# Model Checking of Robot Gathering

**Ha Thi Thu Doan[1], François Bonnet[2], and Kazuhiro Ogata[3]**

1 **Japan Advanced Institute of Science and Technology, Nomi, Japan**
doanha@jaist.ac.jp
2 **Graduate School of Engineering, Osaka University, Osaka, Japan**
francois@cy2sec.comm.eng.osaka-u.ac.jp
3 **Japan Advanced Institute of Science and Technology, Nomi, Japan**
ogata@jaist.ac.jp

## Abstract

Recent advances in distributed computing highlight models and algorithms for autonomous mobile robots that self-organize and cooperate together in order to solve a global objective. As results, a large number of algorithms have been proposed. These algorithms are given together with proofs to assess their correctness. However, those proofs are informal, which are error prone. This paper presents our study on formal verification of mobile robot algorithms. We first propose a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions. We specify this formal model in Maude, a specification and programming language based on rewriting logic. We then use its model checker to formally verify an algorithm for robot gathering problem on ring enjoys some desired properties. As the result of the model checking, counterexamples have been found. We detect the sources of some unforeseen design errors. We, furthermore, give our interpretations of these errors.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification, H.3.4 Systems and Software

**Keywords and phrases** Mobile Robot, Robot Gathering, Formal Verification, Model Checking, Maude

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2017.12

## 1 Introduction

Theoretical research on distributed mobile robot focuses mainly on computability aspects; the goal is to determine whether a problem can be solved given some assumptions, such as synchrony, multiplicity detection and chirality. Various models and problems have been proposed for the last two decades (*e.g.* Suzuki and Yamashita first paper [18], or the book from Flocchini *et al.* [15]). A few major models have emerged (*e.g.* ASYNC, weak multiplicity detection) for which some of the main problems (*e.g.* gathering, exploration) are now fully understood. In this work, we would like to take a step back and look at what have been accomplished so far. Now that results are stable, it is the right time to spend some energy carefully reviewing the proposed algorithms.

There have been already some efforts to unify and formalize existing results [9, 10, 8, 14]. While earlier papers described algorithms based on a (potentially long and cryptic) list of rules (*e.g.* [4]), more recent publications usually describe algorithms based on mathematical abstractions and real pseudo-codes (*e.g.* [8]). Naturally, at the same time, proofs become also more formal. We believe this is going, of course, in the good direction. Being closer to the mathematical world also makes these algorithms/proof well suited to be checked systematically.


21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 12; pp. 12:1–12:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Formal verification and related work.** Classic distributed algorithms have been formally verified [19, 17, 13, 12]. We aim to obtain similar achievements for distributed robot algorithms. However, due to the mobility aspect, mobile robot algorithms are often complex, arguably even more complex than classic distributed systems. This inherent difficulty explains probably the limited number of attempts in obtaining formal verifications.

Recently Courtieu *et al.* [7] and Balabonski *et al.* [2] formally proved the correctness of two gathering algorithms using the Coq proof assistant. In both cases, robots move on the continuous 2-dimensional plane. The difference lies in the timing model; the first paper considers the semi-synchronous (SSYNC) model while the second studies the fully-synchronous (FSYNC) model (and remove some other assumptions). The asynchronous model (ASYNC) is not considered in these papers since the gathering problem is generally not solvable in ASYNC in continuous space (except for trivial cases).

Closely related to our current approach, Berard *et al.* [3] and Doan *et al.* [11] analyze the perpetual exploration algorithm described in [4]. This algorithm, while being quite simple (only 3 robots on the ring and 8 rules), was refuted. Both papers use similar techniques; they formally specify the algorithm and then encode in Linear Temporal Logic (LTL) the properties that should be satisfied. The first paper uses DiVinE and ITS tools, while the second one uses Maude.

This paper is somehow an extension of [11] to a different problem. Note however that this is not an easy generalization; our new formalization should accommodate any number of robots instead of only three. The previous studies [7, 2, 3, 11] basically consider either FSYNC or ASYNC without multiplicity assumption. The lack of multiplicity, of course, simplifies the model. Here we consider ASYNC and the possibility for multiple robots to be located on the same node.

**Context.** In this paper, we restrict our attention to discrete models only, and more specifically to the ring topology. About timing assumption, we consider the more general asynchronous model ASYNC. In addition, we take into account multiplicity assumption, which makes it much harder to formalize mobile robot algorithms. We present how to formalize a mobile robot algorithm as a state machine and then specify the state machine in Maude. Maude is a rewriting logic-based programming and specification language and equipped with a powerful system (or environment). Rewriting logic makes it possible to naturally specify dynamic systems, and the Maude system has an LTL model checker. We have demonstrated in [13, 12, 11] that Maude allows us to specify distributed algorithms/systems more succinctly than others. For instance, it supports *associative* and *commutative* operator attributes that are very necessary to concisely specify mobile robot algorithms as showing in [11].

We then use the Maude LTL model checker to formally verify an algorithm for robot gathering problem on ring enjoys desired properties. We focus on the gathering problem and analyze the algorithm proposed by D'Angelo *et al.* [8] as a case study. As the result of the model checking, counterexamples have been found. We detect the sources of unforeseen design errors. We, furthermore, give our explanations on these errors.

**Contributions.** The paper presents a study on how to specify and model check mobile robot algorithms. The contribution of this paper is the proof by example that formal methods must be used to verify distributed robot algorithms. Indeed, even algorithms described and proven using mathematical abstractions (may) still contain errors. While some of them are minor and could have been detected by a careful reader (simple typos), some errors would have been almost *impossible* to detect without model-checking. Said differently, we believe that

informal and semi-formal mathematical proofs are not enough for this kind of algorithms. The complexity of analyzing all situations may be too high for human brains.

Two main contributions are: (1) a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions – our model is general enough and could be applied to other problems (in the ring); (2) a refutation by model checking that the algorithm enjoys desired properties – in detail, the algorithm contains design errors that prevent robots from gathering into one location.

The additional contributions are a preliminary set of Maude modules that could be re-used for future verifications and the interpretations of the errors found. While it may not be straightforward to understand Maude formalism, we hope that it may still be useful for other people.

