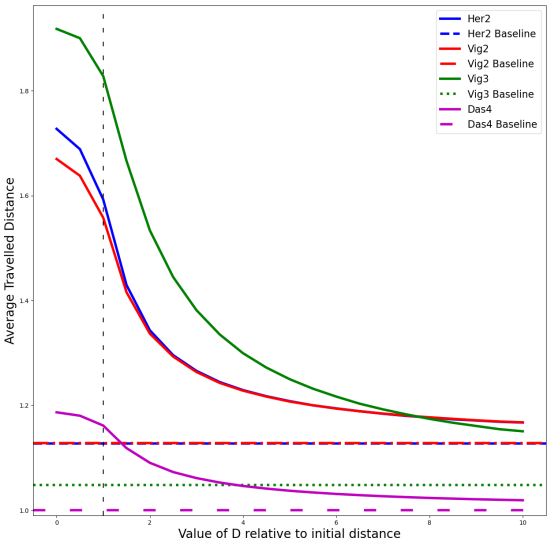
(b) Average traveled distance under the FSYNC non-rigid scheduler.



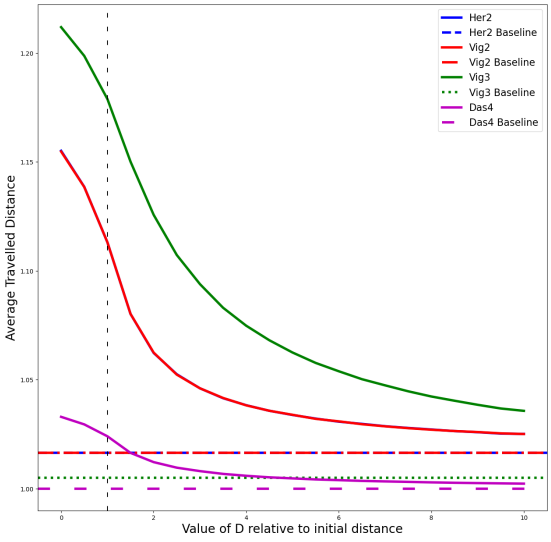
(c) Average traveled distance under the SSYNC rigid scheduler.



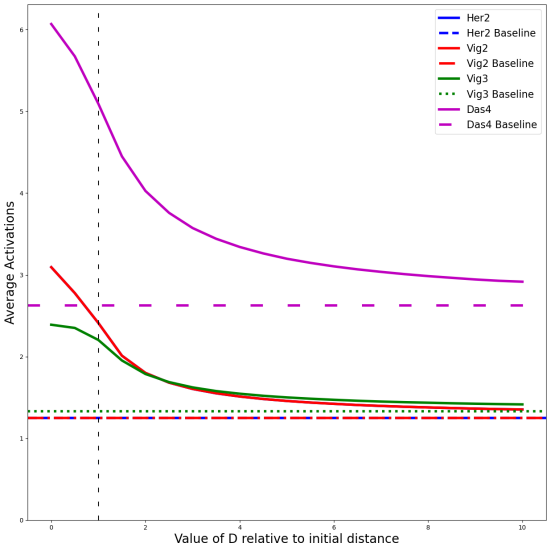
(d) Average traveled distance under the SSYNC non-rigid scheduler.



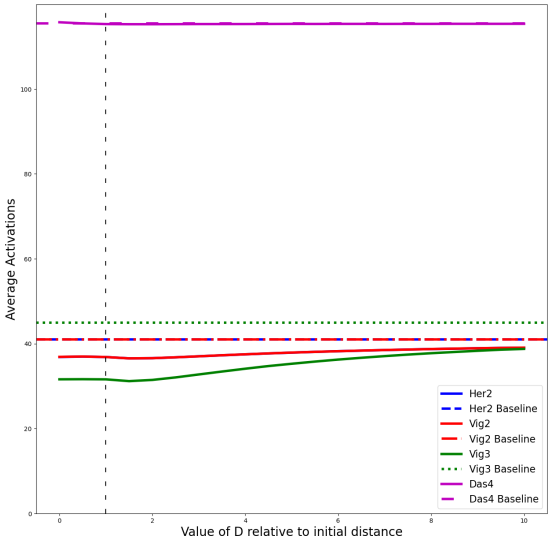
(e) Average traveled distance under the ASYNC rigid scheduler.



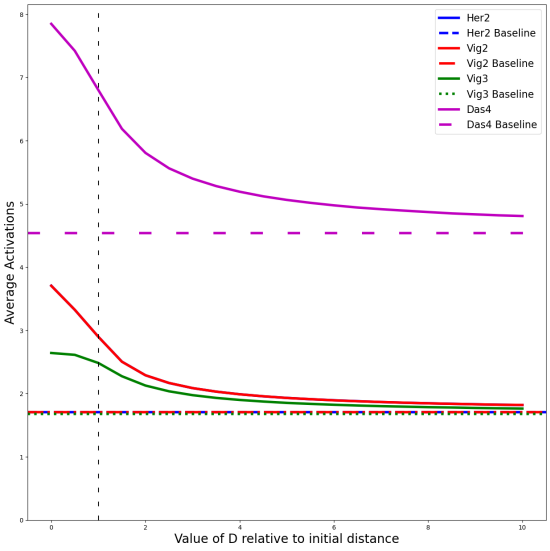
(f) Average traveled distance under the ASYNC non-rigid scheduler.



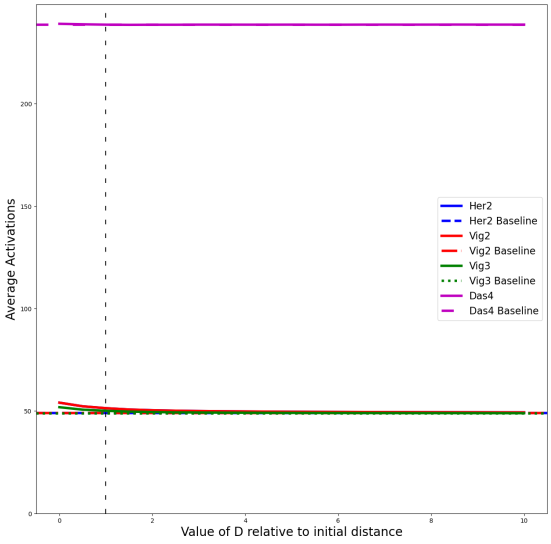
(a) Average Activations under the FSYNC rigid scheduler.



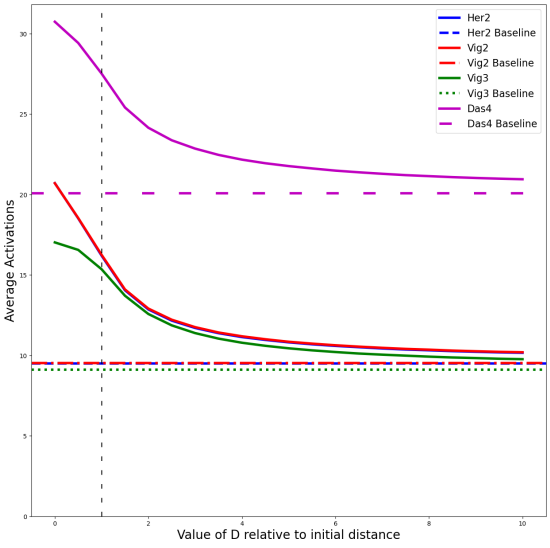
(b) Average Activations under the FSYNC non-rigid scheduler.



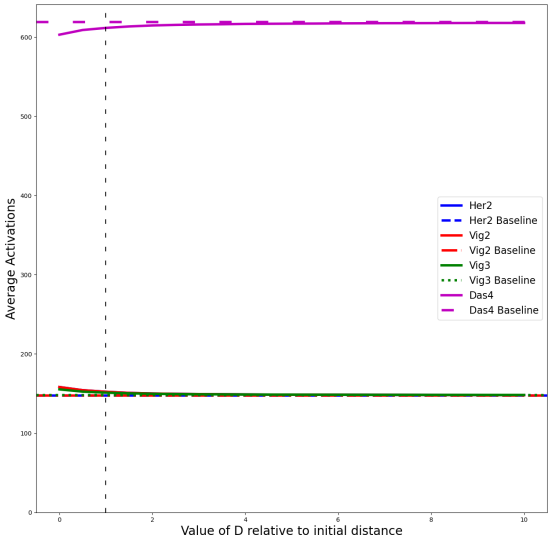
(c) Average Activations under the SSYNC rigid scheduler.



(d) Average Activations under the SSYNC non-rigid scheduler.

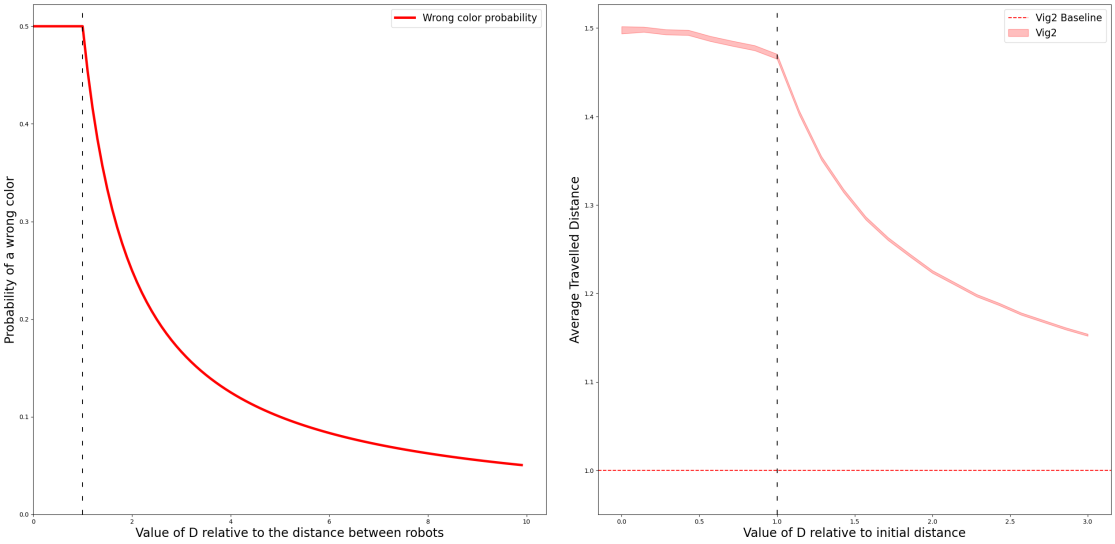


(e) Average Activations under the ASYNC rigid scheduler.



(f) Average Activations under the ASYNC non-rigid scheduler.

It should be noted that, for a given distance, the probability of an error is greater for smaller D.



(a) Probability of perceiving a wrong color depending on the error parameter D.

(b) Average traveled distance for Vig2 under the rigid FSYNC scheduler.

First we note that several algorithm and scheduler combinations confirm our last assumption. Figure 10.8a shows the probability of a color perception error depending on the parameter D. Figure 10.8b shows a zoomed plot of the average distance traveled for the Vig2 algorithm under the rigid FSYNC scheduler, and clearly displays the expected change in behavior when $D = 1$.

Then, let us look at our first assumption about Her2 and Vig2. Both algorithms appear indistinguishable for every model except distance for rigid ASYNC. This is reasonable: ASYNC is the only model involving different behavior, and rigid allows for the most adversarial counter examples.

Regarding our second assumption, the hierarchy between Das4 and the rest is always preserved. However, it appears the hierarchy between Vig3 and Vig2/Her2 is violated for greater error probability for distance in ASYNC and activations in FSYNC.

Our third assumption appears validated for rigid schedulers, as graphs show both distance and activations decrease with a decrease in error probability. However, issues arise when looking at non-rigid scheduling. While graphs appear normal for distance, several graphs, such as Das4 for ASYNC and both vig3 and Her2/Vig2 for FSYNC show an average number of activations smaller than the baseline, and increasing with a decrease of the error probability.

The average of activations in the latter FSYNC case appears to not even be monotonous, and actually crosses the baseline for Das4 in SSYNC.

We currently have no reliable explanation for these anomalies in the behavior of algorithms, and they should be investigated further.

Chapter 11

Improved Convergence and Leader Election for Faulty Visibility Sensors

Following the observations of problematic behaviors in chapter 8.6.2 and 9.2, we provide two new algorithms: a fuel efficient **Convergence** algorithm for two robots, and a **Geoleader Election** algorithm that is resilient to faulty visibility sensors.

11.1 Fuel Efficient Convergence

We provide a new algorithm (11.1) for the ASYNC **Convergence** of two robots. Our algorithm is a simplified version of the two color algorithm by Viglietta [84]. Our algorithm however ensures that no target can ever be outside of the segment between the two robots, ensuring no wasted moves, and that there exists a scheduling such that convergence is eventually achieved. It is denoted by FEC (Fuel Efficient **Convergence**, presented in figure 11.1). Our algorithm still uses two colors (BLACK and WHITE), and when observing the other robot's color, the observing robot either remains still (the 'Self' target) or goes to the computed midpoint between the two robots (the 'Midpoint' target), possibly switching its color to the opposite one.

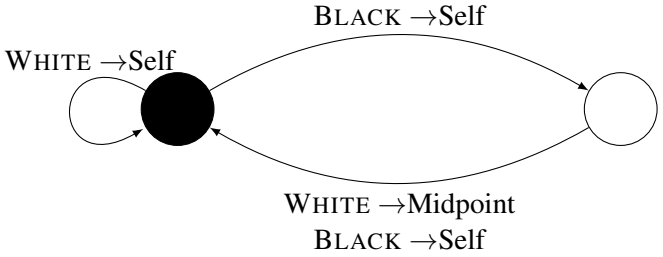


Figure 11.1 – FEC: Fuel Efficient **Convergence** Algorithm for Two Robots

Algorithm 11.1 FEC: Fuel Efficient **Convergence** Algorithm for Two Robots

```
if me.color = WHITE
    me.color ⇐ BLACK
    if other.color = WHITE
        me.destination ⇐ other.position/2
else if me.color = BLACK
    if other.color = BLACK
        me.color ⇐ WHITE
```

As a sanity check, we ran this algorithm through our simulator for one hour ($\simeq$ 30 million data points) under a randomized ASYNC scheduler and could not find a single execution where the

traveled distance was greater than the initial distance.

Theorem 11.1. *The Fuel Efficient* **Convergence** *Algorithm (11.1) guarantees the distance traveled for* **Convergence** *is never greater than the initial distance between the two robots under the ASYNC scheduler, assuming simple initial configurations* [1].

Proof. First, we see that to achieve **Convergence** with an optimal distance, robots should always be moving towards each other. So, for robots to converge using more than the initial distance, it is required that, at one point in the execution, one robot moves not towards the other robot.
In theorem 4.5, we note that a network of two disoriented robots can be simplified as a line. In that sense, the only movement that can increase the maximum **Convergence** distance is when a robot moves opposite the other robot. In other words, when robots 'switch sides'.
Let us now prove that no robot can target a robot while it is in its MOVE phase: Only the {WHITE,WHITE } snapshot can trigger a MOVE phase. Since this transition implies a change of color to BLACK at the end of the COMPUTE phase, robots that move can only be BLACK.
So, if a robot is moving, it is BLACK and the other robot, regardless of color, cannot start moving because its snapshot is different from {WHITE,WHITE }.
Furthermore, because robots switch to BLACK after moving, and can only switch to WHITE if the other robot is BLACK, no robot can execute multiple MOVE in sequence unless the other robot has executed at least a full cycle in between. So a robot cannot move multiple times while the other has pending moves.
We use the same reasoning as for proving theorem 4.4, but only with STAY, M2H.
We look at what happens after each robot completes at least one full cycle. We assume r_1 performs a LOOK, and r_2 performs k cycles before r_1 finishes its MOVE. The distance after r_1 finishes its cycle is presented in table 11.1.

	r_1 has a pending STAY	r_1 has a pending M2H
r_2 executes k STAY	X	$\left[\frac{X}{2}, X-\delta\right]$
r_2 executes 1 M2H[2]	$\left[\frac{X}{2}, X-\delta\right]$	$[0, X-2\cdot\delta]$

Table 11.1 – Distance after a full cycle of r_1 and k full cycles of r_2 with an initial distance of X

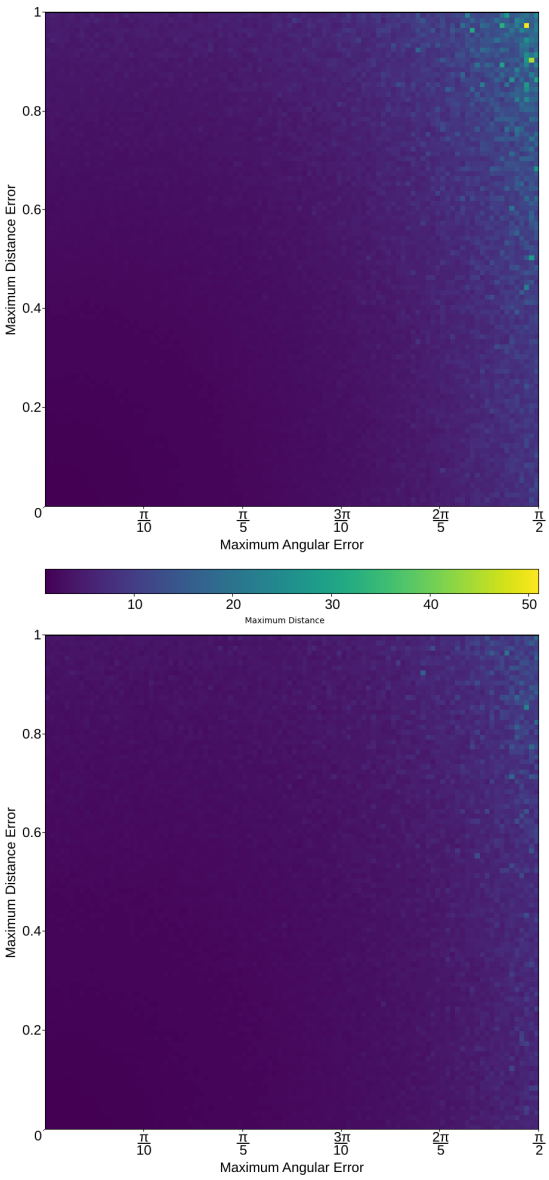
In the case of simultaneous M2H, the distance can be reduced down to zero, but robots cannot switch sides.
In both other cases where a MOVE happens, the distance is reduced at most down to half, and robots cannot switch sides.
Overall, in no cases can the robots move not towards one another, so the maximum distance traveled is always the initial distance between the two robots.

