An Environment for Specifying and Model Checking Mobile Ring Robot Algorithms

Ha Thi Thu Doan[1](✉), Adrián Riesco[2], and Kazuhiro Ogata[1]

[1] Japan Advanced Institute of Science and Technology, Ishikawa, Japan
{doanha,ogata}@jaist.ac.jp
[2] Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es

Abstract. An environment for specifying and model checking mobile robot algorithms on rings (or mobile ring robot algorithms) is proposed. We have developed the Maude Ring Specification Enviaude RSE), a specification environment that explicitly supports ring-shaped networks. Maude RSE is implemented on top of Maude, a rewriting logic-based specification language. The underlying key behind the tool is pattern matching between ring patterns and ring instances, called "ring pattern matching." Because rings are not commonly available data structures in any existing specification language, we encode ring patterns as sets of sequence patterns and simulate ring pattern matching by pattern matching between sets of sequence patterns and sequence instances, which is proven correct and transparent to Maude RSE users. The advantages of Maude RSE are demonstrated by case studies analyzing exploration and gathering algorithms.

Keywords: Distributed mobile robot system · Ring discrete model · Specification environment · Formal verification · Model checking

1 Introduction

The past two decades, theoretical computer science has seen the rapid growth and development of distributed computing by mobile entities. Recent developments focus on models and algorithms for autonomous mobile robots that self-organize and cooperate in order to achieve global goals. Autonomous mobile robots have been proposed for several important applications, such as rescue activities in disaster areas and outer space activities. The seminal model proposes a distributed system of k robots that have low capacities: they are identical

This research was partially supported by JSPS KAKENHI Grant Number JP19H04082, Comunidad de Madrid project BLOQUES-CM (S2018/TCS-4339) co-funded by EIE Funds of the European Union, and MINECO project *TRACES* (TIN2015-67522-C3-3-R).

© Springer Nature Switzerland AG 2019
M. Ghaffari et al. (Eds.): SSS 2019, LNCS 11914, pp. 111–126, 2019.
https://doi.org/10.1007/978-3-030-34992-9_10

(they are indistinguishable and all execute the same algorithm), oblivious (they have no memory of their past actions), and disoriented (they share no common orientation). Moreover, the robots do not communicate by sending or receiving messages, but have the ability to sense their environment and see the relative positions of the other robots.

Various models and algorithms [20,25] have been proposed to solve particular problems for autonomous mobile robots. This paper focuses on ring discrete models [4,5,10], in which robots perform their activities in a ring-shaped network. What and how problems can be solved by a group of autonomous mobile robots on ring-shaped networks is an important topic in the area, as shown by the large number of algorithms that have been proposed: e.g. the papers [4,12–14,18,27] propose algorithms for ring exploration, robot gathering on rings is solved in [5,9,11,24,26,30], and some other problems are solved in [10,19]. It is possible to make virtual rings over arbitrary-shaped network topologies and then mobile ring robot algorithms can be essentially applied to such topologies. Therefore, mobile ring robot algorithms are generic and worth investigating.

In the literature, the correctness of such algorithms relies on handmade mathematical proofs, which are error-prone. The untrustfulness of handmade mathematical proofs has been pointed out in [1,3,15,16]. Formal, automatic techniques could help us increase the confidence of the existing algorithms/proofs, as shown in [1,3,8,15,16]. For discrete models, model-checking has been proven useful to find errors in the proposed algorithms [3,15,16]. However, ring discrete models are not well supported by any existing specification language, such as DVE [2], SPIN [22], and Maude [7]. This is because of the particular symmetries owned by *rings*. Consequently, the specifiers, such as Berard *et al.* in [3] and Doan *et al.* in [15,16], need to specify rings by adapting other defined structures, such as *sequences*. It, therefore, makes the specification task tedious as well as time-consuming, while the specifications obtained are complicated and lengthy.

Context. Because rings cannot be directly supported by any existing specification language, we defined rings as associative sequences that satisfy two properties: rotative and reversible. We used Maude [7] as specification language because it allows us to use associative sequences. Now, the Maude Ring Specification Environment (Maude RSE), which explicitly supports ring-shaped networks, has been implemented on top of Maude. One key behind the tool is pattern matching between ring patterns and ring instances, called "ring pattern matching." Because of the above-mentioned reason, however, we encode ring patterns as sets of sequence patterns and simulate ring pattern matching by standard pattern matching between sets of sequence patterns and sequence instances, which is proven correct and transparent to Maude RSE users.