**Outline.** Section 2 describes the model, problem, and the main ideas of the algorithm under study. Section 3 presents how we formalize and then specify the system in Maude. Section 4 describes our model checking of the algorithm, showing and explaining the result. Section 5 finally concludes the paper.

## 2 Robots Gathering in the Ring under ASYNC

As mentioned in the Introduction, we consider the classic gathering problem in the ring under the asynchronous scheduler (ASYNC). We analyze the most general algorithm that solves the problem for (almost) all initial configurations. The considered algorithm is said to be general in a sense that it solves the problem for all valid (*i.e.* without multiplicity) initial configurations outside of $NG \cup SP4$, where (1) $NG$ is the set of $N$on-$G$atherable configurations (such as periodic configurations), from which it is impossible to gather robots, and (2) $SP4$ is the "small" set of $SP$ecial configurations with 4 robots from which it is still unknown whether it is possible to gather robots (Bonnet *et al.* gave a partial answer [5]).

In the remainder of this section, we succinctly present the model, the problem, and the algorithm under study. For each part, a more complete description can be found in the original paper [8].

### 2.1 Computational Model

This model description is adapted from [5] for our specific context. The ring is *anonymous*, that is, there is neither node nor edge labeling. The robots are *identical*, *i.e.*, they are indistinguishable and all execute the same algorithm. Moreover, the robots are *oblivious* and *disoriented*, meaning that they have no memory of past actions, and they share no common orientation (no chirality).

The robots cannot explicitly communicate, but have the ability to sense their environment and see the relative positions of the other robots, in their local coordinate system. We assume the global weak multiplicity detection; each robot can distinguish whether a node is empty, occupied by one robot, or more than one robot. When there is strictly more than one robot, we use the term *multiplicity*. Robots follow a three-phase behavior: *Look*, *Compute*, and *Move*. During its Look phase a robot takes a snapshot of all robots' positions. The collected information (position of the other robots in the egocentric view) is used in the Compute phase during which the robot decides to move or stay idle. In the Move phase, the robot may move to one of the two adjacent nodes, as computed in the previous phase. The moves are assumed to be instantaneous which means that, during a Look phase, robots can be located on nodes only.

The computational model we consider is the classic asynchronous ASYNC model [15]. It means that, the start and duration of each Look-Compute phases and the start of each Move phase of each robot are arbitrary and determined by an adversary. Note that it is possible for a robot to make a move based on a previously observed configuration which is not the current one anymore (*e.g.* if its Look phase occurred before the Move phase of another robot).

A move that has been computed (during a Compute phase) but not yet executed (in the subsequent Move phase) is called a *pending move*.

## 2.2 Gathering Problem

The gathering problem requires each robot to terminate on the same node. The problem is solved if all robots are on the same location and there is no pending move.

## 2.3 Gathering Algorithm

This paper analyzes the algorithm [8] for robot gathering. The algorithm executes these four phases sequentially:

1. Starting from an initial configuration without multiplicity, the algorithm executes a procedure MULTIPLICITY-CREATION that creates either one or two symmetric multiplicities.
2. A second phase named COLLECT consists in moving all but four robots in the previously created multiplicities.
3. A third phase called MULTIPLICITY-CONVERGENCE makes the two multiplicities to merge into a single one.
4. Finally the phase CONVERGENCE allows the remaining single robots to join the unique multiplicity, which concludes the gathering.

This is a short (partially incorrect) summary. In some *rare* cases, the sequence of four phases may be temporarily broken. (*e.g.* the system may go back to the COLLECT phase while executing the CONVERGENCE phase). But eventually all robots should gather at the same location.

The algorithm contains some specific subroutines for configurations with four or six robots.[1] At the moment, we decided not to include them in our analysis; as in the paper, they could be dealt separately.

**Plaintext vs. Pseudo-code.** In [8], the algorithms are described and explained in plaintext and also given in term of pseudo-codes. We think that the pseudo-code version contains less ambiguities than the plaintext version. In (pseudo-)code, there is usually no place for interpretation; it is thus either correct or incorrect. Since our goal is to formally model-check, we believe that it makes more sense to base our analysis on the most formal available version. That is why this paper analyzes the pseudo-code version of the algorithms.

This is certainly an arguable decision. Indeed, some of the errors, at least the ones from Section 4.3, do not exist in the plaintext description of the algorithms. Other detected errors may or may not exist in the plaintext version. We can not conclude anything about the correctness of the plaintext algorithm since it is subject to interpretation.

[1] The precise algorithm is not given for four or six robots, but refers to other papers.

# 3 Formal Model for Mobile Robot Algorithms

In this section, we propose a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions. Unavoidably, multiplicity and asynchrony make arduous to formalize the systems. We pay much attention to this problem and solve it in our model. To describe the model, we use state machines. A state machine consists of a set $S$ of states, some of which are initial states, and a binary relation $T \subseteq S \times S$. Each element $(s, s') \in T$ is called a state transition from $s$ to $s'$. We then transfer this model into Maude specification language. In Maude, the basic units of specifications and programs are modules. A module contains syntax declarations, providing suitable language to describe a system. There are two kinds of modules: functional modules and system modules. Functional modules are those in which data structures, such as pair and sequence, are specified, and system modules are those in which systems, such as distributed systems, are specified. A distributed system is formalized as a state machine and then the state machine is specified in Maude as a system module.

For these systems, a state of the system is called a configuration. A configuration is described in terms of a view starting from any robot and traversing the ring in one arbitrary direction. When a robot wakes up, it takes the snapshot of the current configuration of the system, and computes a move (called a computed move) based on this snapshot. The computed move is either staying idle or moving to one of its adjacent nodes. In the latter case, it moves to the adjacent node, eventually. In the following part, Maude notation is used to describe state machines. We consider how to express a state and how to describe an event as a state transition.