□

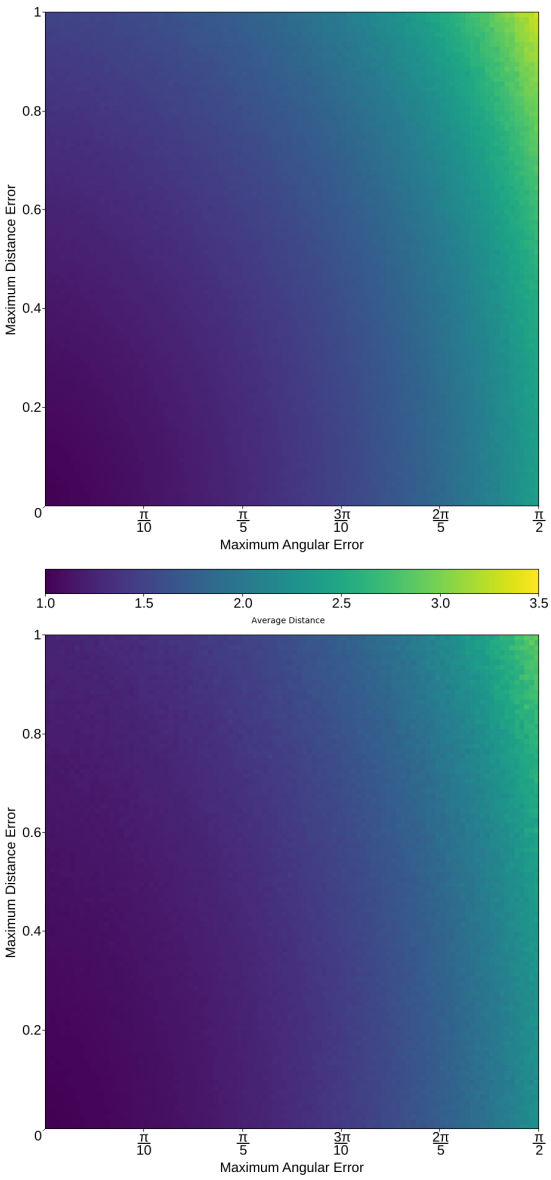
However, while the randomized scheduler we use for the simulator ensures convergence is always achieved, a rapid analysis of the algorithm shows this algorithm ensure fuel efficiency, but does not actually ensures convergence. In fact, a simple SSYNC scheduling can infinitely prevent robots from moving. This further highlights that simulations and formal proofs are complementary. We conjecture that Fuel Efficient Convergence is not actually possible for two colors, and that algorithms using three colors may even yield Fuel Efficient Rendezvous.

[1] See section 4.2.1
[2] As explained above, moving a second time requires at least a full cycle from the other robot.

We also compare the resilience of this algorithm to vision error with the center of gravity algorithm in figures 11.2a and 11.2b, which shows this algorithm is also slightly more resilient to vision errors.



(a) Maximum Distance Traveled by COG (top) and FEC (bottom)



(b) Average Distance Traveled by COG (top) and FEC (bottom)

11.2 Error Resilient Geoleader Election

The **Geoleader Election** algorithm by Canepa and Gradinariu Potop-Butucaru [19] was *not* designed under the assumption that the visibility sensors could be prone to errors. In this section, we use this knowledge to create a new, error-resilient, version of this algorithm, using our simulation framework.

11.2.1 Geoleader Election for Four Robots

One intuitive way of building a fully resilient algorithm for **Leader Election** could be based on robots computing the bounds of the error zone. While this seems feasible for a 3-robot election, it becomes far less trivial for four robots or more.

Figures 11.3 through 11.8 show the result of attempting to elect a **GEOLEADER** using algorithm 5.2 by Canepa and Gradinariu Potop-Butucaru [19] which should fail whenever two robots are identically close to the center of the smallest enclosing circle. However, as we have shown, such cases are statistically impossible with perfect sensors and simply become a small subset of the error points of error-prone sensors.

Robots r_1 and r_2 are fixed at coordinate $(-0.5, 0)$ and $(0.5, 0)$, respectively. Robot r_3 has a fixed location for each image, on a grid in the lower left quarter of the image. Symmetries of the network allow us to easily extrapolate results for the remainder of the positions of r_3. The position of r_4 is chosen at random, and each point show the result for a given position. The error is absolute with $err = 0.001$.

As before, colors red, green, and blue denote that robot r_1, r_2 and r_3 are the chosen **GEOLEADER**, respectively. Color cyan denotes that robot r_4 is the chosen **GEOLEADER**, and yellow denotes that two different **GEOLEADER** robots have been elected due to sensor error. Similarly to previous simulations, each image contains one million points.

As can be seen from the following figures, computing precisely the bounds of the error zone is extremely complex in practice, as the formula would be different and more complex the larger the network is.

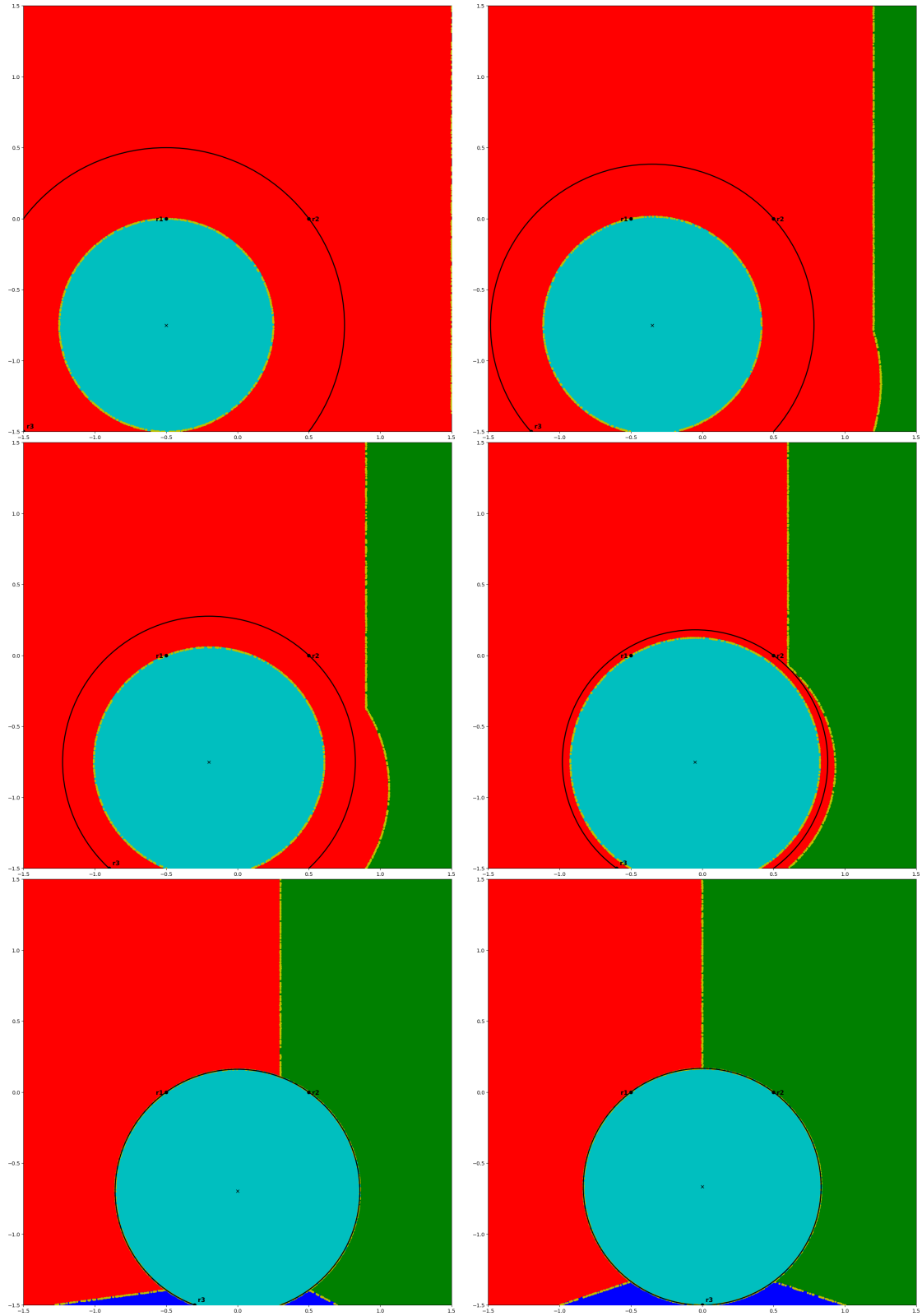


Figure 11.3

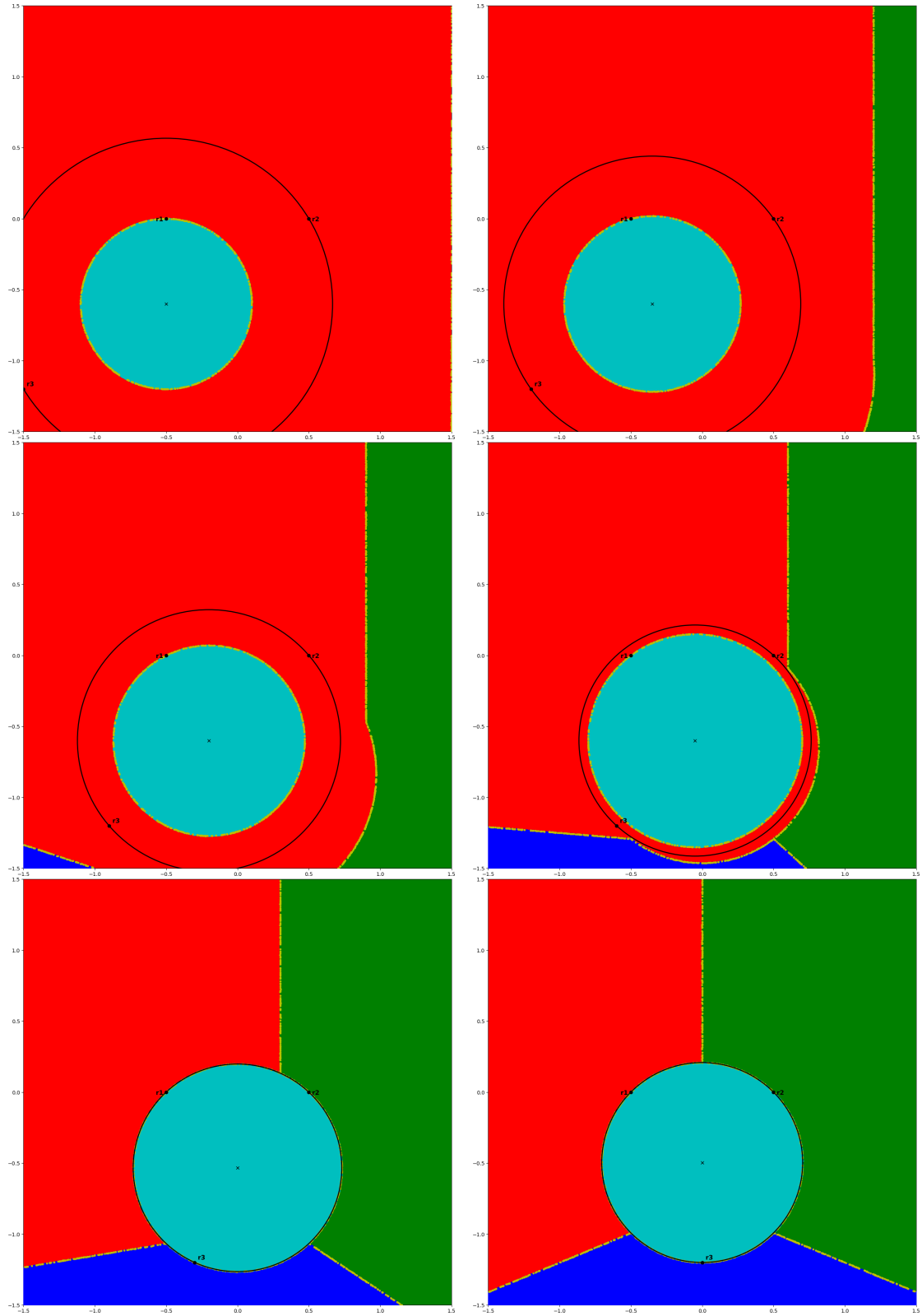


Figure 11.4

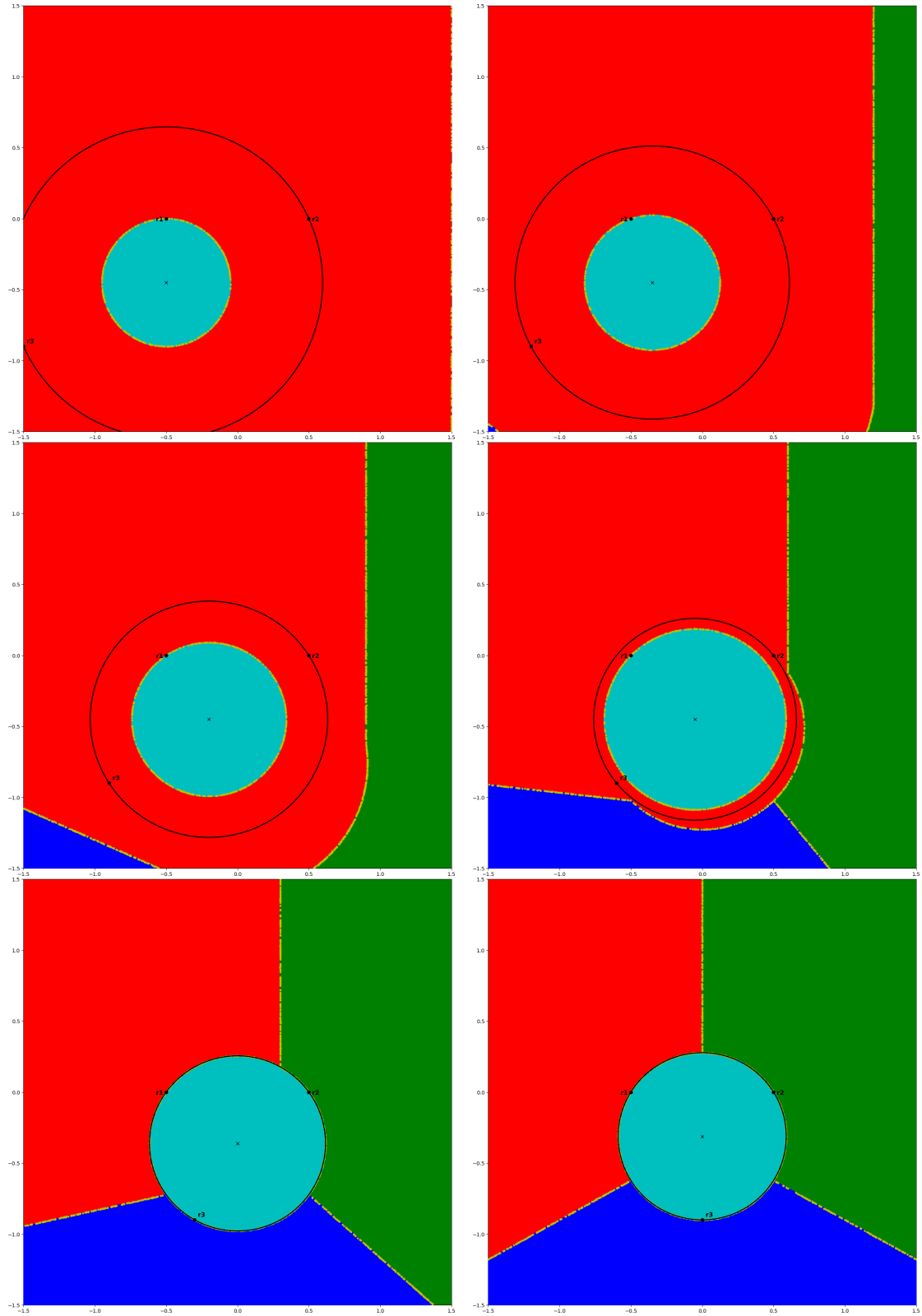


Figure 11.5

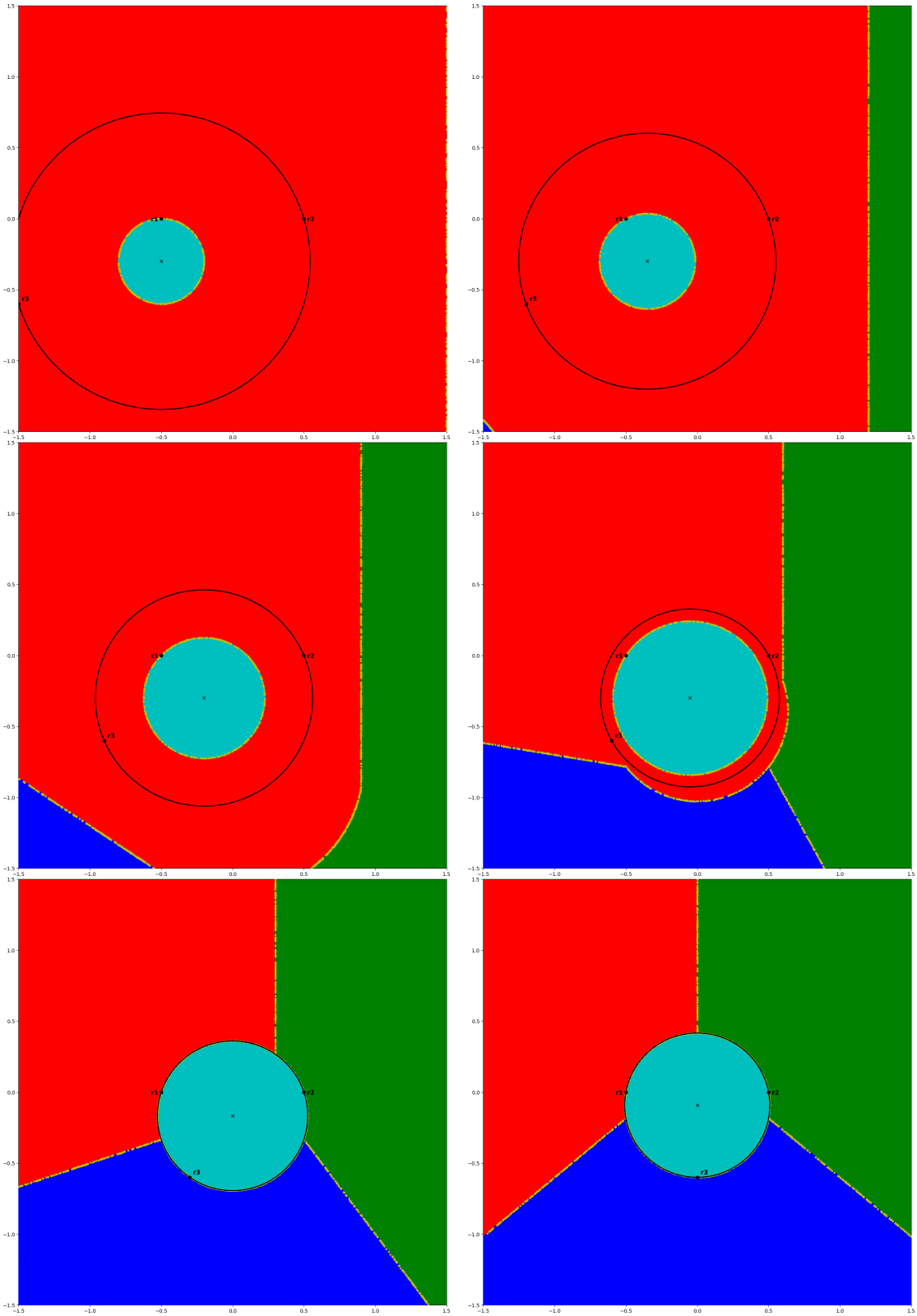


Figure 11.6

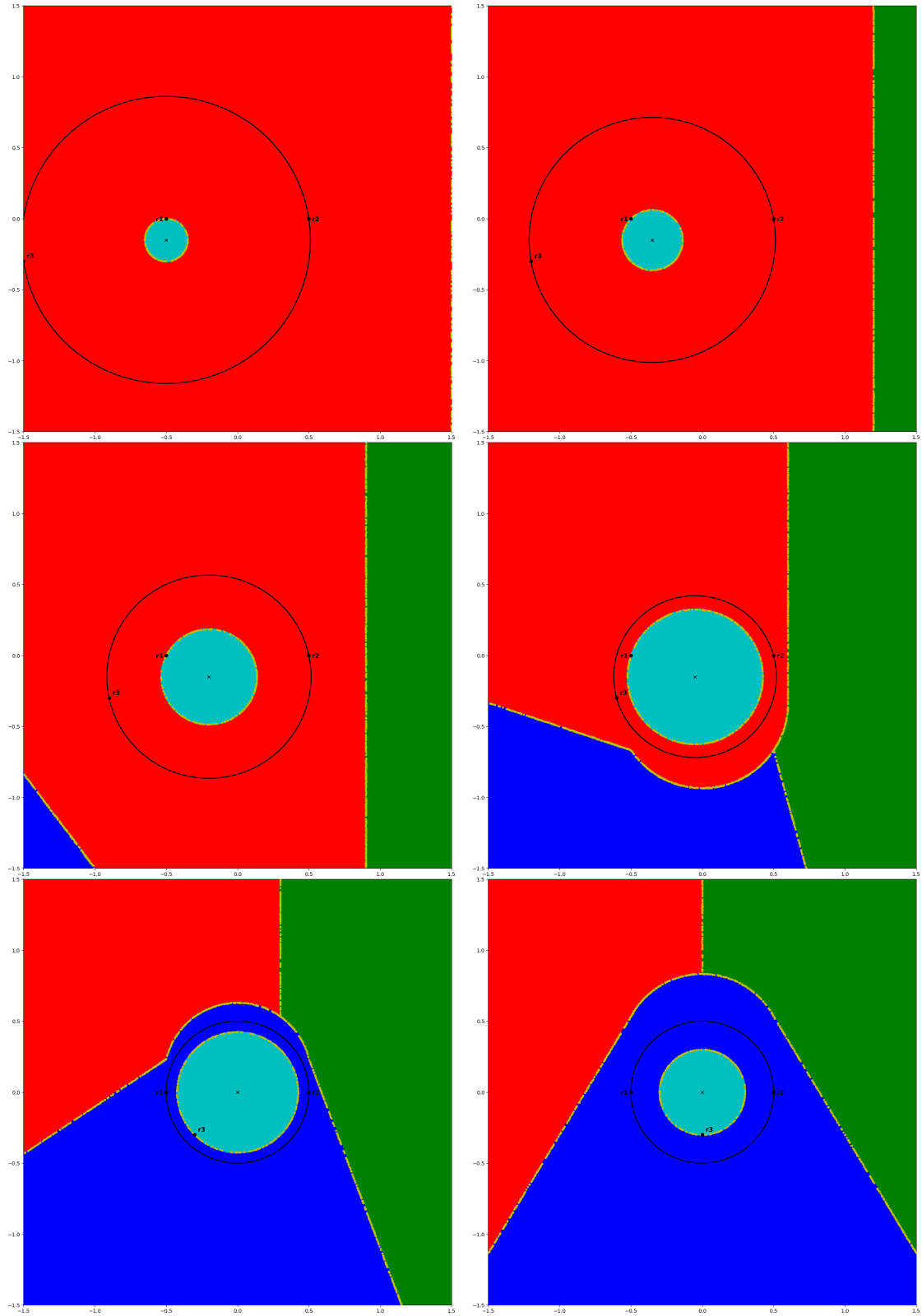


Figure 11.7

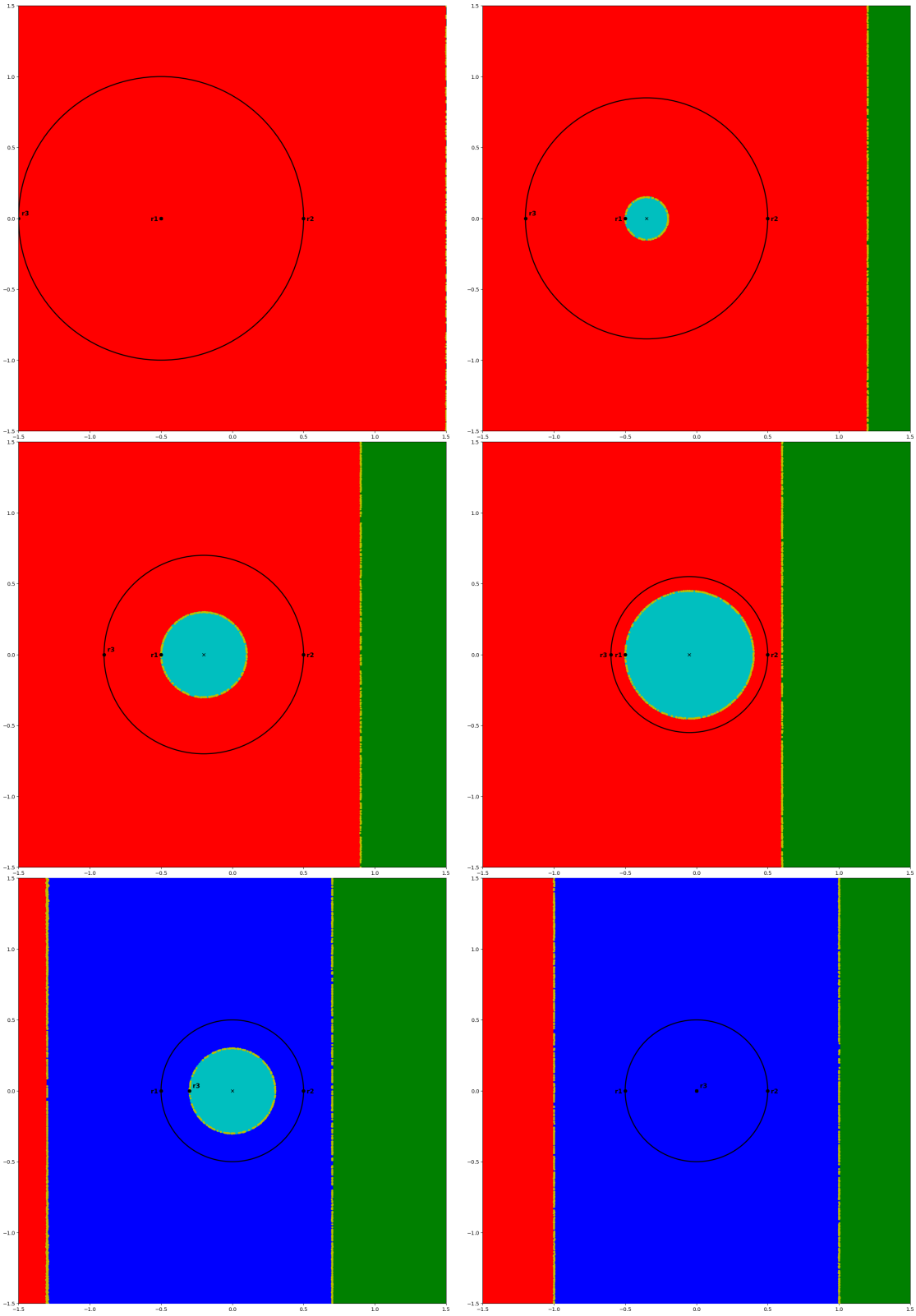


Figure 11.8

11.2.2 Proposed Algorithm

In chapter 9.2, we used the framework to detect failed elections caused by visibility sensor errors. Since mobile robots are able to run any algorithm during their COMPUTE phase, then they can also run the simulation framework to do precisely that.

The improved algorithm relies on the knowledge of the vision error model and its upper bounds to simulate random errors in a robot's position and snapshot and determine whether there exists a possibility of the other robots electing different **LEADER** robots.

Note that absolutely knowing that the election cannot fail (*i.e.*, the election cannot yield two different **LEADER** robots for two different robots) would require checking the entire surface of possible errors, which is not feasible in practice. So, we assume that robots perform a finite number of trials and decide accordingly.

Each robot internally simulates a position error for each robot in its snapshot within the known margins, performs a simulated election for each robot in its snapshot, and checks for discrepancies in the resulting **LEADER** robots. This is repeated with new random errors for a given number of tries, similar to a Monte-Carlo approach.

Once a robot believes the election process can succeed, it chooses the **LEADER** normally.

Otherwise, it picks a random direction and distance, and performs a MOVE to "scramble" the network.

This process repeats until all robots believe the election can succeed.

This is detailed in algorithm 11.2.

Algorithm 11.2 Reliable **Leader Election** algorithm

$L = self.\text{COMPUTE}('LeaderElection')$
$my\_network = self.snapshot \cup self$
$counter = 0$
while $counter < nb\_tries$ **do**
  for r_1 in $my\_network$ **do**
    $r_v = r_1$
    Change $r_v.x$ and $r_v.y$ randomly according to error parameters
    $r_v.snapshot = my\_network / \{r_1\}$
    for r_2 in $r_v.snapshot$ **do**
      Change $r_2.x$ and $r_2.y$ randomly according to error parameters
    $L_v = r_v.\text{COMPUTE}('LeaderElection')$
    if $L \neq L_v$
      Move randomly
      Exit
  $counter \mathrel{+}= 1$
L is elected **LEADER**

We now perform simulations using this algorithm.

Each point is sorted according to the following:

- If no robot detects a possible error, it is a valid point.
- If at least one robot has detected a possible error, and decided to move as a result, it is a detected possible error point.
- If no robot moves, but two robots have different **LEADER** robots, it is an undetected error point.

We measure the proportion of undetected error and possible error points for nb_{tries} between 0 and 30. Results are presented in figure 11.9.

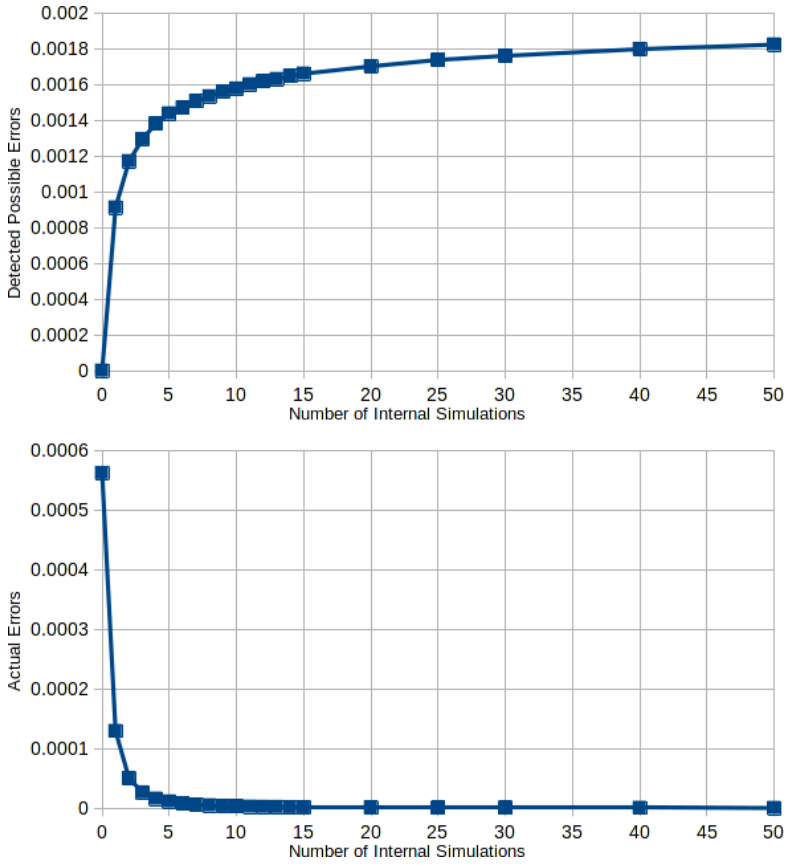


Figure 11.9 – Performance of the Error-Resilient Election Algorithm
$err = 0.001$

Note that the number of undetected error points, while decreasing, has not reached zero under our testing conditions. Also, using a single internal simulation typically results in a $\sim 80\%$ reduction in the number of undetected error points. Using 10 internal simulations resulted in a reduction of 99.5% of undetected error points.

Which number of internal simulations is the best suited would depend on both the speed and reliability requirements.

Importantly, we notice that, were we to choose an error model and error bounds such that it models the possible errors of representing real numbers using limited precision floats, then this particular algorithm, when used with an infinitely large, similar to $\mathbb{R}^2$ number of random tries, can be made to reliably detect anomalies due to the errors of evolving in the continuous plane, yet only perceiving a discretized plane. Actually, me make the conjecture that this algorithm can be adapted to allow any algorithm that makes decisions based on robot locations to function in a perceived discretized plane.

Furthermore, using this algorithm allows us to reduce the size of a snapshot to a finite, storable amount to realistically use the **SyncSim** protocol [31], and fully simulate the FSYNC scheduler in $\mathcal{LUMINOUS}$ ASYNC.

Conclusion: Real World Performance

In this chapter, we introduce a modular framework designed to simulate mobile robots for any given setting.

We discuss the limitations and constraints of this approach, and use it to compute the maximum distance traveled, or fuel efficiency, of multiple algorithms in several settings, with interesting results. In particular, we note that the algorithm by Izumi *et al.* [61] can lead to an unbounded increase in distance before eventually gathering. Similarly, the center of gravity algorithm is inherently sub-optimal for $n > 2$ robots, and robots should use an algorithm based on the geometric median instead.

We then use this framework to simulate inaccurate sensors for mobile robots and verify the behavior of **Convergence** and motion based **Leader Election** under this new model. We also introduce errors in the perception of colors for $\mathcal{LUMINOUS}$ robots performing state-of-the-art two-robot **Gathering**.

Finally, we designed two new algorithms. The first one is designed to perform two-robot **Convergence** under the ASYNC scheduler with optimal fuel efficiency. The second algorithm uses the simulator itself to allow robots to solve motion based **Leader Election** with inaccurate sensors. The latter can be adapted to allow for decision making algorithm, such as **Leader Election**, to function using discretized snapshots, and so, to use the **SyncSim** protocol to simulate the FSYNC scheduler in $\mathcal{LUMINOUS}$ ASYNC.

Overall, this framework achieves its planned objective of being both easy to use and able to produce useful results for researchers. As a test, we timed the full implementation, and testing in FSYNC, SSYNC and ASYNC, of the Obstructed Token Transmission algorithms 7.1 and 7.2 to require less than an hour, including basic network monitoring.

The source code and instructions for our simulator are provided in the appendix and at the following repository: `https://github.com/UberPanda/PyBlot-Sim`

Chapter 12

Conclusion: Networks of Realistic Robots

12.1 Our contributions

The goal of this thesis is to, first, survey and analyze the current work done by the distributed robotics community to find the more realistic variations of the standard $\mathcal{OBLOT}$ model [83]. Then we aim to further investigate existing, and develop new such variations to decide which approach should be used in the long term to bridge the realism gap between mobile robots and actual robots.

We first perform an extensive investigation on state-of-the-art models designed for more realistic perception, movement, communication and synchronization.

We develop a new, optimal algorithm for 2-robot **Gathering**. However, because of major difficulties encountered when attempting to prove this algorithm with pen and paper, we build and prove a complete model checking framework for 2-robot **Gathering** in the continuous plane based on the SPIN model checker. In the process of proving the framework, we prove several fundamental results, including that an algorithm for oblivious robots is not necessarily self-stabilizing under the ASYNC scheduler. We confront known state-of-the-art **Rendezvous** algorithms and find results consistent with the literature. We also introduce a new model for colors which matches the common notions of *safe*, *regular* and *atomic* registers and test a **Rendezvous** algorithm for regular lights. Similarly, we use lights to build more robust **Leader Election** algorithms, which allow for stricter constraints of safety and preventing scheduler bias on the election.

We then design a new vision model for mobile robots, named *Uncertain Visibility*, which introduces a vision adversary for sensors to register false negatives, *i.e.* not see robots that are actually there, and prove tight bounds under this new model for the problems of n-robot **Gathering**, **Uniform Circle Formation**, **Leader Election** and $\mathcal{LUMINOUS}$ **Rendezvous**. We also focus on the already existing obstructed visibility, or opaque robot model, and define a new problem, **Obstruction Detection**, which requires robots to not move, and compute which visible robot is obstructing them from seeing another robot. However, after proving several fundamental results and two unsuccessful attempts at solving the problem, including an algorithm based on token transmission, we acknowledge the massive difficulty inherent to both the problem and the model itself.

Because of how difficult working with more complex algorithms under realistic models turns out to be, we decide to change our approach: we develop a framework for Monte-Carlo simulations of mobile robots from the ground up. This framework is modular which allows us to simulate any robot model, scheduler or algorithm with minimal effort. This simulator is **not** a model checker and has known limitations. Because of these limitations, finding no failed executions does not immediately guarantee the algorithm is correct, and finding failed

executions does not imply the algorithm fails for a given model. As such, it should currently be viewed as a replacement for researcher *"intuition"*, and used to look for unexpected behavior in mobile robot networks that would be then verified using formal techniques.

For instance, this simulator allows us to notice that the **Rendezvous** algorithms for unreliable compasses by Izumi *et al.* [61] do not guarantee a consistent decrease in distance throughout the execution, and can actually increase the distance between two robots with no upper bounds, before decreasing to zero in a few cycles. This behavior is easy to understand once a researcher knows to look for it, but is not easily found by simply studying the algorithm. Our pen and paper analysis also fails to detect an inefficient execution of the three-color **Rendezvous** algorithm by Viglietta [84], which gathers robots using a full initial distance more than necessary, under the ASYNC scheduler, which further proves our point. We notice that using the Center of Gravity algorithm for **Gathering** is not optimal in distance, as the optimal point is the geometric median, which is known to not be computable.

We then use this simulator to implement and test errors in vision. We test the Center of Gravity algorithm for **Rendezvous** with errors in perceived distances and angles, which lead us to assume that angles are more sensitive to errors than distances. Similarly, we demonstrate that **Geoleader Election** is not possible in this vision model. We test state-of-the-art $\mathcal{LUMINOUS}$ algorithms against a model which allows for distance-dependent errors in color perception, and find that algorithms can be substantially delayed using this model. Some details in behavior in this error model remain to be explained.

Finally, we introduce another two algorithms: the first algorithm uses two colors to ensure ASYNC **Convergence** for two robots and guarantees the distance traveled is minimal ; the second allows for **Leader Election** with errors in vision: robots use the simulator itself to verify for possible errors in the election and move randomly if an error is detected. This particular design philosophy can be used to adapt some algorithms to function in a continuous setting using discretized snapshots, and therefore can be used to realistically implement the **SyncSim** protocol and simulate a FSYNC scheduler in $\mathcal{LUMINOUS}$ ASYNC.

12.1.1 Published Work

Our two-color algorithm for ASYNC **Rendezvous** presented in chapter 2.1 was published with its original proof at ICDCN 2018 [53]. An extended version is currently under review for publication in TCS.

Our model-checking system for verifying **Rendezvous** algorithms presented in chapter 3.2 was first published as a brief announcement at DISC 2019 [33]. The full version was published at SRDS 2020 [34].