## 3.1 State Expressions

We denote a robot as a pair $\langle I, P\rangle$, where $I$ denotes the size of the interval[2] between it and the next robot, and $P$ denotes the computed move. The value of $P$ could be $nil$, $fc$ or $fc-$ ($fc$ stands for $f$ollowing the $c$onfiguration). $nil$ means that the robot has no pending move (*i.e* last computed move was idle). $fc$ (resp. $fc-$) means that the robot has a pending move to the adjacent node located after it (resp. before it) following the direction of the configuration. Initially, the computed move of each robot is $nil$. If $P$ is $nil$, the robot may be activated and will compute a new move and update $P$ accordingly. If $P$ is not $nil$, the robot may be activated and will execute the move and update $P$ to $nil$ after the move. We use the sort[3] *Pending* to denote computed moves and the sort *Pair* to denote pairs. They are expressed by the following operators that are constructors as specified with *ctor*.

op $\langle$_, _$\rangle$ : Int Pending $\rightarrow$ Pair [ctor] .

The sort *Int* is used for denoting integers. The operator $\langle$_, _$\rangle$ is used to construct *Pair*. For $c_1 \in$ *Int*, $c_2 \in$ *Pending*, $\langle c_1, c_2\rangle \in$ *Pair*.

A configuration is expressed as a sequence of pairs. It contains the information about the locations of all robots and their states. The corresponding sort is *Config*. Configurations are defined by the following operators.

[2] An interval is a maximal set of empty consecutive nodes.

[3] The types of data are called *sorts*. A sort denotes the set of elements in the same type. For example, the sort Nat is used to denote the set of Natural numbers. A sort is a subsort of another sort if and only if the set denoted by the former is a subset of the one by the latter, and the latter is called a supersort of the former.

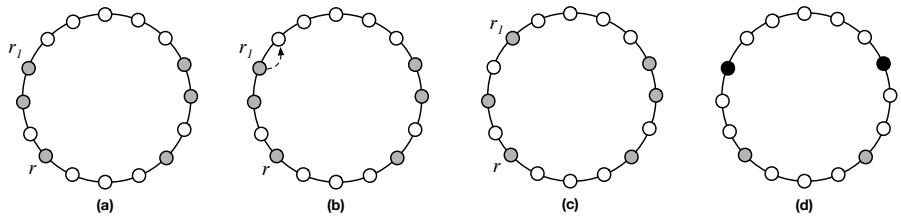


**Figure 1** Some configurations. A dashed arrow represents a pending move. Black nodes represent multiplicities.

subsort Pair < Seq .
op empS : → Seq [ctor] .
op _ _ : Seq Seq → Seq [ctor assoc id: empS] .
op {_} : Seq → Config [ctor].

where the sort *Seq* is used for sequences of pairs. *empS* denotes the empty sequence of pairs. *Seq* is a supersort of *Pair*, which means that each *Pair* is treated as the singleton sequence only consisting of the pair. The juxtaposition operator _ _ is used to construct non-trivial sequences of pairs. For $c_1$, $c_2 \in$ *Seq*, $c_1\ c_2 \in$ *Seq*. The juxtaposition operator _ _ is associative as specified with *assoc*, and *empS* is an identity of the operator specified with *id: empS*.

A configuration is of the form {_} of a sequence. States of the system are expressed as terms of the sort *Config*. A term of a sort $S$ is a variable of $S$ or $f(t_1, \ldots, t_n)$ if $f$ is an operator declared as $f : S_1 \ldots\ S_n \to S$ $(n \geq 0)$ and $t_1, \ldots, t_n$ are terms of $S_1, \ldots, S_n$. If $f$ has any underscores _, such as $\langle \_, \_ \rangle$, then a different notation than $f(t_1, \ldots, t_n)$ is used, such as $\langle I, P \rangle$ that is a term of the sort *Pair*, where $I$ is term of the sort *Int* and $P$ is a term of the sort *Pending*. Constructor terms are those consisting of constructors only and variables. Ground term are those having no variables. Ground constructor terms hence are those composed of constructors only and no variables . Ground constructor terms of the sort *Config* express concrete states of the system. For example, the initial configuration of the system as shown in Fig. 1(a) could be expressed as the view starting from the robot $r$ in clockwise order, $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. Let us assume that robot $r_1$ is activated, takes a look at the configuration, and computes a move. Assuming that the move is to move to the node located after it, the system reaches the configuration as shown in Fig. 1(b). It is expressed as $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. If the robot $r_1$ is activated again, it executes the move; the system becomes as shown Fig. 1(c) which is expressed as $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. A robot is not allowed to look at the second element of the pairs of other robots and it calculates a move based on its own view of the system. For example, the view of robot $r_1$ in Fig. 1(a) could be either $\{\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\}$ or $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\}$ without knowing anything about clockwise order. It is worth noting that this allows us to guarantee that robots have no sense of direction and do not know the pending moves of other robots.

Due to the multiplicity assumption, it is possible that a robot moves to a node that is occupied by other robots. The node is, or becomes a multiplicity. There may be more than one of such robots in one multiplicity. Since the robots are anonymous, we can denote all of them by a pair $\langle I, P \rangle$ in which the value of $I$ is set to the negative of the additional number of robots located on the multiplicity ($-3$ indicates 3 additional robots, which means a multiplicity of 4 robots). Note that this notation allows us to represent the exact number