Our new model for uncertain vision presented in chapter II was first published at SIROCCO 2019 [55]. The full version was accepted for publication in PPL [56].

Our preliminary results for **Obstruction Detection** presented in chapter 6.5 were presented at the IEICE COMP / IPSJ-AL 2018 workshop [54].

12.2 Short-Term Perspectives

12.2.1 Analyzing More Models and Algorithms

As we discuss in the previous section, our simulator is modular to allow for use for any given algorithm and model. So it seems logical that it should, ideally, implement every existing model and test all major algorithms in the literature. In particular, we should analyze existing **Mutual Visibility** algorithms to confirm a proper model for obstructed visibility and begin testing of **Obstruction Detection** algorithms in this model.

12.2.2 Gathering of n Robots Using Two Colors

As a continuation to the 2-color algorithm for ASYNC 2-robot **Gathering** provided in chapter 2.1, we design a similar 2-color algorithm proposal for SSYNC $n \geq 3$-robot **Gathering**, using strong global multiplicity detection.

This algorithm proposal functions as follows:

- Activated robots perform a LOOK.
- Robots look through their snapshot for the position with the most robots, *i.e.* the largest tower.
- If there is a single largest tower, it is the destination for their MOVE.
- If it is not unique, they compare these largest tower to find the ones with the most robots in the BLACK color. This is the largest BLACK tower.
- If there is a single largest BLACK tower, it is the destination for their MOVE.
- Otherwise:
  - if the robot is WHITE, it moves towards the centroid of the largest BLACK towers and switches its color to BLACK.
  - if the robot is BLACK, it moves towards the closest largest BLACK tower and switches its color to WHITE.

Algorithm 12.1 SSYNC n-Robot **Gathering** with Two Colors: Proposal

```
largest-tower = [position with largest number of robots]
if largest-tower is unique
    me.destination = largest-tower
else
    largest-BLACK-tower = [largest-tower with largest number of BLACK robots]
    if largest-BLACK-tower is unique
        me.destination = largest-BLACK-tower
    else
        if me.color = WHITE
            me.destination = centroid of largest-BLACK-tower
            me.color ⇐ BLACK
        else if me.color = BLACK
            me.destination = closest largest-BLACK-tower
            me.color ⇐ WHITE
```

For similar reasons, this proposal is extremely tricky to prove.

Our approach with the SPIN model checker is not immediately usable as the fundamental theorems do not hold for $n \geq 3$ robots.

Using our simulator, we tested this proposal for all values of n between 2 and 20 under the ASYNC and SSYNC schedulers. No counter-examples were found, which encourages us in trying to formally prove its validity.

12.3 Long-Term Perspectives

12.3.1 A Proven Simulator

As previously stated, while interesting for researchers, our simulator is not a tool for formal proofs. However, one could also argue that in its current state, we have not proven that the simulator actually simulates mobile robots, even within our degraded hypotheses. We believe that the simulator itself should be formally proven to match the model of mobile robots it claims to simulate. Note that the usefulness of this proof would be limited, as the addition of any new module may require proving the entire simulator again.

12.3.2 Stronger Simulator Adversaries

In its current form, the simulator relies on a randomized adversary for scheduling, non-rigid behavior, and simultaneous LOOK and MOVE for ASYNC. This is obviously very limiting, as seen for the Fuel Efficient Convergence algorithm. In practice, scheduling only requires the adversary to choose among a finite set of possible activations, while the other two choices, in theory, require choosing a single value in a continuous interval[1]. From the point of view of the adversary, the algorithm could be seen as a game adversary, defeated when it fails to solve a given problem. Recent advances in artificial adversaries using Machine Learning may be used to create a practical adversary designed to defeat an algorithm under the 'game rules' of mobile robots. In particular, techniques of Monte-Carlo Tree Search for scheduling and Gradient Descent for motion seem the most promising approach.

12.3.3 Obstruction Detection

There is currently no proven algorithm to solve the **Obstruction Detection** problem for a non-line network. As explained in chapter 6.5, we currently have found no functional deterministic algorithm, and proving randomized, color based algorithms, is extremely tricky. However, once a proper obstructed visibility model is implemented in our simulator, it could be used to quickly eliminate invalid algorithms and massively speed-up the design process.

12.3.4 Expanding Uncertain Visibility

The current model of Uncertain Visibility is limited to the FSYNC scheduler, as it relies on the notion of rounds to be properly defined. We believe this model to be interesting enough to warrant further investigation. In particular, it should be carefully expanded to the SSYNC scheduler, which would require extreme care regarding the fairness and boundedness of the scheduler to prevent trivial counter-examples where the scheduler simply repeats a LOOK until the vision adversary removes a relevant message.

Both vision adversaries introduced rely on an adversary. A fully randomized model of Uncertain Visibility, requiring a randomized choice of the number of messages to be dropped would be the weakest model yet, and should also be rigorously studied.

Moreover, we have shown that uncertain visibility is orthogonal to asynchrony in the case of $\mathcal{LUMINOUS}$ **Rendezvous**. A possible relationship with self-stabilization may be worth investigating: self-stabilizing algorithms can tolerate a single transient fault putting the network in an arbitrary state, while uncertain visibility can be seen as a limited transient fault, but happening each round.

Finally, our analysis in this model only tests algorithms designed for the unlimited vision model. It is paramount that algorithms designed for the limited visibility model be tested. It

[1] It is in fact a finite by the nature of numerical simulations, but the size of the set can be seen as infinite for in this case.

would seem, at first, that such algorithms, as they seem to succeed against the $n \cdot (n-1)$-enemy, should trivially succeed against any uncertain adversary. However, the limited visibility model ensures symmetry in vision, while the uncertain model does not. So, if an algorithm relies on the fact that, when it sees another robot, the other robot also sees it, it might fail against weaker uncertain adversaries. And in fact, may not work against the $n \cdot (n-1)$-enemy, as it can block *up to* $n \cdot (n-1)$ vision messages.

12.3.5 Robots with Finite Memory Snapshots

As explained in chapter 10.5, our **Leader Election** algorithm for errors in vision is able to function in a continuous setting using discretized snapshots. The design philosophy behind this algorithm of using randomized tries to simulate sensor errors is not specific to the **Leader Election** problem, and it could be used for other algorithms that rely on making decisions based on the locations of robots in the network and that are sensitive to errors in perception. Building new algorithms that can use these finite snapshots allows us to use the **SyncSim** protocol [31] and simulate a FSYNC scheduler in $\mathcal{LUMINOUS}$ ASYNC and would be a major advantage for resilience to asynchrony.

Appendix A

Details and Results of the Model Checker

A.1 Movement Resolution

The movement resolution rules described in Figure 4.7 are implemented by the Promela code described in Listing A.1 below, during the MOVE_E phase of the cycle.

In Promela, the meaning of `if` and the guarded actions `guard -> action` that follow is different from other languages in the sense that, when several guards are enabled, the execution faces a non-deterministic choice and the exploration of the model checker branches into several executions to explore all enabled guards. The guard `else` is exclusive in that it is enabled only when no other guards are enabled.

Listing A.1 – Movement resolution

```
if
:: (robot[me].is_moving) ->
    local position_t new_position = position;
    assert( robot[me].pending != STAY );
    if
    :: (position == NEAR || position == SAME) ->
        if
        :: (robot[me].pending == MISS) ->
            { robot[other].pending = MISS }
                unless (robot[other].pending == STAY);
            new_position = NEAR;
        :: (robot[me].pending == TO_OTHER) ->
            { robot[other].pending = MISS }
                unless (robot[other].pending == STAY
                || position == SAME);
            new_position = SAME;
        :: (robot[me].pending == TO_HALF) ->
            if
            :: (robot[other].pending == TO_HALF)
                -> robot[other].pending = TO_OTHER
            :: (robot[other].pending == STAY)
                -> skip /* do nothing */
            :: else
                -> robot[other].pending = MISS
            fi
        :: else -> assert( false )
        fi;
    fi;
    if
    :: (position != new_position) ->
        eventPositionChange:
            position = new_position
    :: else -> skip /* do nothing */
    fi
:: else -> skip
fi;
robot[me].is_moving = false;
robot[me].pending = STAY;
```

A.2 Verified Algorithms Written in Promela

Listing A.2 – No Move Algorithm

```
inline Alg_NoMove(obs, command)
{
    command.move    = STAY;
    command.new_color = BLACK
}
```

Listing A.3 – Move to Half Algorithm

```
inline Alg_ToHalf(obs, command)
{
    command.move    = TO_HALF;
    command.new_color = BLACK
}
```

Listing A.4 – Move to Other Algorithm

```
inline Alg_ToOther(obs, command)
{
    command.move    = TO_OTHER;
    command.new_color = BLACK
}
```

Listing A.5 – Viglietta's 2 colors algorithm [84] for LC-atomic ASYNC

```
inline Alg_Vig2Cols(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.me == BLACK) ->
        if
        :: (obs.color.other == BLACK)
            -> command.new_color = WHITE
        :: (obs.color.other == WHITE)
            -> skip
        fi
    :: (obs.color.me == WHITE) ->
        if
        :: (obs.color.other == BLACK)
            -> command.move = TO_OTHER
        :: (obs.color.other == WHITE)
            -> command.move = TO_HALF;
                command.new_color = BLACK
        fi
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.6 – Viglietta's 3 colors algorithm [84] for ASYNC

```
inline Alg_Vig3Cols(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.me == BLACK) ->
        if
        :: (obs.color.other == BLACK)
```

```
            -> command.move = TO_HALF;
                command.new_color = WHITE
        :: (obs.color.other == WHITE)
            -> command.move = TO_OTHER
        :: (obs.color.other == RED)
            -> skip
        fi
    :: (obs.color.me == WHITE) ->
        if
        :: (obs.color.other == BLACK)
            -> skip
        :: (obs.color.other == WHITE)
            -> command.new_color = RED
        :: (obs.color.other == RED)
            -> command.move = TO_OTHER
        fi
    :: (obs.color.me == RED) ->
        if
        :: (obs.color.other == BLACK)
            -> command.move = TO_OTHER
        :: (obs.color.other == WHITE)
            -> skip
        :: (obs.color.other == RED)
            -> command.new_color = BLACK
        fi
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.7 – Heriban's 2 colors algorithm [53] for ASYNC

```
inline Alg_Optimal(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.me == BLACK) ->
        if
        :: (obs.color.other == BLACK)
            -> command.new_color = WHITE
        :: (obs.color.other == WHITE)
            -> skip
        fi
    :: (obs.color.me == WHITE) ->
        if
        :: obs.same_position -> skip
        :: else ->
            if
            :: (obs.color.other == BLACK)
                -> command.move = TO_OTHER
            :: (obs.color.other == WHITE)
                -> command.move = TO_HALF;
                    command.new_color = BLACK
            fi
        fi
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.8 – Flocchini's external lights 3 colors algorithm [48] for SSYNC

```
inline Alg_FloAlgo3Ext(obs, command)
{
```

```
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.other == BLACK)
        -> command.move = TO_HALF;
            command.new_color = WHITE
    :: (obs.color.other == WHITE)
        -> command.new_color = RED
    :: (obs.color.other == RED)
        -> command.move = TO_OTHER;
            command.new_color = BLACK
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.9 – Okumura's external lights 5 colors algorithm [73] for LC-atomic ASYNC

```
inline Alg_Wada5Ext(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.other == BLACK)
        -> command.move = TO_HALF;
            command.new_color = WHITE
    :: (obs.color.other == WHITE)
        -> command.new_color = RED
    :: (obs.color.other == RED)
        -> command.move = TO_OTHER;
            command.new_color = YELLOW
    :: (obs.color.other == YELLOW)
        -> command.new_color = GREEN
    :: (obs.color.other == GREEN)
        -> command.new_color = BLACK
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.10 – Okumura's external lights 4 colors algorithm [73] for quasi-self-stabilizing LC-atomic ASYNC

```
inline Alg_Oku4ColsX(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.other == BLACK)
        -> command.move = TO_HALF;
            command.new_color = WHITE
    :: (obs.color.other == WHITE)
        -> command.new_color = RED
    :: (obs.color.other == RED)
        -> command.move = TO_OTHER;
            command.new_color = YELLOW
    :: (obs.color.other == YELLOW)
        -> command.new_color = BLACK
    :: else -> command.new_color = BLACK
    fi
}
```

Listing A.11 – Okumura's external lights 3 colors algorithm [73] for non-self-stabilizing rigid LC-atomic ASYNC

```
inline Alg_Oku3ColsX(obs, command)
{
    command.move      = STAY;
    command.new_color = obs.color.me;
    if
    :: (obs.color.other == BLACK)
        -> command.move = TO_HALF;
            command.new_color = WHITE
    :: (obs.color.other == WHITE)
        -> command.new_color = RED
    :: (obs.color.other == RED)
        -> command.move = TO_OTHER;
            command.new_color = WHITE
    :: else -> command.new_color = BLACK
    fi
}
```

A.3 Compile Options

```
spin -a -DALGO=ALGORITHM
-DSCHEDULER=SCHEDULER MainGathering.pml

clang -DMEMLIM=1024 -DXUSAFE -DNOREDUCE
-O2 -w -o pan pan.c

./pan -m100000 -a -f -E -n gathering
```

A.4 Output

A.4.1 Vig2Cols in ASYNC (failure)

```
Depth=   19582 States=
    1e+06 Transitions= 3.86e+06
Memory=   146.311 t=     2.25 R=   4e+05
pan:1: acceptance cycle (at depth 2723)
pan: wrote MainGathering.pml.trail

(Spin Version 6.4.9 -- 17 December 2018)
Warning: Search not completed

Full statespace search for:
        never claim
            + (gathering)
        assertion violations
            + (if within scope of claim)
        acceptance   cycles
            + (fairness enabled)
        invalid end states
            - (disabled by -E flag)

State-vector 107 byte,
depth reached 25730, errors: 1
189620 states, stored (1.45028e+06 visited)
4046596 states, matched
5496875 transitions (= visited+matched)
34174756 atomic steps
hash conflicts:     16160 (resolved)

Stats on memory usage (in Megabytes):
   24.413       equivalent memory usage
       for states (stored*(State-vector
                 + overhead))
   17.660       actual memory usage
       for states (compression: 72.34%)
                 state-vector as stored
                 = 70 byte + 28 byte overhead
  128.000       memory used for
                 hash table (-w24)
    6.104       memory used for
                 DFS stack (-m100000)
  151.682       total actual memory usage

pan: elapsed time 3.25 seconds
pan: rate 446239.69 states/second
```

After a failed execution, SPIN provides a (very verbose) counter-example that can be investigated with the following command.

```
spin -t MainGathering.pml
```

A.4.2 Her2Cols in ASYNC (Success)

```
Depth=   16958 States=
    1e+06 Transitions= 3.84e+06
Memory=   146.115 t=
    2.26 R=   4e+05

(Spin Version 6.4.9 -- 17 December 2018)

Full statespace search for:
        never claim
            + (gathering)
        assertion violations
            + (if within scope of claim)
        acceptance   cycles
            + (fairness enabled)
        invalid end states
            - (disabled by -E flag)

State-vector 107 byte,
depth reached 17016, errors: 0
232931 states, stored (1.80493e+06 visited)
5061150 states, matched
6866078 transitions (= visited+matched)
42744752 atomic steps
hash conflicts:     30286 (resolved)

Stats on memory usage (in Megabytes):
   29.989       equivalent memory usage
        for states (stored*(State-vector
                + overhead))
   21.666       actual memory usage
        for states (compression: 72.25%)
                state-vector as stored
                = 70 byte + 28 byte overhead
  128.000       memory used
        for hash table (-w24)
    6.104       memory used
        for DFS stack (-m100000)
  155.686       total actual memory usage

pan: elapsed time 4.07 seconds
pan: rate 443471.25 states/second
```

Appendix B

Example of an Instance of the Simulator

We present a minimum working example of an instance of the simulator. It simulates executions of the Vig2 algorithm [84] in the standard $\mathcal{OBLOT}$ model with rigid motion, under the FSYNC scheduler. It monitors the number of cycles needed to complete degraded gathering[1]. Results predictably show the possible need of two full cycles in the case where both robots start in the BLACK color. For better readability, more advanced features of the simulator are not included.

[1] i.e. robots are closer than 10^{-10} with the initial distance being 1.

Listing B.1 – Robot Class File: Common/lib_robot.py

```
import Common.lib_algorithms as lib_algos

class Robot:
  def __init__(self, name, x, y, color=0):
    self.name = name
    self.x = x ## Real position in the network
    self.y = y

    self.phase = 'WAITING'

    # Available info for COMPUTE :

    self.snapshot = [] ## snapshot
    self.color = color ## color
    self.target = ()


  def LOOK(self,network,scheduler):
    self.snapshot = []
    for R2 in network:
      if self != R2:
        R2x = R2.x
        R2y = R2.y
        self.snapshot.append(Robot(R2.name, R2x, R2y, R2.color))

    self.phase = 'COMPUTING'



  def COMPUTE(self,algo):
    try:
      result = getattr(lib_algos, algo)(self)
    except:
      raise AttributeError('This algorithm does not exist in the current version (or you
          ↪ made a typo ?)')

    return result


  def MOVE(self):

    self.x = self.target[0]
    self.y = self.target[1]

    self.target = ()
    self.phase = 'WAITING'
```

Listing B.2 – Miscellaneous Functions File: Common/lib_misc_functions.py

```
from math import sqrt

def rob_dist(R1,R2): # Using Robots
  return dist((R1.x,R1.y),(R2.x,R2.y))

def dist(T1,T2): # USing Tuples
  return sqrt((T1[0] - T2[0])**2+(T1[1] - T2[1])**2)
```

Listing B.3 – Algorithms File: Common/lib_algorithms.py

```
def vig2(R1):
  R2 = R1.snapshot[0]
  if R1.color == 0:
    if R2.color == 0:
      R1.target = ((R1.x+R2.x)/2, (R1.y+R2.y)/2)
      R1.color = 1
      R1.phase = 'MOVING'
    else:
      R1.target = (R2.x, R2.y)
      R1.phase = 'MOVING'
  else:
    R1.phase = 'WAITING'
    if R2.color == 1:
      R1.color = 0
```

Listing B.4 – Scheduler File: Common/lib_schedulers.py

```
def scheduler(sched,network,algo):

  if sched == 'FSYNC':
    for R1 in network:
      R1.LOOK(network,'FSYNC')
    for R1 in network:
      R1.COMPUTE(algo)
    for R1 in network:
      if R1.phase == 'MOVING':
        R1.MOVE()

  else:
    raise Exception('Unknown Scheduler')
```

Listing B.5 – Simulation Functions File: Common/lib_sim_functions.py

```
# None Needed for this Example
```

Listing B.6 – Simulation File: Vig2.py

```
from random import SystemRandom, choice, uniform
from math import sqrt, pi, cos, sin
from Common.lib_robot import Robot
from Common.lib_schedulers import scheduler
from Common.lib_misc_functions import rob_dist

_sysrand = SystemRandom()

########################################################

SSTAB = True
COLOR_RANGE = range(2) # Total number of colors.