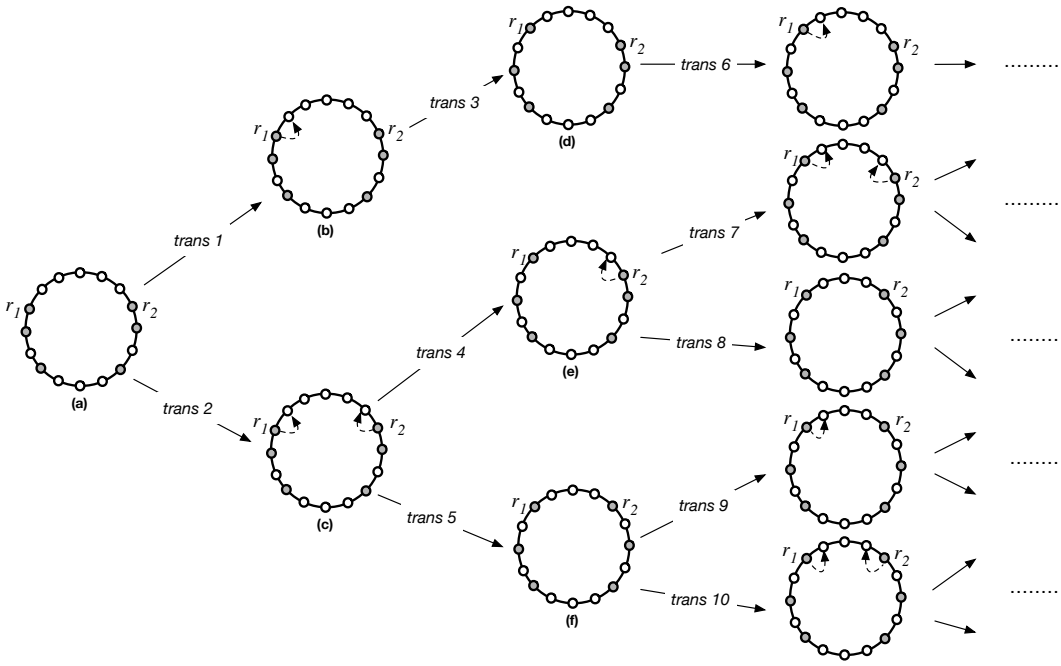


**Figure 2** A transition graph of one specific initial configuration (a).

of robots in multiplicities. But robots do not have access to this information; they can only know if there is a multiplicity (*i.e.* a negative number). Our encoding allows a simple conversion to consider global strong multiplicity detection. For instance, the configuration as shown in Fig. 1(d) assuming that there are two robots in each multiplicity, is expressed as $\{\langle 2, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 2, nil\rangle\langle 3, nil\rangle\}$. We use this encoding to match as closely as possible the definitions introduced in [8].

## 3.2 State Transitions

Because the *Compute* phase uses the snapshot of the system taken in the *Look* phase as input and a robot does not perform any movements during two phases, to model the system, we combine the two phases into one called the *Look-Compute* phase in which a robot takes the snapshot of the system and computes a move. When either (1) a robot takes the snapshot of the system and then computes a move, or (2) a robot executes its pending move, the current configuration of the system changes to another. Such changes are called a state transition (or a transition). A transition is expressed as a pair $(l, r)$, where $l$ and $r$ are configurations.

Let us examine the following scenario. Given an initial configuration as shown in Fig. 2(a) and assumed that both robots $r_1$ and $r_2$ are allowed to move, it may happen that only one robot (assuming $r_1$) looks at the system and computes a move, or both $r_1$ and $r_2$ do. In the former case, the configuration of the system is transferred to the one as shown in Fig. 2(b). The transition is named $trans1$ and expressed by the pair $(\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle, \langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle)$. In the latter case, the configuration of Fig. 2(c) is established. The configuration of Fig. 2(d) is obtained after $r_1$ in the configuration of Fig. 2(b) executes its pending move. The configuration of Fig. 2(e) and Fig. 2(f) are obtained from the configuration of Fig. 2(c). The graph in Fig. 2 shows possible transitions from the initial configuration. A sequence of transitions

starting from an initial configuration, e.g $trans1$, $trans3$, $trans6$, ..., is called a possible execution. There may exist more than one execution from a given initial configuration.

We describe the actions of robots as transition rules. A transition rule is described in the form of a rewrite rule. Each rewrite rule is defined only over *Config* that does not have any sub-sorts and in the form L ⇒ R such that L only consists of constructors and variables. We give here a simple example to explain how an action can be expressed as a transition rule. The following transition rule describes the action corresponding to a robot executing its pending move when there is no multiplicity in the system.

crl [fc-pending] : {S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2} ⇒ {S1 ⟨ I1 + 1, P ⟩ ⟨ I2 - 1, nil ⟩ S2}
if nonMul({S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2}).

where $S1$, $S2 \in$ *Seq*, $I1$, $I2 \in$ *Int* and $P \in$ *Pending* are variables of those sorts; The function *nonMul* returns *true* when the configuration has no multiplicity and *false* otherwise.

The above rule is a conditional writing rule and the condition is specified in the $if$ part. The rule then will be applied if the condition is satisfied. The configuration {S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2} expresses any state such that the robot ⟨ I2, fc ⟩ holds a pending move $fc$ and the robot before it is ⟨ I1, P ⟩. Such a state may have some more robots before and after the two robots that are expressed as $S1$ and $S2$, respectively. The ground constructor term $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expresses the state as shown in Fig. 2(b). There is no multiplicity in this configuration. The left-hand side of the above rewrite rule *fc-pending* matches this ground term by substituting *S1, I1, P, I2* and *S2* with $\langle 1, nil\rangle$, 0, $nil$, 5 and $\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle$, and the rewrite rule can be applied to the term, changing it to $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expressing the state as shown in Fig. 2(d). In this way, a rewrite rule expresses a set of state transitions.

To make T, a set of transitions $(s, s')$, from rewrite rules L ⇒ R, let $\sigma$ be a substitution from the variables in L to appropriate constructor terms, $\sigma(L)$ is a constructor term, but $\sigma(R)$ is not necessarily, and then it is necessary to reduce $\sigma(R)$ with equations. Let $nf(t)$ be the term obtained by reducing $t$ with equations. So, $(\sigma(L), nf(\sigma(R)))$ is a state transition obtained from L ⇒ R. Let $\sigma(L) \Rightarrow nf(\sigma(R))$ be called a ground instance of L ⇒ R.

▶ **Definition 1** ($TR_{RS}$)**.** Let $TR_{RS}$ be the set of all ground instances of the all transition rules.

## 3.3 Formal Model

We formalize a mobile robot system as a state machine. The state machine includes the set of all possible states as the set of all ground constructor terms $S_{RS}$, the set of initial states $I_{RS}$ and the binary relation over states $T_{RS}$. $I_{RS}$ is a subset of $S_{RS}$ such that for each state $s \in S_{RS}$, there is no multiplicity and the configuration does not belong to $NG \cup SP4$. $T_{RS}$ is the binary relation over $S_{RS}$ made from $TR_{RS}$.