########################################################

simus = 0
act_list = []

while simus < 1000000:
  simus += 1

  # Network initialization

  network = []
  network.append(Robot('r1',0,0,0))

  ang = uniform(-pi,pi)
  network.append(Robot('r2',cos(ang),sin(ang),0))

  for R1 in network:
    if SSTAB == True:
      R1.color = choice(COLOR_RANGE)

  ## Beginning of the simulation

  steps = 0
  while True:

    scheduler('FSYNC',network,'vig2')

    ## Live state monitoring

    steps += 1

    ## Victory condition

    if rob_dist(network[0],network[1]) < 10**(-10):
      break

    ## Defeat condition

  act_list.append(steps)


print('FSYNC R Vig2')
print('Min : ' + str(min(act_list)))
print('Max : ' + str(max(act_list)))
print('Avg : ' + str(sum(act_list) / len(act_list)))
```

Appendix C

Details of Color Perception Error

This appendix contains the detailed results of our simulations for the color perception error model.
Each figure shows the 99.9% confidence interval.
Das4 is shown top-left, Vig3 top-right, Vig2 bottom-left, Her2 bottom Right.

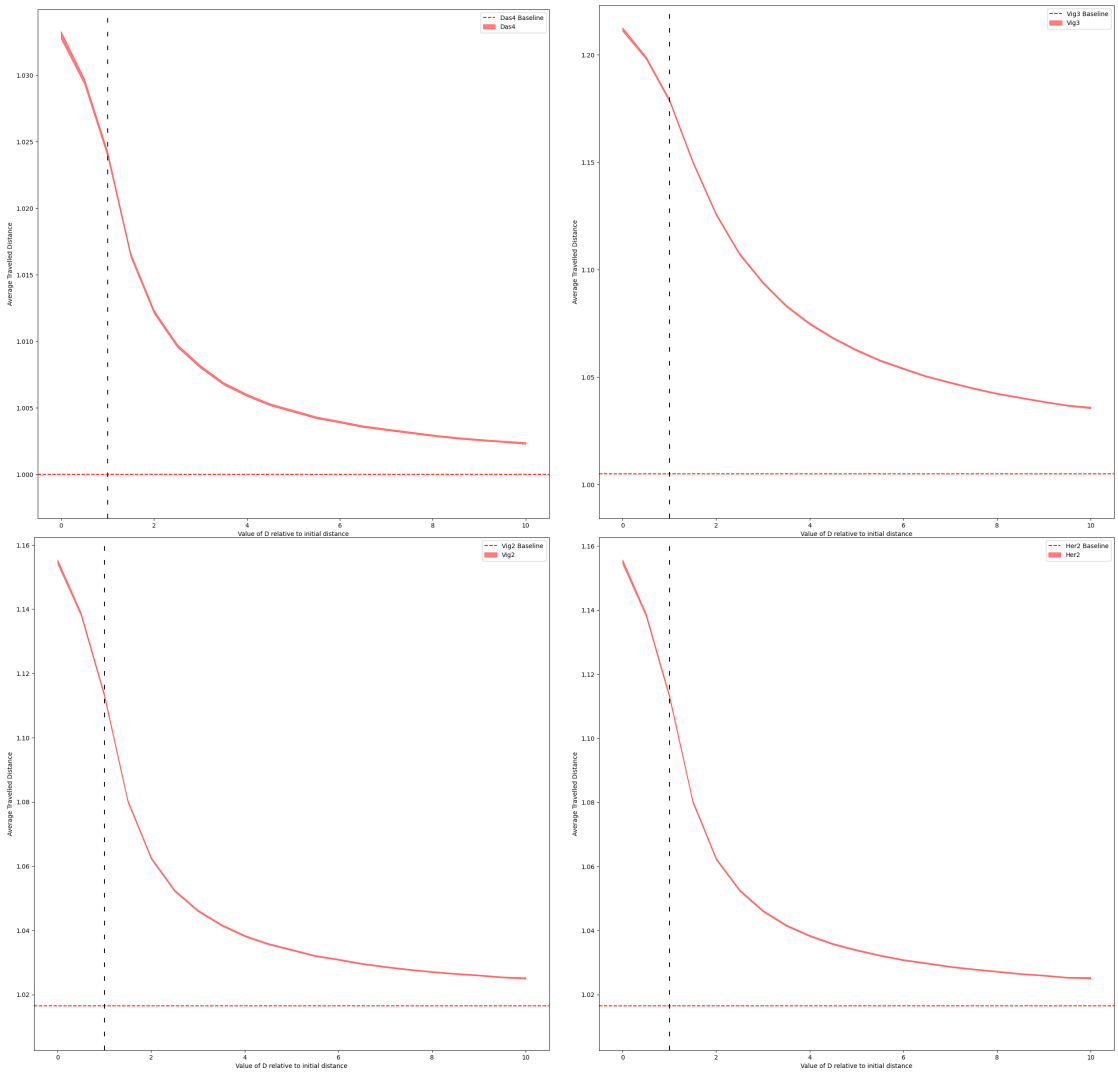


Figure C.1 – Distance for non-rigid ASYNC

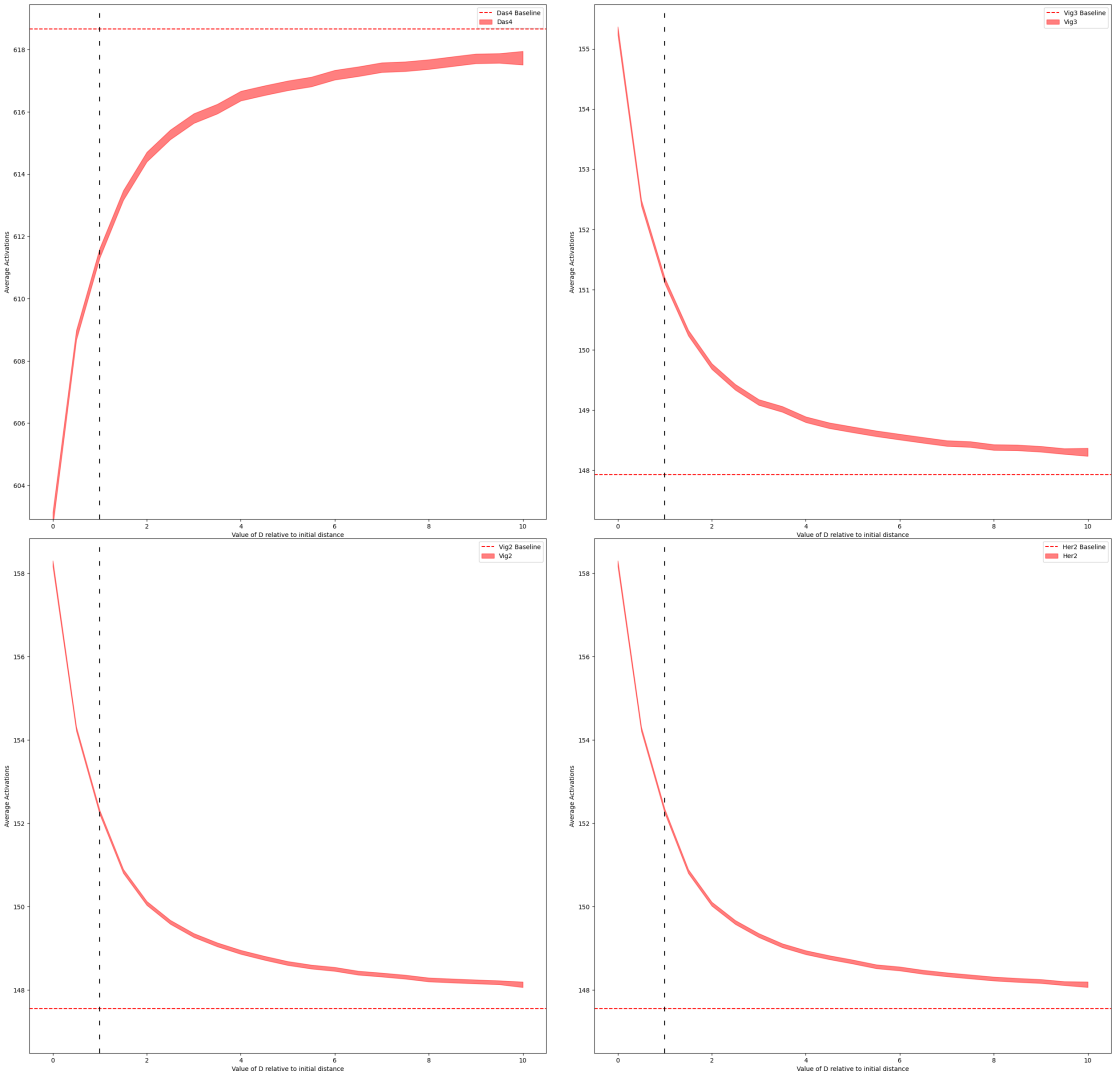


Figure C.2 – Activations for non-rigid ASYNC

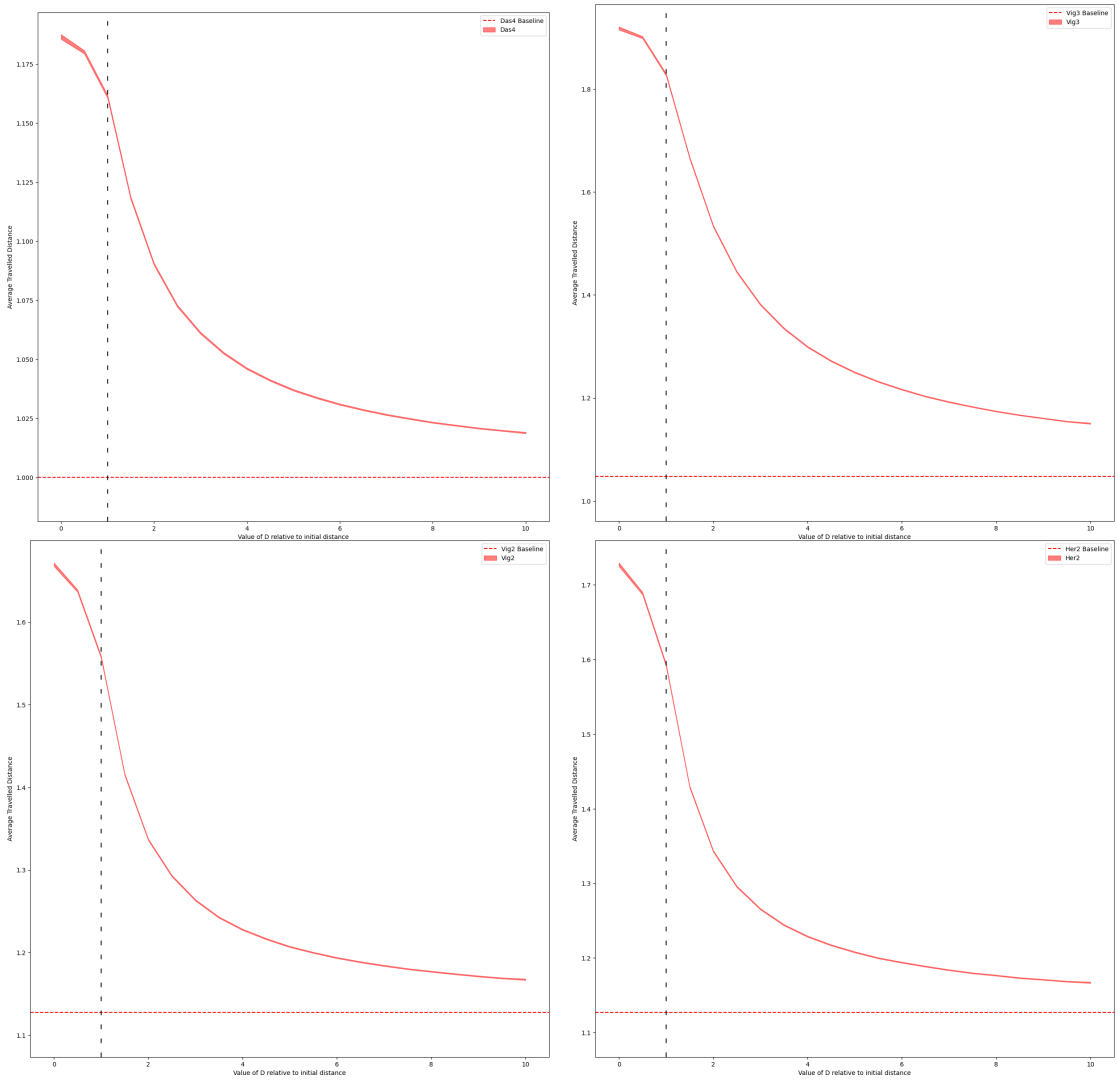


Figure C.3 – Distance for rigid ASYNC

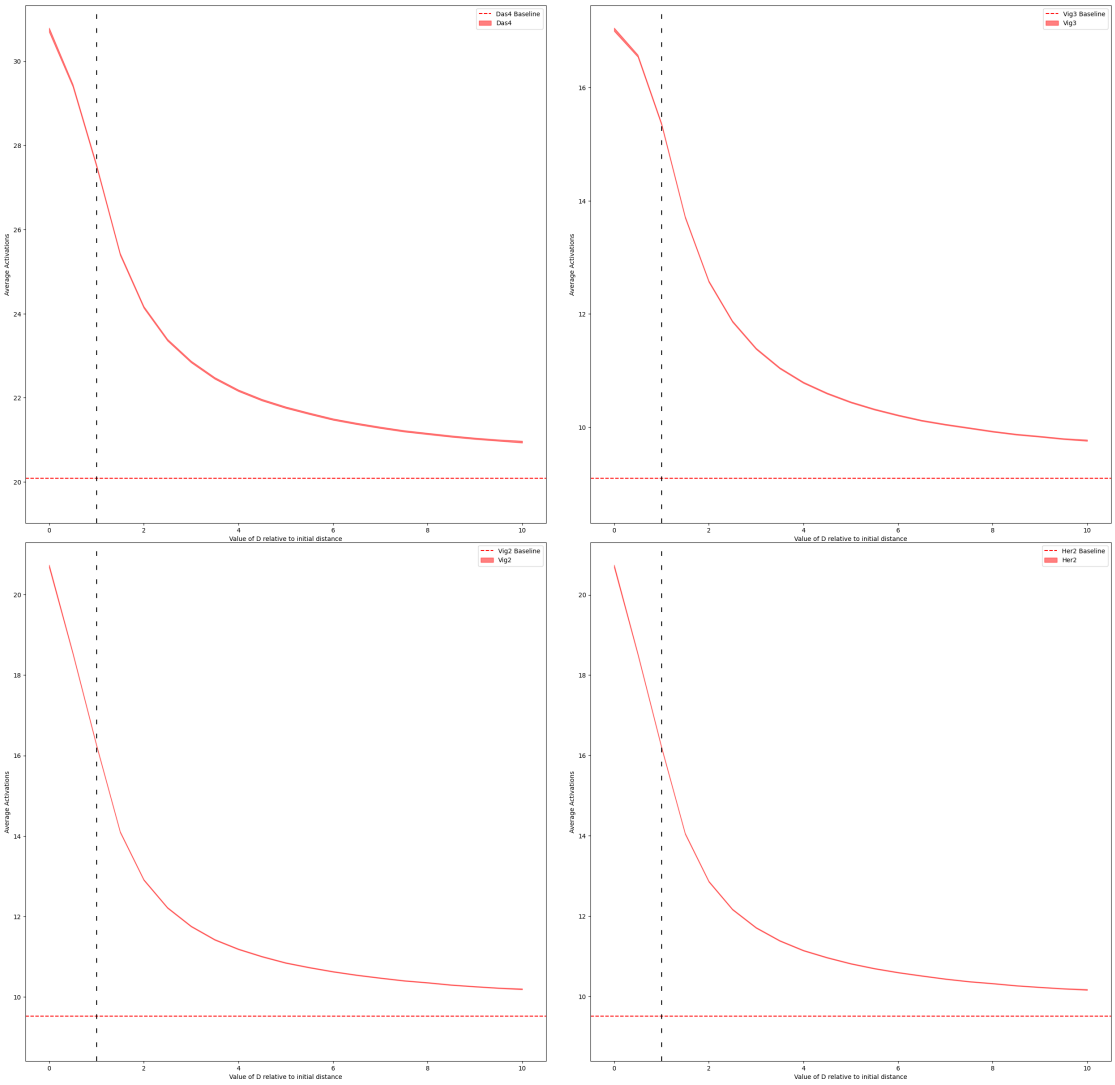


Figure C.4 – Activations for rigid ASYNC

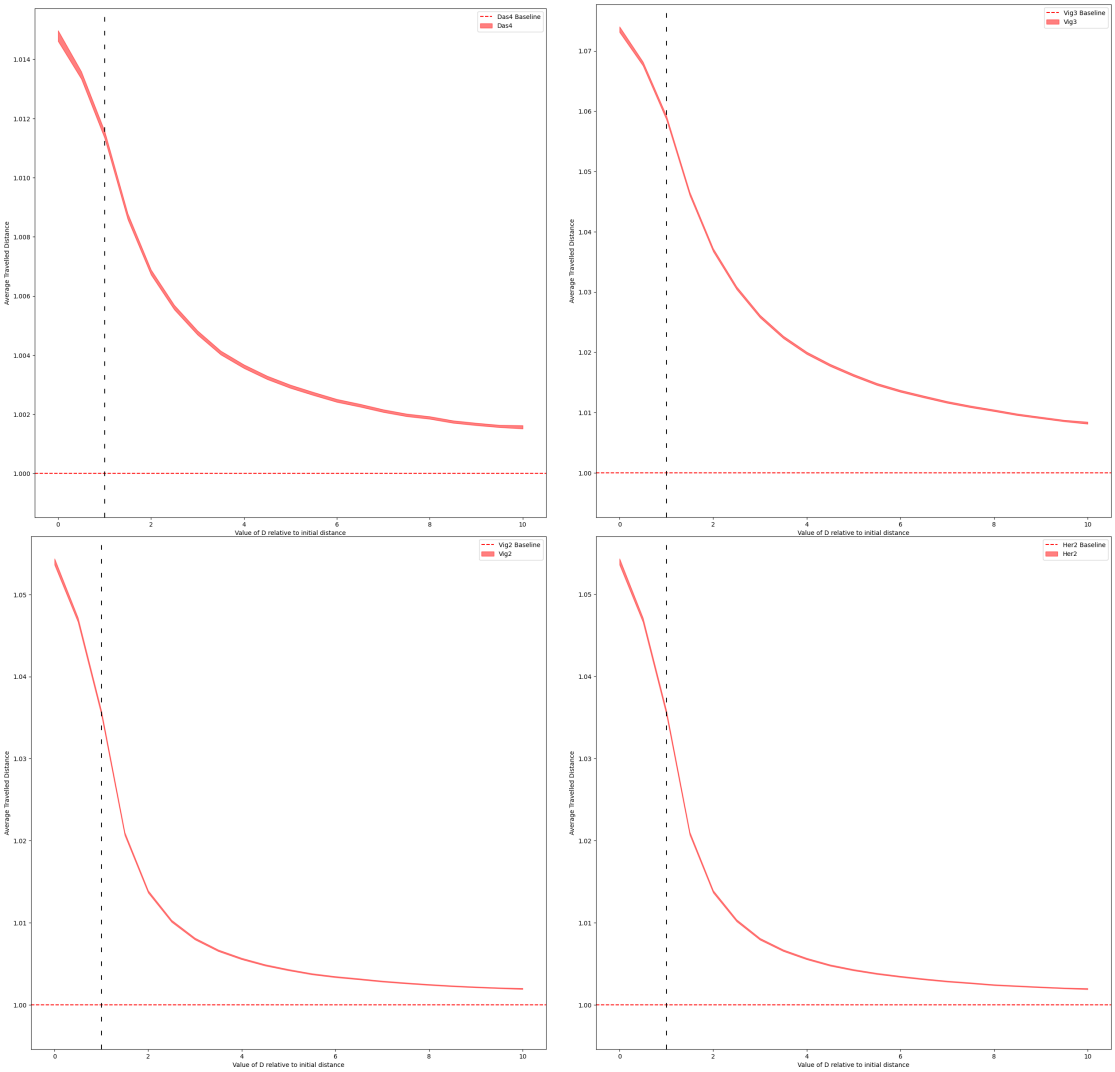


Figure C.5 – Distance for non-rigid SSYNC

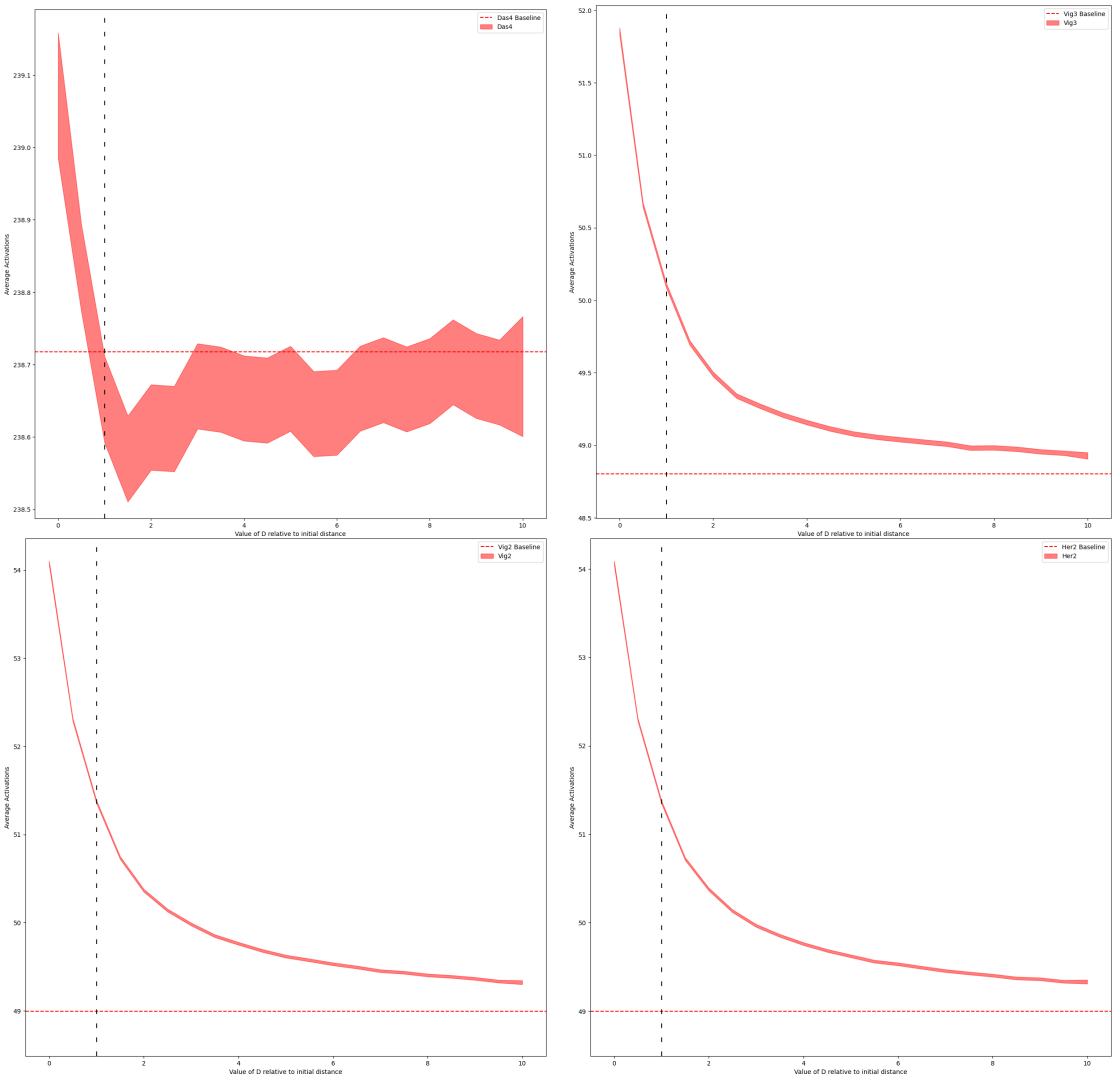


Figure C.6 – Activations for non-rigid SSYNC

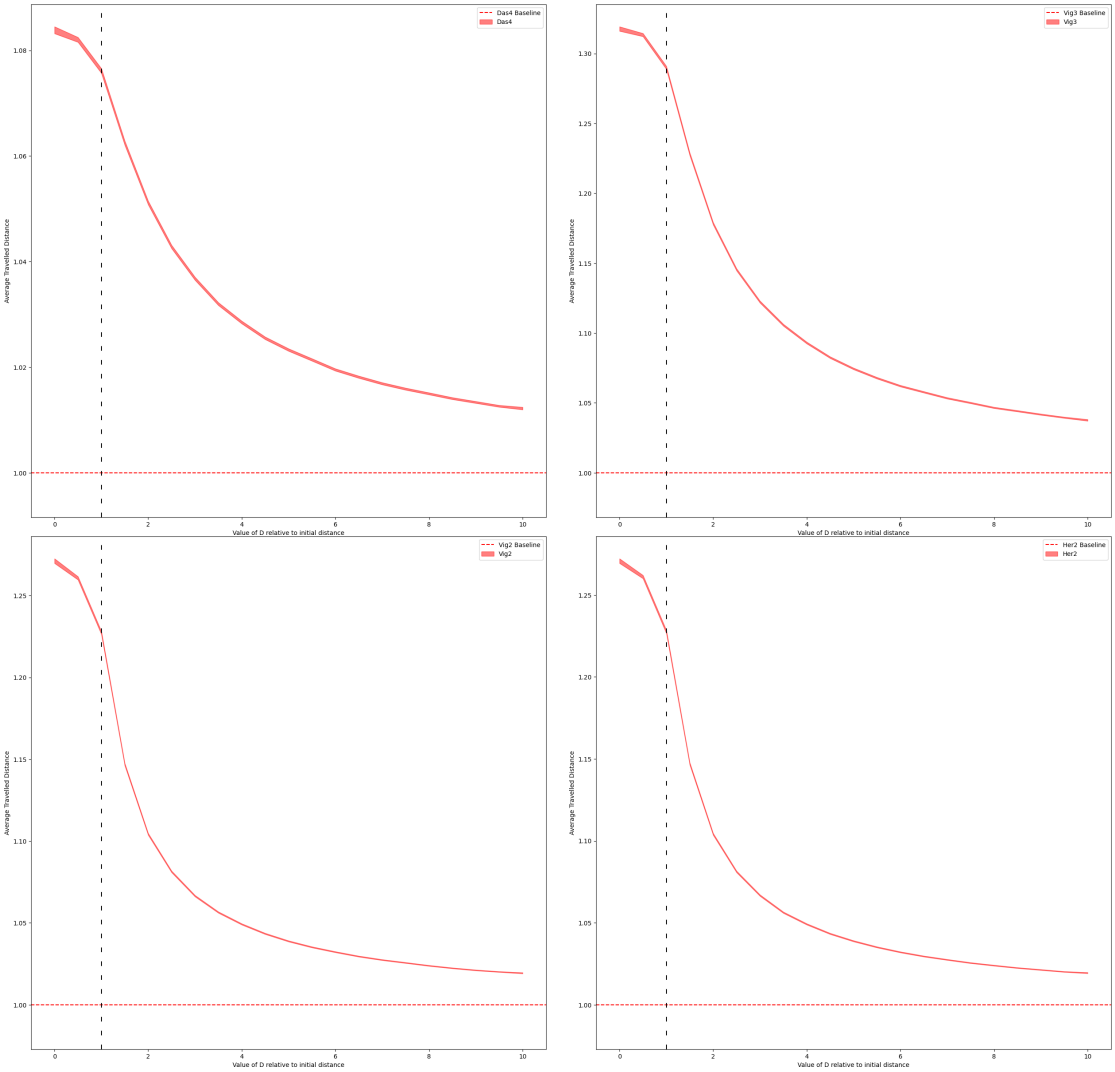


Figure C.7 – Distance for rigid SSYNC

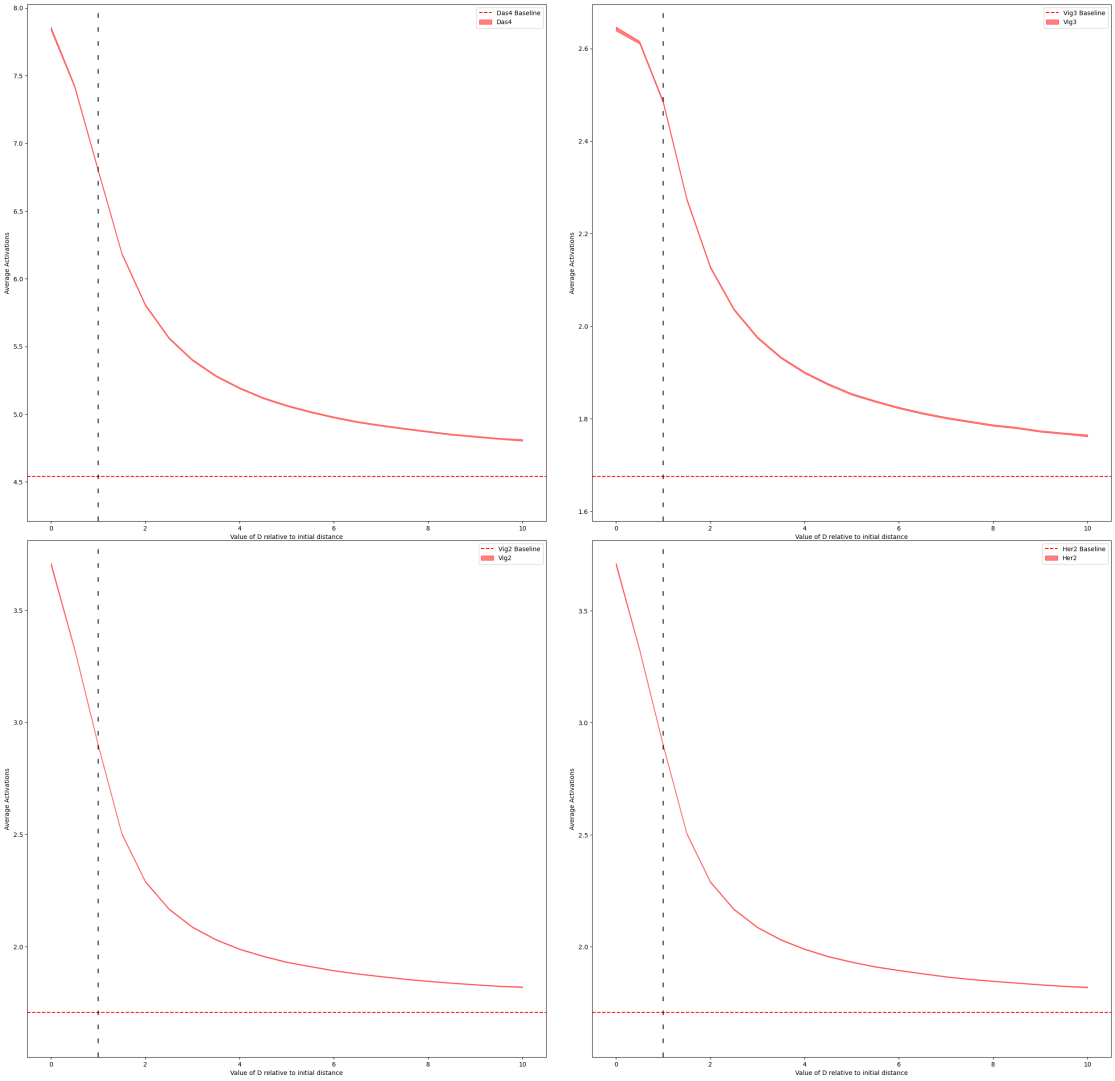


Figure C.8 – Activations for rigid SSYNC

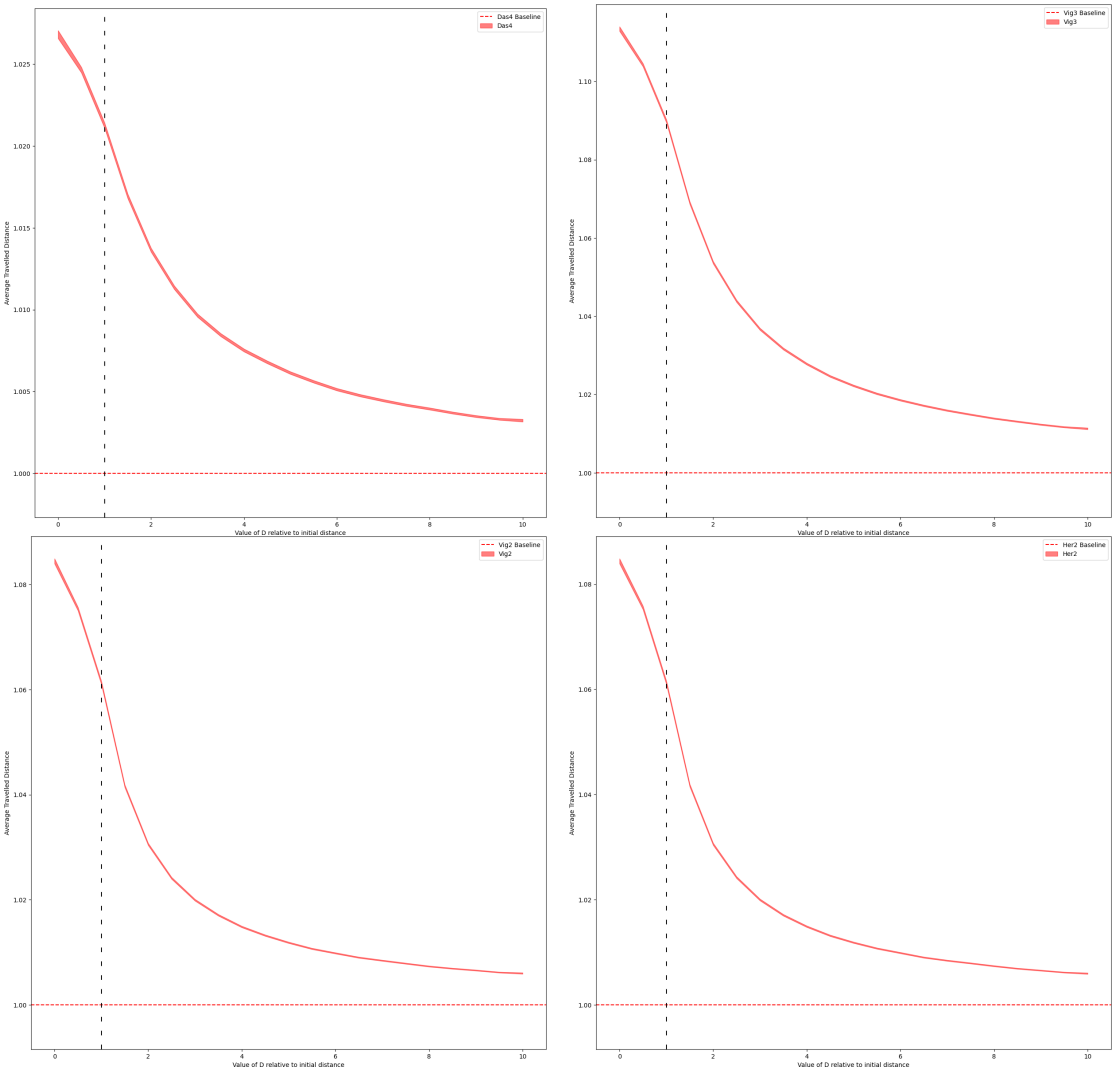


Figure C.9 – Distance for non-rigid FSYNC

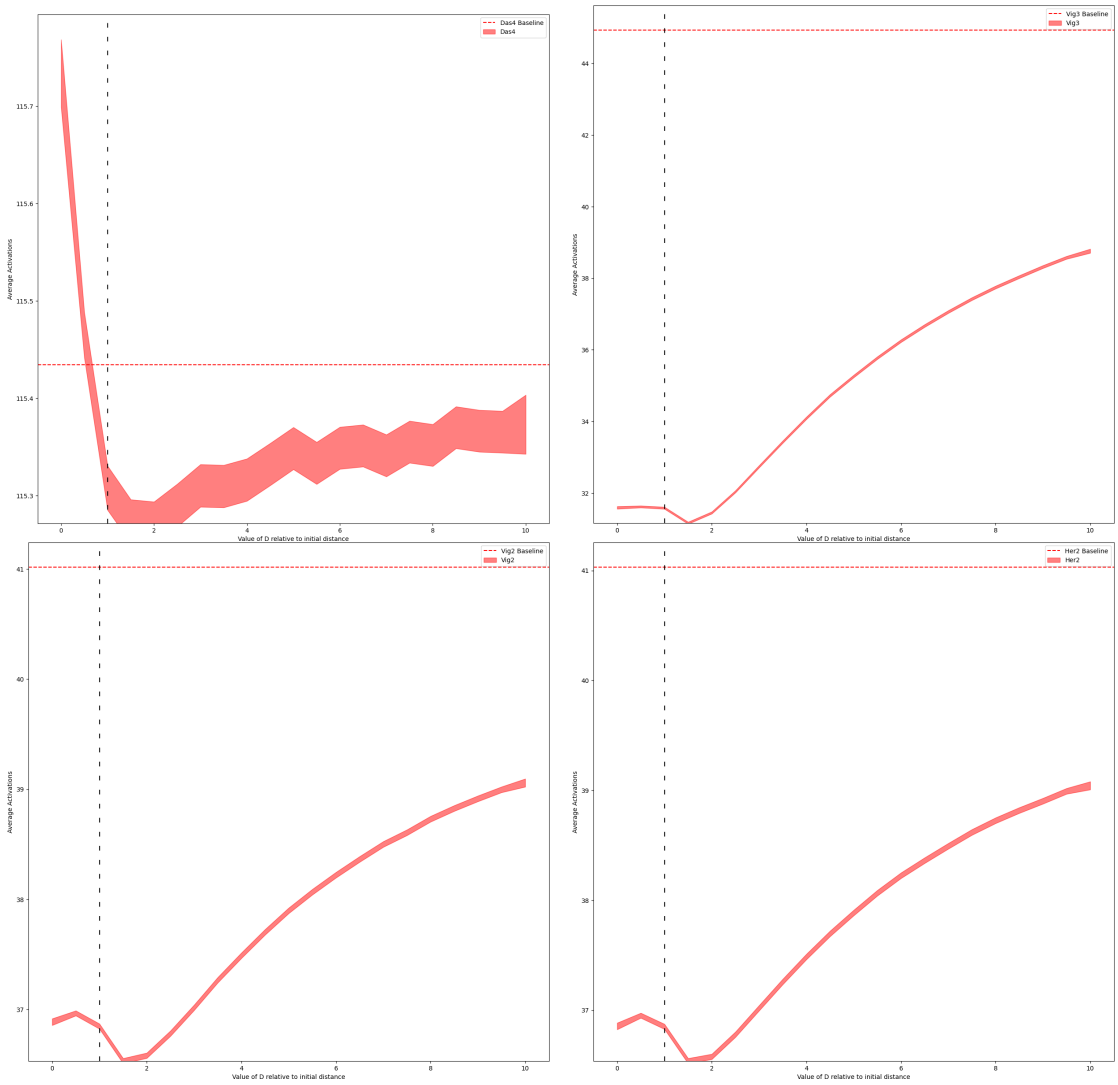


Figure C.10 – Activations for non-rigid FSYNC

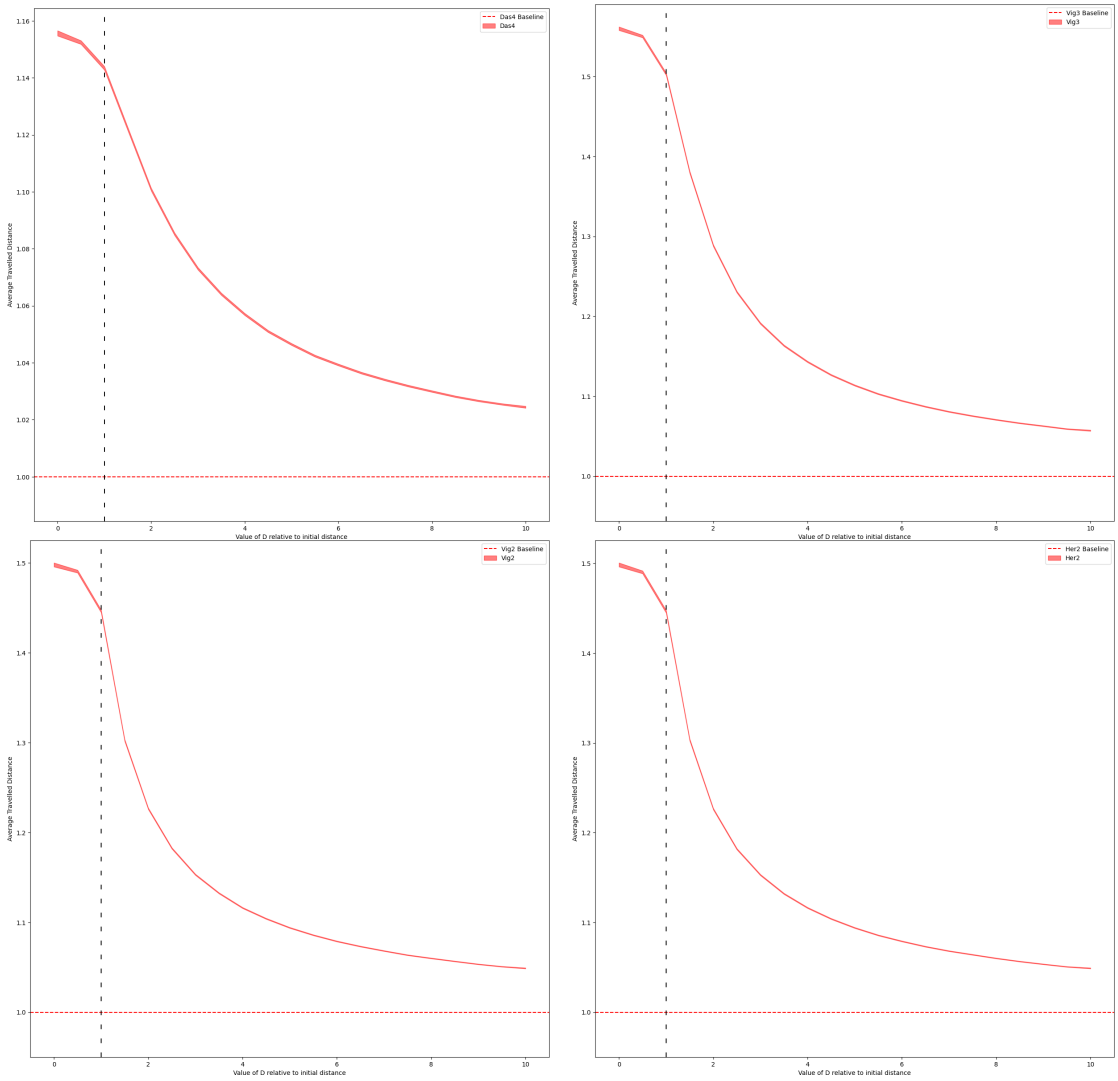


Figure C.11 – Distance for rigid FSYNC

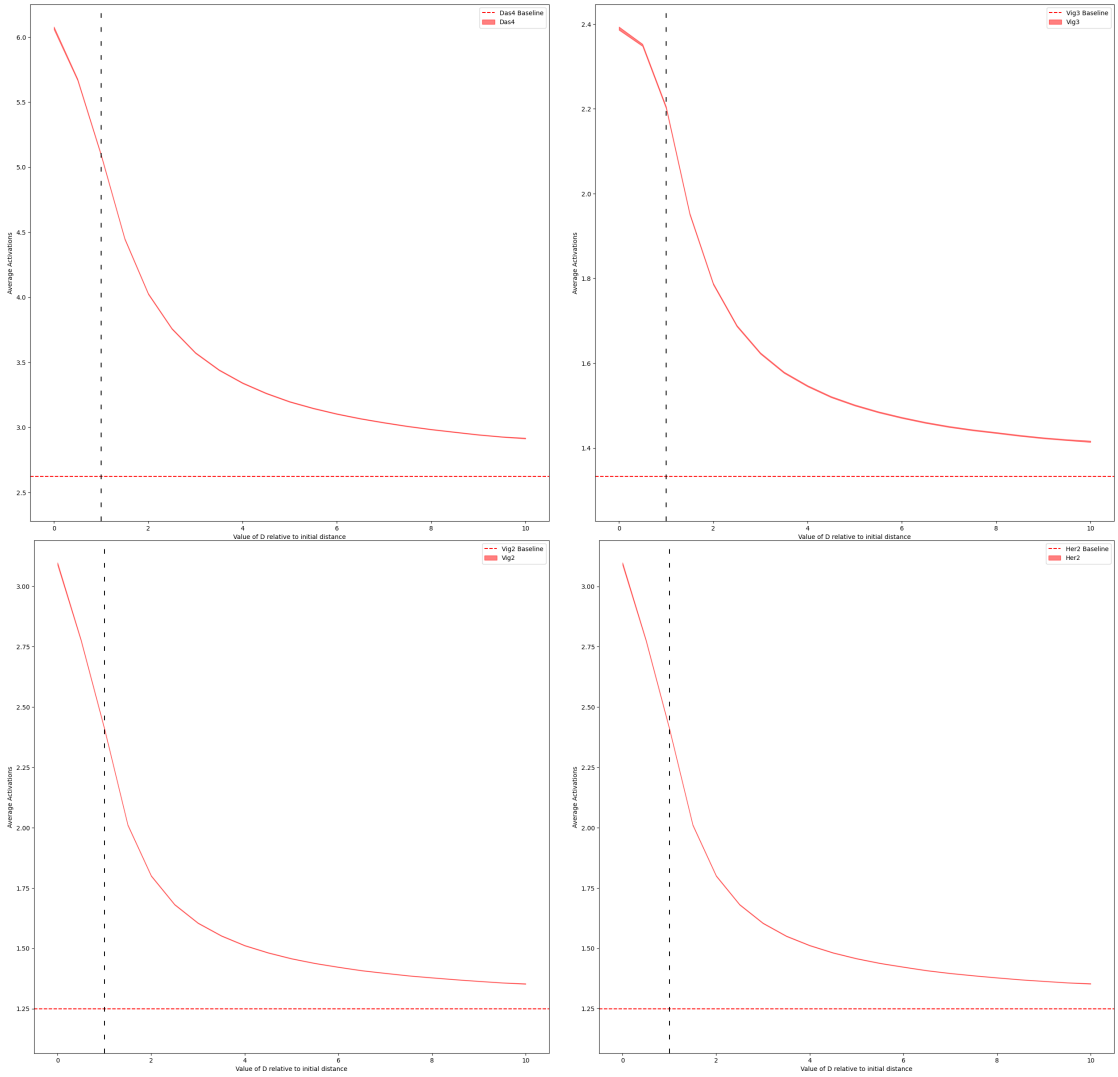


Figure C.12 – Activations for rigid FSYNC

List of Acronyms

ASYNC Asynchronous Scheduler.

FSYNC Fully Synchronous Scheduler.

LCM LOOK-COMPUTE-MOVE.

M2H Move To Half.

M2O Move To Other.

OBLOT Oblivious Robot Model.

SEC Smallest Enclosing Circle.

SSYNC Semi Synchronous Scheduler.

Bibliography

[1] Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. "Evaluating and Optimizing Stabilizing Dining Philosophers". In: *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*. IEEE Computer Society, 2015, pp. 233–244. DOI: 10.1109/EDCC.2015.11.

[2] Noa Agmon and David Peleg. "Fault-Tolerant Gathering Algorithms for Autonomous Mobile Robots". In: *SIAM J. Comput.* 36.1 (2006), pp. 56–82. DOI: 10.1137/050645221.

[3] Hideki Ando, Yoshinobu Oasa, Ichiro Suzuki, and Masafumi Yamashita. "Distributed memoryless point convergence algorithm for mobile robots with limited visibility". In: *IEEE Trans. Robotics and Automation* 15.5 (1999), pp. 818–828. DOI: 10.1109/70.795787.

[4] Divansh Arora, Parikshit Maini, Pedro Pinacho Davidson, and Christian Blum. "Route planning for cooperative air-ground robots with fuel constraints: an approach based on CMSA". In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, ed. by Anne Auger et al. ACM, 2019, pp. 207–214. DOI: 10.1145/3321707.3321820.

[5] Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)* Wiley series on parallel and distributed computing. Wiley, 2004. ISBN: 978-0-471-45324-6.

[6] Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. "Certified Impossibility Results for Byzantine-Tolerant Mobile Robots". In: *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, ed. by Teruo Higashino et al. Vol. 8255. Lecture Notes in Computer Science. Springer, 2013, pp. 178–190. DOI: 10.1007/978-3-319-03089-0_13.

[7] Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Brief Announcement Continuous vs. Discrete Asynchronous Moves: A Certified Approach for Mobile Robots". In: *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, ed. by Taisuke Izumi et al. Vol. 11201. Lecture Notes in Computer Science. Springer, 2018, pp. 404–408. DOI: 10.1007/978-3-030-03232-6_29.

[8] Thibaut Balabonski, Amélie Delga, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Synchronous Gathering without Multiplicity Detection: a Certified Algorithm". In: *Theory Comput. Syst.* 63.2 (2019), pp. 200–218. DOI: 10.1007/s00224-017-9828-z.

[9] Thibaut Balabonski, Robin Pelle, Lionel Rieg, and Sébastien Tixeuil. "A Foundational Framework for Certified Impossibility Results with Mobile Robots on Graphs". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, ed. by Paolo Bellavista et al. ACM, 2018, 5:1–5:10. DOI: 10.1145/3154273.3154321.

[10] Evangelos Bampas, Lélia Blin, Jurek Czyzowicz, David Ilcinkas, Arnaud Labourel, Maria Potop-Butucaru, and Sébastien Tixeuil. "On asynchronous rendezvous in general graphs". In: *Theor. Comput. Sci.* 753 (2019), pp. 80–90. DOI: 10.1016/j.tcs.2018.06.045.

[11] Béatrice Bérard, Pierre Courtieu, Laure Millet, Maria Potop-Butucaru, Lionel Rieg, Nathalie Sznajder, Sébastien Tixeuil, and Xavier Urbain. "[Invited Paper] Formal Methods for Mobile Robots: Current Results and Open Problems". In: *International Journal of Informatics Society* 7.3 (2015), pp. 101–114.

[12] Béatrice Bérard, Pascal Lafourcade, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. "Formal verification of mobile robot protocols". In: *Distributed Comput.* 29.6 (2016), pp. 459–487. DOI: 10.1007/s00446-016-0271-1.

[13] Subhash Bhagat, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. "Formation of General Position by Asynchronous Mobile Robots Under One-Axis Agreement". In: *WALCOM: Algorithms and Computation - 10th International Workshop, WALCOM 2016, Kathmandu, Nepal, March 29-31, 2016, Proceedings*, ed. by Mohammad Kaykobad et al. Vol. 9627. Lecture Notes in Computer Science. Springer, 2016, pp. 80–91. ISBN: 978-3-319-30138-9. DOI: 10.1007/978-3-319-30139-6_7.

[14] Subhash Bhagat and Krishnendu Mukhopadhyaya. "Optimum Algorithm for Mutual Visibility Among Asynchronous Robots with Lights". In: *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, ed. by Paul G. Spirakis et al. Vol. 10616. Lecture Notes in Computer Science. Springer, 2017, pp. 341–355. DOI: 10.1007/978-3-319-69084-1_24.

[15] François Bonnet, Xavier Défago, Franck Petit, Maria Potop-Butucaru, and Sébastien Tixeuil. "Discovering and Assessing Fine-Grained Metrics in Robot Networks Protocols". In: *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*. IEEE Computer Society, 2014, pp. 50–59. DOI: 10.1109/SRDSW.2014.34.

[16] Zohir Bouzid, Shantanu Das, and Sébastien Tixeuil. "Gathering of Mobile Robots Tolerating Multiple Crash Faults". In: *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 2013, pp. 337–346. DOI: 10.1109/ICDCS.2013.27.

[17] Quentin Bramas and Sébastien Tixeuil. "Wait-Free Gathering Without Chirality". In: *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, ed. by Christian Scheideler. Vol. 9439. Lecture Notes in Computer Science. Springer, 2015, pp. 313–327. DOI: 10.1007/978-3-319-25258-2_22.

[18] Davide Canepa, Xavier Défago, Taisuke Izumi, and Maria Potop-Butucaru. "Flocking with Oblivious Robots". In: *Stabilization, Safety, and Security of Distributed Systems - 18th International Symposium, SSS 2016, Lyon, France, November 7-10, 2016, Proceedings*, ed. by Borzoo Bonakdarpour et al. Vol. 10083. Lecture Notes in Computer Science. 2016, pp. 94–108. ISBN: 978-3-319-49258-2. DOI: 10.1007/978-3-319-49259-9_8.