▶ **Definition 2** ($M_{RS}$)**.** The state machine formalizing a mobile robot system is $M_{RS}$, where

1. $S_{RS}$ is the set of all ground constructor terms whose sorts are *Config*;
2. $I_{RS}$ is a subset of $S_{RS}$ such that $(\forall s \in I_{RS})$ (numMul($s$) = 0) and (not ng&sp4($s$));
3. $T_{RS}$ is the binary relation over $S_{RS}$ defined as follows:
   {(l, r) | l ⇒ r $\in TR_{RS}$}.

The function numMul counts the number of the multiplicities in the system and the function ng&sp4 returns *true* when a configuration is in $NG \cup SP4$ and *false* otherwise.

We specify this formal model in Maude specification languages. Note that our specification is coherent [6]. The source files are available for download [1].

## 4 Model Checking the Algorithm

Model checking is a verification technique that explores all possible system states and checks whether a desired property that should be satisfied by an algorithm is satisfied. The desired property is required to be formally expressed. A model checker then verifies whether the formula is satisfied for all possible executions. If the formula is not satisfied, a counterexample is found. We specify the formal model of the system in Maude. We then apply Maude LTL model checker to formally verify the algorithm. The original paper [8] has given very important lemmas, such as *Lemma 5, 6, and 7*, that state properties that need to be satisfied at the end of each phase. These lemmas are used to model check the algorithm. We have formally expressed these lemmas as LTL formulas [16]. We give here the formalization of *Lemma 6* as an example. *Lemma 6* states a property that must be satisfied at the end of the COLLECT phase. Namely, it states that the configuration obtained at the end of the COLLECT phase contains two multiplicities and satisfies a condition (called located condition) that the configuration needs to be in some specific configurations in which robots are located in some specific locations. Note that the COLLECT phase can start only if the initial configuration is symmetric or at one step from specific symmetric configurations. To model check this lemma, we expressed it as an LTL formula. We define the atomic propositions *endOfColl* and *coll* as follows:

$C \models \text{endOfColl} = \text{checkOfColl}(C)$ .

$C \models \text{coll} = \text{checkColl}(C)$ .

$\text{checkColl}(C) = \text{checkAllowedSym}(C) \text{ and } (\text{numMul}(C) == 2) \text{ and } \text{checkCondition}(C)$ .

The function *checkOfColl* checks whether a system state C is at the end of the COLLECT phase. The atomic proposition *endOfColl*, thus, is true if and only if the COLLECT phase phase has just finished. The function *checkAllowedSym* checks whether a configuration is an allowed symmetric configuration. The function *checkCondition* returns *true* when the configuration satisfies the located condition and *false* otherwise. The atomic proposition *coll*, thus, is *true* when the configuration is an allowed symmetric configuration containing two multiplicities[4] and satisfies the located condition and *false* otherwise.

The lemma then is formally expressed as an LTL formula as follows.

$\text{lemma6} = \Box\ (\text{endOfColl} \rightarrow \text{coll}) \wedge \Diamond\ \text{endOfColl}$ .

where $\Box$ stands for always (globally) and $\Diamond$ stands for eventually (in the future).

Intuitively, the formula states that it is always true that the phase COLLECT will finally terminate and whenever the phase has been just over, then *coll* is true. This means that the *Lemma 6* is satisfied at the end of the COLLECT phase.

This formula is used to conduct the model checking for the algorithm. To model check, we separate the formula into two sub-formulas and model check them separately in order to easily detect the source of errors (if some are found). Namely, we use the two following formulas:

$\text{lemma6-1} = \Diamond\ \text{endOfColl}$ .

$\text{lemma6-2} = \Box\ (\text{endOfColl} \rightarrow \text{coll})$ .

As the result of the model checking, counterexamples are found. A counterexample is of the form of a possible execution: it includes all visited states and the sequence of transition rules applied. This helps to analyze counterexamples to detect the source of the detected errors. We present two counterexamples as follows:

[4] The mathematical notation == stands for equivalence.

The first one results from the model checking of the formula lemma6-1.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-1) .
rewrites: 89524096 in 40088ms cpu (40173ms real) (2233163 rewrites/second)
result ModelCheckResult: counterexample(
      {{< 0,nil > < 1,nil > < 0,nil > < 3,nil>
        < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
      {{< 0,nil > < 1,nil > < 0,nil > < 3,nil > < 0,fc >
        < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
      {{< 0,nil > < 1,nil > < 0,nil > < 4,nil > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo2}
      {{< 0,nil > < 1,nil > < 0,nil > < 4,fc- > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
      {{< 0,nil > < 1,nil > < -1,nil > < 5,nil > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},deadlock})
```

The counterexample shows that the system falls into a deadlock state. Indeed, the configuration $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 0, nil\rangle\}$ belongs to the kind of configurations that need to be handled by the COLLECT phase. However, the COLLECT phase could not deal with it. Thus, the system is not able to reach a valid state at the end of the COLLECT phase. Analyzing this counterexample, we detect the error, which is described later in Section 4.1. This also means that the algorithm fails in gathering all robots in the same location.

The second counter-example results from the model checking of the formula lemma6-2.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-2) .
rewrites: 15982060 in 7064ms cpu (7114ms real) (2262435 rewrites/second)
result ModelCheckResult: counterexample(
    {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
      < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
    {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
      < 0,fc > < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
    {{< 0,nil > < 1,nil > < 0,nil > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo1}
    {{< 0,nil > < 1,nil > < 0,fc- > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
    {{< 0,nil > < 0,nil > < 1,nil > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'ofColl})
```

This counterexample occurs because the state

$$\{\langle 0, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 0, nil\rangle\}$$

is at the end of the COLLECT phase, but the atomic propositions *coll* returns *false*. Specifically, at end of the COLLECT phase, the configuration does not contain two multiplicities and satisfy the located condition.

Since counterexamples are found, the lemma does not hold. The remainder of this section reports on some errors that we have found. In addition, we also give our opinions about the origins of these errors.

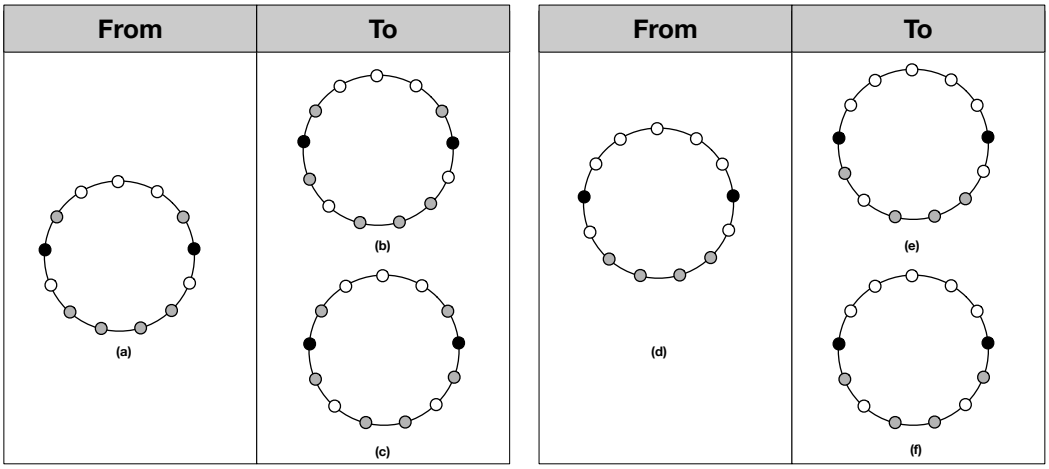


**Figure 3** Expected executions for the two specific symmetric configurations with two multiplicities.

## 4.1 Omission of Special Cases

We report here the error that corresponds to the first counterexample. The error occurs when the algorithm deals with symmetric configurations with two multiplicities. Single robots in such configurations should move such that the configuration eventually reaches a symmetric one with (i) size nodes occupied, (ii) two multiplicities, (iii) two robots adjacent to these multiplicities, and (iv) other robots in specific locations. However, it is not straightforward to design a strategy for these robots. Two examples are shown in Fig. 3. For the configuration of Fig. 3(a), the two symmetric robots are expected to move such that either the configuration of Fig. 3(b) or Fig. 3(c) is obtained. In the same way, the configurations of Fig. 3(e) and Fig. 3(f) are obtained from the configuration of Fig. 3(d).

Unfortunately, a counterexample is found. Our model checking detects that the algorithm does not work correctly for the configuration of Fig. 3(d). The two single robots do not move as expected. The same configuration is obtained instead of the configurations of Fig. 3(e), 3(f). Indeed, the algorithm only works correctly for configurations in which there are two single robot adjacent to the two multiplicities, *e.g.* the configurations similar to the one of Fig. 3(a), but it does not work for configurations such as the one of Fig. 3(d).

This error exists because some cases were missing. For instance, the configurations as the configuration of Fig. 3(d) are not considered. That is why the algorithm is not able to deal with them. This would be very difficult to detect without the help of an automatic tool. The error leads to the fact that the system will fall to a deadlock state and all robots in the system are unable to locate at the same location. This error could be fixed by finding a strategy to deal with these case. However, we need to take care of the fact that the new strategies may be conflicting with exiting strategies.

## 4.2 Design Errors (difficult to detect by mathematical proof)

This section report errors of a different kind compared to the one of Section 4.1. One of the main issue to be handled by the algorithm concerns symmetric configurations in which two symmetric robots are supposed to move symmetrically. Let us explain the idea with an example. For the symmetric configuration as shown in Fig. 4(a), the two symmetric robots $r$ and $r_1$ are allowed to symmetrically move. It may happen that both $r$ and $r_1$ move and

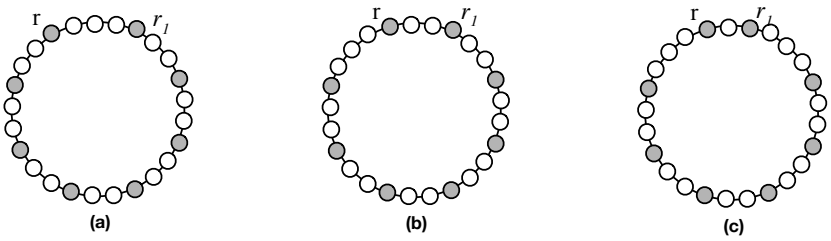


**Figure 4** Three configurations where (a) is some initial configuration, (b) is the configuration obtained if only one robot moves, and (c) the one obtained if both symmetric robots move.

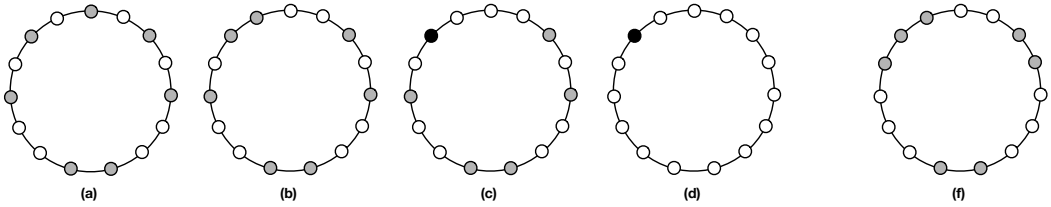


**Figure 5** Five configurations. The gathering algorithm should follow the sequence (a), (b), (c), (d), but the configuration (f) is actually obtained from (b).

the configuration of Fig. 4(c) is obtained. It may also happen that only $r$ moves, while $r_1$ already computed the move, but has not yet moved (means that it holds a pending move) or $r_1$ has not yet performed its Look-Compute phase and the configuration of Fig. 4(b) is obtained. This configuration is an asymmetric configuration that may contain a possible pending move (only robot $r_1$ knows if it has already computed its move; other robots do not know it). The procedures CHECK-REDUCTION and PENDING-REDUCTION are designed to deal with this case. The robots in such configuration are supposed to detect this situation and the robot $r_1$ is expected to move in order to reach the configuration of Fig. 4(c). It is not so difficult to design procedures that let robots in these configurations to recognized these configurations and move as expected. However, the problem is when these procedures are included in the entire program.