[19] Davide Canepa and Maria Gradinariu Potop-Butucaru. "Stabilizing Flocking Via Leader Election in Robot Networks". In: *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings*, ed. by Toshimitsu Masuzawa et al. Vol. 4838. Lecture Notes in Computer Science. Springer, 2007, pp. 52–66. ISBN: 978-3-540-76626-1. DOI: 10.1007/978-3-540-76627-8_7.

[20] Jérémie Chalopin, Shantanu Das, Matús Mihalák, Paolo Penna, and Peter Widmayer. "Data Delivery by Energy-Constrained Mobile Agents". In: *Algorithms for Sensor Systems - 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS 2013, Sophia Antipolis, France, September 5-6, 2013, Revised Selected Papers*, ed. by Paola Flocchini et al. Vol. 8243. Lecture Notes in Computer Science. Springer, 2013, pp. 111–122. ISBN: 978-3-642-45345-8. DOI: 10.1007/978-3-642-45346-5_9.

[21] Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. "Minimum-Traveled-Distance Gathering of Oblivious Robots over Given Meeting Points". In: *Algorithms for Sensor Systems - 10th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS 2014, Wroclaw, Poland, September 12, 2014, Revised Selected Papers*, ed. by Jie Gao et al. Vol. 8847. Lecture Notes in Computer Science. Springer, 2014, pp. 57–72. DOI: 10.1007/978-3-662-46018-4_4.

[22] Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. "Asynchronous Arbitrary Pattern Formation: the effects of a rigorous approach". In: *Distributed Comput.* 32.2 (2019), pp. 91–132. DOI: 10.1007/s00446-018-0325-7.

[23] Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. "Asynchronous Robots on Graphs: Gathering". In: *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, ed. by Paola Flocchini et al. Vol. 11340. Lecture Notes in Computer Science. Springer, 2019, pp. 184–217. DOI: 10.1007/978-3-030-11072-7_8.

[24] Reuven Cohen and David Peleg. "Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems". In: *SIAM J. Comput.* 34.6 (2005), pp. 1516–1528. DOI: 10.1137/S0097539704446475.

[25] Reuven Cohen and David Peleg. "Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements". In: *SIAM J. Comput.* 38.1 (2008), pp. 276–302. DOI: 10.1137/060665257.

[26] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Impossibility of gathering, a certification". In: *Inf. Process. Lett.* 115.3 (2015), pp. 447–452. DOI: 10.1016/j.ipl.2014.11.001.

[27] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Certified Universal Gathering in $\mathbb{R}^2$ for Oblivious Mobile Robots". In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, ed. by Cyril Gavoille et al. Vol. 9888. Lecture Notes in Computer Science. Springer, 2016, pp. 187–200. DOI: 10.1007/978-3-662-53426-7_14.

[28] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. "Swarms of Mobile Robots: towards Versatility with Safety". Under review. 2020.

[29] Jurek Czyzowicz, Leszek Gasieniec, and Andrzej Pelc. "Gathering few fat mobile robots in the plane". In: *Theor. Comput. Sci.* 410.6-7 (2009), pp. 481–499. DOI: 10.1016/j.tcs.2008.10.005.

[30] Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. "The Power of Lights: Synchronizing Asynchronous Robots Using Visible Bits". In: *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012*. IEEE Computer Society, 2012, pp. 506–515. ISBN: 978-1-4577-0295-2. DOI: 10.1109/ICDCS.2012.71.

[31] Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. "Autonomous mobile robots with lights". In: *Theor. Comput. Sci.* 609 (2016), pp. 171–184. DOI: 10.1016/j.tcs.2015.09.018.

[32] Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. "Fault-Tolerant and Self-stabilizing Mobile Robots Gathering". In: *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, ed. by Shlomi Dolev. Vol. 4167. Lecture Notes in Computer Science. Springer, 2006, pp. 46–60. DOI: 10.1007/11864219_4.

[33] Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada. "Brief Announcement: Model Checking Rendezvous Algorithms for Robots with Lights in Euclidean Space". In: *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, ed. by Jukka Suomela. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 41:1–41:3. DOI: 10.4230/LIPIcs.DISC.2019.41.

[34] Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada. "Using Model Checking to Formally Verify Rendezvous Algorithms for Robots with Lights in Euclidean Space". In: *39th Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*. IEEE, 2020.

[35] Xavier Défago, Maria Potop-Butucaru, and Philippe Raipin Parvédy. "Self-stabilizing gathering of mobile robots under crash or Byzantine faults". In: *Distributed Comput.* 33.5 (2020), pp. 393–421. DOI: 10.1007/s00446-019-00359-x.

[36] Stéphane Devismes, Anissa Lamani, Franck Petit, Pascal Raymond, and Sébastien Tixeuil. "Optimal Grid Exploration by Asynchronous Oblivious Robots". In: *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, ed. by Andréa W. Richa et al. Vol. 7596. Lecture Notes in Computer Science. Springer, 2012, pp. 64–76. DOI: 10.1007/978-3-642-33536-5_7.

[37] Yoann Dieudonné, Florence Levé, Franck Petit, and Vincent Villain. "Deterministic geoleader election in disoriented anonymous systems". In: *Theor. Comput. Sci.* 506 (2013), pp. 43–54. DOI: 10.1016/j.tcs.2013.07.033.

[38] Yoann Dieudonné and Franck Petit. "Self-stabilizing gathering with strong multiplicity detection". In: *Theor. Comput. Sci.* 428 (2012), pp. 47–57. DOI: 10.1016/j.tcs.2011.12.010.

[39] Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. "Model Checking of a Mobile Robots Perpetual Exploration Algorithm". In: *Structured Object-Oriented Formal Language and Method - 6th International Workshop, SOFL+MSVL 2016, Tokyo, Japan, November 15, 2016, Revised Selected Papers*, ed. by Shaoying Liu et al. Vol. 10189. Lecture Notes in Computer Science. 2016, pp. 201–219. DOI: 10.1007/978-3-319-57708-1_12.

[40] Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. "Model Checking of Robot Gathering". In: *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, ed. by James Aspnes et al. Vol. 95. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 12:1–12:16. DOI: 10.4230/LIPIcs.OPODIS.2017.12.

[41] Miroslaw Dynia, Miroslaw Korzeniowski, and Christian Schindelhauer. "Power-Aware Collective Tree Exploration". In: *Architecture of Computing Systems - ARCS 2006, 19th International Conference, Frankfurt/Main, Germany, March 13-16, 2006, Proceedings*, ed. by Werner Grass et al. Vol. 3894. Lecture Notes in Computer Science. Springer, 2006, pp. 341–351. ISBN: 3-540-32765-7. DOI: 10.1007/11682127_24.

[42] Paola Flocchini. “Gathering”. In: *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, ed. by Paola Flocchini et al. Vol. 11340. Lecture Notes in Computer Science. Springer, 2019, pp. 63–82. DOI: 10.1007/978-3-030-11072-7_4.

[43] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, eds. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*. Vol. 11340. Lecture Notes in Computer Science. Springer, 2019. ISBN: 978-3-030-11071-0. DOI: 10.1007/978-3-030-11072-7.

[44] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. “Distributed computing by mobile robots: uniform circle formation”. In: *Distributed Computing* 30.6 (2017), pp. 413–457. DOI: 10.1007/s00446-016-0291-x.

[45] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. “Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots”. In: *Algorithms and Computation, 10th International Symposium, ISAAC '99, Chennai, India, December 16-18, 1999, Proceedings*, ed. by Alok Aggarwal et al. Vol. 1741. Lecture Notes in Computer Science. Springer, 1999, pp. 93–102. DOI: 10.1007/3-540-46632-0_10.

[46] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. “Gathering of asynchronous robots with limited visibility”. In: *Theor. Comput. Sci.* 337.1-3 (2005), pp. 147–168. DOI: 10.1016/j.tcs.2005.01.001.