As written in [8], the procedure CHECK-REDUCTION also recognizes some other configurations where PENDING-REDUCTION should not be applied. One of them is the configuration depicted in Fig. 5(b). The authors use this configuration as an example to explain the algorithm (Figure 3 in [8]). The algorithm is expected to work as follows: since the configuration of Fig. 5(a) is symmetric with one robot on the axis, the robot on the axis has to move, obtaining the configuration of Fig. 5(b). This configuration is asymmetric and and contains only one *supermin*[5]. There is one important point emphasized by authors: the robots can recognize that there are no pending moves in this configuration. Therefore, the unique supermin is reduced until a multiplicity is created. In this example, the configuration Fig. 5(c) is obtained, and then all robots join the unique multiplicity one-by-one, until achieving the gathering as in Fig. 5(d). However, the algorithm actually works as follows: from configuration of Fig. 5(b), the robot which is supposed to create a multiplicity does not move. Instead two other robots move and we obtain the configuration of Fig. 5(f) (instead of Fig. 5(c)). Checking each step of the execution and the transition rules applied, we discover that the source of this error: Robots do not recognize the correct type

[5] The definition of *supermin* is given in [8]. One characteristic of a *supermin* is that it is smallest interval.

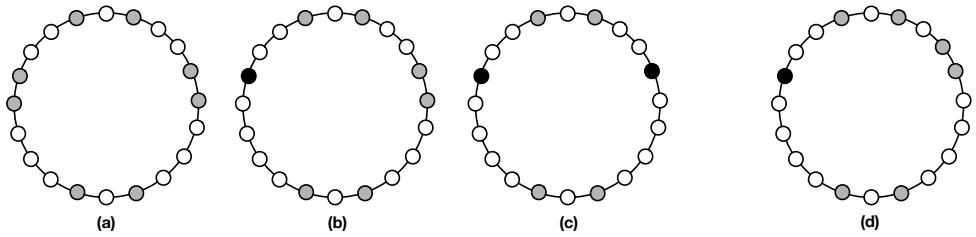


**Figure 6** Four configurations. The gathering algorithm should follow the sequence (a), (b), (c), but the configuration (d) is actually obtained from (b).

of the configuration of Fig. 5(b), which is classified into a group named W7 that performs the procedure CHECK-REDUCTION and PENDING-REDUCTION. When executing the procedure CHECK-REDUCTION on this configuration, robots incorrectly categorize it as an as asymmetric configuration with possible pending moves. Then the PENDING-REDUCTION is executed while it should not be for this configuration. Thus, the two symmetric robots decide to move. The configuration of Fig. 5(f) is obtained. This means that the CHECK-REDUCTION and PENDING-REDUCTION procedures are not correct. This error would be very difficult to detect manually. Indeed, the procedures are correct for the configurations for which they are supposed to be used. It is only incorrect because it is also applied to configurations that are not supposed to use these procedures[6].

The second error is in the COLLECT phase. When entering into this phase, the configuration of the system belongs to one of three categories. One of them is called COLL-A-1. They are asymmetric configurations with one multiplicity that can be obtained from symmetric configurations and while satisfying also some other conditions. To simulate how the algorithm works, the authors give the scenario depicted in Fig. 6. The configuration of Fig. 6(b) is in COLL-A-1. It is at one move from the symmetric configuration of Fig. 6(a). When the configuration is in COLL-A-1, the algorithm checks whether the current configuration satisfies some conditions. If the conditions are satisfied, it moves the robot and re-establish the previous axis of symmetry, leading to a symmetric configuration with two multiplicities. In this specific case, the configuration of Fig. 6(c) is obtained.

Unfortunately, instead of moving the robot and re-establishes the previous axis of symmetry, the algorithm moves both adjacent robots in the same direction. That means the symmetric configuration with two multiplicities is not obtained. In this specific case, the configuration of Fig. 6(d) is obtained. This error is somehow similar to the previous design error, but at a different level; robot instead of configuration. Here, the robot that is supposed to move (from the plaintext description of the algorithm) really moves, but an additional robot also moves.

These design errors prevent the robots from gathering. Said differently, the algorithm fails in gathering all robots in one location. Both errors could certainly be fixed by including additional tests before computing the moves. However adding these tests may lead to other problems and is therefore not straightforward.

[6] As explained in Section 2, we analyze the pseudo-code of the algorithms [8]. It is unclear whether such error exists in the informal plaintext description of the algorithm.

## 4.3 Some minor errors

We also want to report some other small errors. They could be detected by carefully checking the pseudo-code. However, it may be difficult to find where they are in the code. Fortunately, we can also detect them by model checking.

### 4.3.1 Minor "typo" error

If a configuration is periodic, it is impossible to gather all robots to one location. Therefore, it is important to detect whether a configuration is periodic or not. To check the periodicity, the authors give the following procedure:

**Input**: a configuration $C = (q_0, q_1, ..., q_j)$ .

**Output**: **true** if $C$ is periodic, **false** otherwise.

1. *periodic* := **false** .
2. **for** $i := 0, 1, ..., j$ **do if** $C = C_i$ **then** *periodic* = **true**.
3. **return** *periodic* .

where *periodic* is a boolean variable. It is expected to return *true* if the configuration $C$ is periodic, *false* otherwise. We discover that *periodic* returns *true* also for aperiodic configurations. This is obviously wrong. The source of the error is that the condition $C = C_0$ is true for any configuration $C$. Thus, the procedure always returns *true* for any input configuration $C$. One needs to simply start iterations from $i = 1$ instead of $i = 0$.

### 4.3.2 Error of inattention

The second error detected is in the main procedure for phase MULTIPLICITY-CREATION whose purpose is to create one multiplicity or two symmetric multiplicities. Since this is the first phase of the algorithm, which deals with initial configurations, all possible initial configurations are partitioned into seven groups. One of them is called *W6*. In this case, the procedure is given as follows:

**Input**: CT, $C = Q(r) = (q_0, q_1, ..., q_j)$.

**Case** $CT = W6$

1. $C' := (q_0 + 1, q_1 - 1, ..., q_j)$.
2. **if** $C' = \overline{C'}$ *and* $q_0$ *is odd* **then** move towards $q_0$;
3. **else**
   $C'' := (q_0, ..., q_{j-1} - 1, q_j + 1)$.
   **if** $C''_j = \overline{C''_j}$ *and* $q_j$ *is odd* **then** move towards $q_j$;

where $C = Q(r) = (q_0, q_1, ..., q_j)$ is the configuration that is perceived by robot $r$. $\overline{C}$ corresponds to $(q_0, q_j, q_{j-1}, ...q_1)$ in the case where C $= (q_0, q_1, ..., q_{j-1}, q_j)$.