[47] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. “Arbitrary pattern formation by asynchronous, anonymous, oblivious robots”. In: *Theor. Comput. Sci.* 407.1-3 (2008), pp. 412–447. DOI: 10.1016/j.tcs.2008.07.026.

[48] Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. “Rendezvous with constant memory”. In: *Theor. Comput. Sci.* 621 (2016), pp. 57–72. DOI: 10.1016/j.tcs.2016.01.025.

[49] Vincenzo Gervasi and Giuseppe Prencipe. “Coordination without communication: the case of the flocking problem”. In: *Discrete Applied Mathematics* 144.3 (2004), pp. 324–344. DOI: 10.1016/j.dam.2003.11.010.

[50] Noam Gordon, Yotam Elor, and Alfred M. Bruckstein. “Gathering Multiple Robotic Agents with Crude Distance Sensing Capabilities”. In: *Ant Colony Optimization and Swarm Intelligence, 6th International Conference, ANTS 2008, Brussels, Belgium, September 22-24, 2008. Proceedings*, ed. by Marco Dorigo et al. Vol. 5217. Lecture Notes in Computer Science. Springer, 2008, pp. 72–83. ISBN: 978-3-540-87526-0. DOI: 10.1007/978-3-540-87527-7_7.

[51] Noam Gordon, Israel A. Wagner, and Alfred M. Bruckstein. “Gathering Multiple Robotic A(ge)nts with Limited Sensing Capabilities”. In: *Ant Colony Optimization and Swarm Intelligence, 4th International Workshop, ANTS 2004, Brussels, Belgium, September 5 - 8, 2004, Proceedings*, ed. by Marco Dorigo et al. Vol. 3172. Lecture Notes in Computer Science. Springer, 2004, pp. 142–153. ISBN: 3-540-22672-9. DOI: 10.1007/978-3-540-28646-2_13.

[52] Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman. “A Probabilistically Correct Leader Election Protocol for Large Groups”. In: *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, ed. by Maurice Herlihy. Vol. 1914. Lecture Notes in Computer Science. Springer, 2000, pp. 89–103. ISBN: 3-540-41143-7. DOI: 10.1007/3-540-40026-5_6.

[53] Adam Heriban, Xavier Défago, and Sébastien Tixeuil. "Optimally Gathering Two Robots". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, ed. by Paolo Bellavista et al. ACM, 2018, 3:1–3:10. DOI: 10.1145/3154273.3154323.

[54] Adam Heriban, Michiko Inoue, Fukuhito Ooshita, and Sébastien Tixeuil. "Obstruction detection by asynchronous opaque robots using lights". In: *IEICE Technical Report, Theoretical Foundations of Computing, Nagoya Institute of Technology*. Vol. 118-68. Lecture Notes in Computer Science. 2018, pp. 71–78. URL: https://www.ieice.org/ken/paper/20180526t1EQ/eng/.

[55] Adam Heriban and Sébastien Tixeuil. "Mobile Robots with Uncertain Visibility Sensors". In: *Structural Information and Communication Complexity - 26th International Colloquium, SIROCCO 2019, L'Aquila, Italy, July 1-4, 2019, Proceedings*, ed. by Keren Censor-Hillel et al. Vol. 11639. Lecture Notes in Computer Science. Springer, 2019, pp. 349–352. ISBN: 978-3-030-24921-2. DOI: 10.1007/978-3-030-24922-9_27.

[56] Adam Heriban and Sébastien Tixeuil. "Mobile Robots with Uncertain Visibility Sensors". In: *Parallel Process. Lett.* (2020).

[57] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[58] Anthony Honorat, Maria Potop-Butucaru, and Sébastien Tixeuil. "Gathering fat mobile robots with slim omnidirectional cameras". In: *Theor. Comput. Sci.* 557 (2014), pp. 1–27. DOI: 10.1016/j.tcs.2014.08.004.

[59] David Ilcinkas. "Oblivious Robots on Graphs: Exploration". In: *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, ed. by Paola Flocchini et al. Vol. 11340. Lecture Notes in Computer Science. Springer, 2019, pp. 218–233. DOI: 10.1007/978-3-030-11072-7_9.

[60] Taisuke Izumi, Tomoko Izumi, Sayaka Kamei, and Fukuhito Ooshita. "Randomized Gathering of Mobile Robots with Local-Multiplicity Detection". In: *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, ed. by Rachid Guerraoui et al. Vol. 5873. Lecture Notes in Computer Science. Springer, 2009, pp. 384–398. DOI: 10.1007/978-3-642-05118-0_27.

[61] Taisuke Izumi, Samia Souissi, Yoshiaki Katayama, Nobuhiro Inuzuka, Xavier Défago, Koichi Wada, and Masafumi Yamashita. "The Gathering Problem for Two Oblivious Robots with Unreliable Compasses". In: *SIAM J. Comput.* 41.1 (2012), pp. 26–46. DOI: 10.1137/100797916.

[62] Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada. "Gathering on Rings for Myopic Asynchronous Robots With Lights". In: *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, ed. by Pascal Felber et al. Vol. 153. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 27:1–27:17. DOI: 10.4230/LIPIcs.OPODIS.2019.27.

[63] Johannes Köhler, Alain Pagani, and Didier Stricker. "Detection and Identification Techniques for Markers Used in Computer Vision". In: *Visualization of Large and Unstructured Data Sets - Applications in Geospatial Planning, Modeling and Engineering (IRTG 1131 Workshop), VLUDS 2010, March 19-21, 2010, Bodega Bay, CA, USA*, ed. by Ariane Middel et al. Vol. 19. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 36–44. ISBN: 978-3-939897-29-3. DOI: 10.4230/OASIcs.VLUDS.2010.36.

[64] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[65] Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Federico Poloni, Nicola Santoro, and Giovanni Viglietta. "Mutual visibility by luminous robots without collisions". In: *Inf. Comput.* 254 (2017), pp. 392–418. DOI: 10.1016/j.ic.2016.09.005.

[66] Giuseppe Antonio Di Luna, Paola Flocchini, Federico Poloni, Nicola Santoro, and Giovanni Viglietta. "The Mutual Visibility Problem for Oblivious Robots". In: *Proceedings of the 26th Canadian Conference on Computational Geometry, CCCG 2014, Halifax, Nova Scotia, Canada, 2014*. Carleton University, Ottawa, Canada, 2014. URL: http://www.cccg.ca/proceedings/2014/papers/paper51.pdf.

[67] Marcello Mamino and Giovanni Viglietta. "Square Formation by Asynchronous Oblivious Robots". In: *Proceedings of the 28th Canadian Conference on Computational Geometry, CCCG 2016, August 3-5, 2016, Simon Fraser University, Vancouver, British Columbia, Canada*, ed. by Thomas C. Shermer. Simon Fraser University, Vancouver, British Columbia, Canada, 2016, pp. 1–6. URL: http://www.cccg.ca/proceedings/2016/proceedings2016.pdf.

[68] Sonia Martínez. "Practical multiagent rendezvous through modified circumcenter algorithms". In: *Automatica* 45.9 (2009), pp. 2010–2017. DOI: 10.1016/j.automatica.2009.05.013.

[69] Nicholas Metropolis and S. Ulam. "The Monte Carlo Method". In: *Journal of the American Statistical Association* 44.247 (1949), pp. 335–341.

[70] Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. "On the Synthesis of Mobile Robots Algorithms: The Case of Ring Gathering". In: *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, ed. by Pascal Felber et al. Vol. 8756. Lecture Notes in Computer Science. Springer, 2014, pp. 237–251. DOI: 10.1007/978-3-319-11764-5_17.

[71] El Mustapha Mouaddib, Ryusuke Sagawa, Tomio Echigo, and Yasushi Yagi. "Stereovision with a Single Camera and Multiple Mirrors". In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation, ICRA 2005, April 18-22, 2005, Barcelona, Spain*. IEEE, 2005, pp. 800–805. DOI: 10.1109/ROBOT.2005.1570215.

[72] Shota Nagahama, Fukuhito Ooshita, and Michiko Inoue. "Ring Exploration of Myopic Luminous Robots with Visibility More Than One". In: *Stabilization, Safety, and Security of Distributed Systems - 21st International Symposium, SSS 2019, Pisa, Italy, October 22-25, 2019, Proceedings*, ed. by Mohsen Ghaffari et al. Vol. 11914. Lecture Notes in Computer Science. Springer, 2019, pp. 256–271. DOI: 10.1007/978-3-030-34992-9_20.

[73] Takashi Okumura, Koichi Wada, and Xavier Défago. "Optimal Rendezvous $\mathcal{L}$-Algorithms for Asynchronous Mobile Robots with External-Lights". In: *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, ed. by Jiannong Cao et al. Vol. 125. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 24:1–24:16. ISBN: 978-3-95977-098-9. DOI: 10.4230/LIPIcs.OPODIS.2018.24.

[74] Takashi Okumura, Koichi Wada, and Yoshiaki Katayama. "Brief Announcement: Optimal Asynchronous Rendezvous for Mobile Robots with Lights". In: *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, ed. by Paul G. Spirakis et al. Vol. 10616. Lecture Notes in Computer Science. Springer, 2017, pp. 484–488. DOI: 10.1007/978-3-319-69084-1_36.

[75] Takashi Okumura, Koichi Wada, and Yoshiaki Katayama. "Rendezvous of Asynchronous Mobile Robots with Lights". In: *Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*. 2018, pp. 434–448. DOI: 10.1007/978-3-319-98355-4_25.

[76] David Peleg. "Distributed Coordination Algorithms for Mobile Robot Swarms: New Directions and Challenges". In: *Distributed Computing - IWDC 2005, 7th International Workshop, Kharagpur, India, December 27-30, 2005, Proceedings*, ed. by Ajit Pal et al. Vol. 3741. Lecture Notes in Computer Science. Springer, 2005, pp. 1–12. DOI: 10.1007/11603771_1.

[77] Giuseppe Prencipe. "Impossibility of gathering by a set of autonomous mobile robots". In: *Theor. Comput. Sci.* 384.2-3 (2007), pp. 222–231. DOI: 10.1016/j.tcs.2007.04.023.

[78] Sasha Rubin, Florian Zuleger, Aniello Murano, and Benjamin Aminof. "Verification of Asynchronous Mobile-Robots in Partially-Known Environments". In: *PRIMA 2015: Principles and Practice of Multi-Agent Systems - 18th International Conference, Bertinoro, Italy, October 26-30, 2015, Proceedings*, ed. by Qingliang Chen et al. Vol. 9387. Lecture Notes in Computer Science. Springer, 2015, pp. 185–200. DOI: 10.1007/978-3-319-25524-8_12.

[79] Arnaud Sangnier, Nathalie Sznajder, Maria Potop-Butucaru, and Sébastien Tixeuil. "Parameterized verification of algorithms for oblivious robots on a ring". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, ed. by Daryl Stewart et al. IEEE, 2017, pp. 212–219. DOI: 10.23919/FMCAD.2017.8102262.

[80] Nicola Santoro and Peter Widmayer. "Time is Not a Healer". In: *STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Proceedings*, ed. by Burkhard Monien et al. Vol. 349. Lecture Notes in Computer Science. Springer, 1989, pp. 304–313. ISBN: 3-540-50840-6. DOI: 10.1007/BFb0028994.

[81] Gokarna Sharma, Costas Busch, and Supratik Mukhopadhyay. "Mutual Visibility with an Optimal Number of Colors". In: *Algorithms for Sensor Systems - 11th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2015, Patras, Greece, September 17-18, 2015, Revised Selected Papers*, ed. by Prosenjit Bose et al. Vol. 9536. Lecture Notes in Computer Science. Springer, 2015, pp. 196–210. ISBN: 978-3-319-28471-2. DOI: 10.1007/978-3-319-28472-9_15.

[82] Samia Souissi, Xavier Défago, and Masafumi Yamashita. "Using eventually consistent compasses to gather memory-less mobile robots with limited visibility". In: *ACM Trans. Auton. Adapt. Syst.* 4.1 (2009), 9:1–9:27. DOI: 10.1145/1462187.1462196.

[83] Ichiro Suzuki and Masafumi Yamashita. "Distributed Anonymous Mobile Robots: Formation of Geometric Patterns". In: *SIAM J. Comput.* 28.4 (1999), pp. 1347–1363. DOI: 10.1137/S009753979628292X.

[84] Giovanni Viglietta. “Rendezvous of Two Robots with Visible Bits”. In: *Algorithms for Sensor Systems - 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS 2013, Sophia Antipolis, France, September 5-6, 2013, Revised Selected Papers*, ed. by Paola Flocchini et al. Vol. 8243. Lecture Notes in Computer Science. Springer, 2013, pp. 291–306. DOI: 10.1007/978-3-642-45346-5_21.

[85] Kenta Yamamoto, Taisuke Izumi, Yoshiaki Katayama, Nobuhiro Inuzuka, and Koichi Wada. “The optimal tolerance of uniform observation error for mobile robot convergence”. In: *Theor. Comput. Sci.* 444 (2012), pp. 77–86. DOI: 10.1016/j.tcs.2012.04.038.

[86] Rami Yared, Xavier Défago, Julien Iguchi-Cartigny, and Matthias Wiesmann. “Collision Prevention Platform for a Dynamic Group of Asynchronous Cooperative Mobile Robots”. In: *J. Networks* 2.4 (2007), pp. 28–39. DOI: 10.4304/jnw.2.4.28-39.

[87] Chanyeol Yoo, Robert Fitch, and Salah Sukkarieh. “Online task planning and control for fuel-constrained aerial robots in wind fields”. In: *Int. J. Robotics Res.* 35.5 (2016), pp. 438–453. DOI: 10.1177/0278364915595278.

Adam Heriban **09/12/2020**

Sujet : Réseaux de Robots Réalistes

Résumé : Le but de cette thèse est d'analyser le travail existant par la communauté de robotique distribuée pour trouver des variations réalistes du modèle standard OBLOT, et développer de nouvelles variations viables à long terme.
Nous développons un nouvel algorithme optimal pour le rendezvous avec des lumières, et le prouvons en utilisant le framework de model-checking SPIN. En utilisant ce modèle, nous construisons des algorithmes d'élection de leader robustes, permettant des contraintes plus strictes.
Nous définissons un nouveau modèle de vision pour les robots mobile : Uncertain Visibility, qui utilise un adversaire pour représenter des faux-négatifs des capteurs, et prouvons les bornes de plusieurs problèmes étalons dans ce modèle. Nous définissons et analysons un nouveau problème : Obstruction Detection pour le modèle des robots opaques.
Nous développons un simulateur Monte-Carlo pour les robots mobiles, conçus pour facilement simuler n'importe quel modèle ou algorithme. N'étant pas un model-checker, il vise d'abord a remplacer "l'intuition" des chercheurs pour détecter des comportement imprévus. Nous testons plusieurs algorithmes et modèles, avec des résultats encourageants.
Enfin, nous présentons deux nouveaux algorithmes : le premier assure que la distance parcourue pour la convergence en ASYNC est minimale ; le second permet d'élire un Leader avec des capteurs imprécis.

Mots clés : Robots Mobiles, Vision Imparfaite, Robots Lumineux, Adversaire Asynchrone, Analyse de Performance, Simulation Monte-Carlo

Subject : Networks of Realistic Robots

Abstract: The goal of this thesis is to survey and analyze the current work done by the distributed robotics community to find the more realistic variations of the standard OBLOT model, develop new such variations, and determine which approach should be used in the long term.
We develop a new, optimal Rendezvous algorithm using lights, and prove it using a model checking framework based on the SPIN model checker. The same luminous model is used to build robust Leader Election algorithms, which allow for stricter constraints.
We design a new vision model for mobile robots, Uncertain Visibility, which introduces a vision adversary to model false negatives in sensors, and prove tight bounds under this new model for several benchmark problems. We then define and investigate a new problem, Obstruction Detection, for the obstructed visibility model.
To facilitate analysis of robot networks, we develop a framework for Monte-Carlo simulations of mobile robots, designed to simulate any model or algorithm with minimal effort. It is designed as a complement to researcher "intuition" to look for unexpected behavior. We test this simulator against numerous algorithms and settings, yielding encouraging results. Finally, we introduce another two algorithms: the first ensures the distance traveled for convergence in ASYNC is minimal ; the second allows for Leader Election with errors in vision.

Keywords : Mobile Robots, Restricted Visibility, Luminous Robots, Asynchronous Scheduler, Performance Analysis, Monte-Carlo Simulation