This code is supposed to handle asymmetric configurations that could have been obtained from a symmetric configurations with an odd number of nodes and a node-edge symmetric axis. In the above code, $C'$ (or $C''$) is the configuration of the symmetric configuration. The intention of the authors is to move a robot $r$ in order to reach a new symmetric configuration with the same original axis. One example taken from the Appendix of [8] is given in Fig. 7. The configuration of Fig. 7(b) can be obtained from the symmetric configuration of Fig. 7(a). Thus, the robot $r$ is supposed to move and the configuration of Fig. 7(c) is obtained.

However, the algorithm does not make the robot $r$ to move. We found that the error lies in the conditions: $q_0$ *(or* $q_j$*) is odd*. Let us take a look at symmetric configurations (e.g as shown in Fig. 7(a)) with odd number of nodes and one axis passing through an edge, we can see that the interval crossed by the axis is on an odd interval. That means that $q_0 + 1$ (or $q_j + 1$) is odd, but not $q_0$ (or $q_j$). The error can be fixed by updating the conditions to test if $q_0 + 1$ (or $q_j + 1$) is odd. This error was probably made due to the confusion between the configurations $C$ and $C'$ (or $C''$).

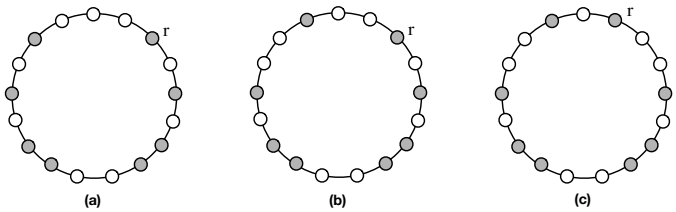


**Figure 7** Three configurations, where (a) may lead to (b), and (c) presents the expected behavior of the algorithm for the specific asymmetric configuration (b).

## 4.4 Summary of model checking

It is very common to use case analysis techniques to tackle non-trivial problems, such as robot gathering. The authors in [8] have split the problem into many cases and used different strategies to deal with them. In detail, they have partitioned not only the initial configurations, but also the configurations in each phase. Since the authors need to consider a large number of cases, there is a risk of missing some cases. Moreover, since different strategies are used for each case, there could be some conflicts between these strategies. To prove the algorithm, the authors have to consider all possible cases. However, it is exhausting for a person to figure out all possible executions. Fortunately, a model checker strongly supports to automatically check all possible executions. We have showed how to formally express the desired properties as LTL formulas and conduct the model checking of the algorithm. The model checking has found counterexamples. Analyzing these counterexamples, we found the source of some errors that would be difficult to detect by handmade proof. These errors make the algorithm fail in gathering all robots in one location. We also gave some explanations about why such errors may have occurred. We hope it may be useful to avoid them in the future.

# 5 Conclusion

How to formalize and model check a new software system is always challenging. We show in this paper how to formalize and model check a new form of distributed system. We have demonstrated the usefulness of model checking techniques to formally verify mobile robot algorithms. Although informal proofs have been given in [8] to guarantee the correctness of the algorithm, there remain some mistakes that are subtle and not easy to find by carefully checking the algorithm, even by experts. We want to emphasize that our goal is not to blame the authors or reviewers of the paper. When reading the paper, we also missed most of these errors. But it means that it is indeed difficult and therefore we should really consider using formal methods to check such algorithms.

## References

1 Our Maude source files. URL: `https://figshare.com/s/3c0a6fdb0e4fec3b0b08`.

2 T. Balabonski, A. Delga, L. Rieg, S. Tixeuil, and X. Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 10083 of *LNCS*, pages 7–19. Springer, 2016.

3 B. Bérard, P. Lafourcade, L. Millet, M. Potop-Butucaru, Y. Thierry-Mieg, and S. Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.

4 L. Blin, A. Milani, M. Potop-Butucaru, and S. Tixeuil. Exclusive perpetual ring exploration without chirality. In *Proceedings of the 24th International Symposium on Distributed Computing*, volume 6343 of *LNCS*, pages 312–327. Springer, 2010.

5 F. Bonnet, M. Potop-Butucaru, and S. Tixeuil. Asynchronous gathering in rings with 4 robots. In *Proceedings of the 15th International Conference on Ad-hoc, Mobile, and Wireless Networks*, volume 9724 of *LNCS*, pages 311–324. Springer, 2016.

6 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

7 P Courtieu, L. Rieg, S Tixeuil, and X. Urbain. Certified universal gathering in $r^2$ for oblivious mobile robots. In *Proceedings of the 30th International Symposium on Distributed Computing*, volume 9888 of *LNCS*, pages 187–200. Springer, 2016.

8 G. D'Angelo, G. Di Stefano, and A. Navarra. Gathering on rings under the look–compute–move model. *Distributed Computing*, 27:255–285, March 2014.

9 G. D'Angelo, G. Di Stefano, A. Navarra, N. Nisse, and K. Suchan. Computing on rings by oblivious robots: A unified approach for different tasks. *Algorithmica*, 72(4):1055–1096, 2015.

10 G. D'Angelo, A. Navarra, and N. Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 28(1):17–48, 2017.

11 H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of a mobile robots perpetual exploration algorithm. In *Proceedings of the 6th International Workshop on Structured Object-Oriented Formal Language and Method (SOFL+MSVL 2016)*, volume 10189 of *LNCS*, pages 201–219. Springer, 2017.

12 H. T. T. Doan, F. Bonnet, and K. Ogata. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In *Proceedings of The 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, pages 1586–1596, 2017.

13 H. T. T. Doan, W. Zhang, M. Zhang, and K. Ogata. Model checking chandy-lamport distributed snapshot algorithm revisited. In *Proceedings of the 2nd International Symposium on Dependable Computing and Internet of Things*, pages 7–19, 2015.

14 P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

15 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool Publishers, 2012.

16 Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

17 I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

18 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

19 T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5):341–358, 2011.