Contributions. Maude RSE itself and its theoretical foundations are the main achievements. Our research illustrates the power of rewriting logic in that Maude RSE can be implemented by extending Maude, more precisely Full Maude. That is, we do not need to implement such formal tools from scratch but we can do so by extending Maude and/or new formal tools on top of Maude. The case studies conducted in Maude RSE demonstrate that, because Maude RSE supports ring

structures, mobile ring robot algorithm specifications in Maude RSE are more concise and compact than those in Maude, while the time overhead incurred by handling rings is almost irrelevant. From a theoretical point of view, we prove that ring pattern matching can be simulated by pattern matching between sets of sequence patterns and sequence instances. Therefore, Maude RSE will benefit researchers in both the formal methods community and the distributed computing community.

Outline. Section 2 overviews mobile robots on ring architectures and the problems of specifying mobile ring robot algorithms. Section 3 introduces Maude RSE and outlines the theory of ring-pattern matching. It, then, presents how to specify mobile ring robot algorithms in Maude RSE. Section 4 evaluates Maude RSE. Finally, Sect. 5 concludes the paper. The source code of the tool and three case studies, the detailed descriptions of the ring pattern match theory, and more information in Sect. 3 are available at [29].

2 Problems

In this paper, we restrict our attention to discrete models, and more specifically to the ring topology. About timing assumption, we consider the more general asynchronous model ASYNC. In addition, we take into account multiplicities, which make much harder to formalize mobile robot algorithms. Multiplicities appear in robot algorithms when more than one robot is allowed in one node; in the following, we will call *multiplicities* to these nodes.

Robots follow a three-phase behavior: *Look*, *Compute*, and *Move*. During its Look phase a robot takes a snapshot of other robots' positions. The collected information is used in the Compute phase during which the robot decides whether to move or stay idle. There may be lag between the Compute phase and the subsequent Move phase and then some other movements by other robots may be done in-between. A move that has been decided by a robot in a Compute phase but has not yet been conducted by the robot in the subsequent Compute phase is called a pending move. In the Move phase, the robot may move to one of the two adjacent nodes, as computed in the previous phase. Rings are *anonymous*, that is, there are neither node nor edge labeled.

Anonymous rings have *rotative* and *reversible* characteristics, which cannot be directly handled by any existing specification language. Let us illustrate these problems with a simple example. Assume that we specify the ring (the system state on a ring) shown in Fig. 1(a), in which robots are *disoriented*. Such a system state can be expressed as a sequence $q_0\ q_1\ \ldots\ q_{j-1}\ q_j$ of intervals, where an interval q_i is the number of consecutive empty nodes between two non-empty nodes, in a view starting from any robot and traversing the ring in one arbitrary direction. System state representations are called *configurations*.

The system state as shown in Fig. 1(a) could be expressed as 2 1 0 3 1 in the (clockwise) view starting from the one at the bottom. Because it is a ring, the state could be also expressed, starting from other robots, as 1 0 3 1 2, 0 3 1 2 1, 3 1 2 1 0, and 1 2 1 0 3. Since robots are disoriented, the state

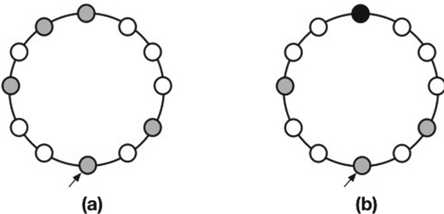

Fig. 1. (a) A system with two adjacent robots and (b) The system after the movement.

could be expressed as 1 3 0 1 2, 2 1 3 0 1, 1 2 1 3 0, 0 1 2 1 3, and 3 0 1 2 1 by reversing (i.e., considering counterclockwise) these sequences. All these configurations should be considered the same. Generally, given a sequence q_0 q_1 ... q_{j-1} q_j, the state it expresses is equivalent, in a ring, to all sequences obtained by rotating it — q_0 q_1 ... q_{j-1} q_j, q_1 ... q_{j-1} q_j q_0, ..., q_j q_0, q_1 ... q_{j-1} — and by reversing them — q_j q_{j-1} ... q_1 q_0, q_{j-1} ... q_1 q_0 q_j, ..., q_0 q_j q_{j-1} ... q_1. Unfortunately, it is impossible to directly specify this in any existing specification language. Actually, all the expressions above are considered totally different from any existing specification language point of view, so specifiers are required to implement their own strategies to handle them. Consequently, specifiers need to specify rings by adapting other defined structures, such as *sequences*. For instance, in [16], Doan et al. use associative operators in Maude.

To illustrate the idea used in [16], let us show how to specify a mobile ring robot algorithm in Maude. Given a ring on which there are two robots located at two adjacent nodes, respectively (such two robots are called adjacent robots), we want to put them together (i.e., create a multiplicity) by moving one of them to the node at which the other is located, where there is a non-empty node closer to the node at which the former is located than to the other node. For example, in Fig. 1(a) (where we define nodes with respect to the bottom node, with an arrow) we have two adjacent robots on the top, the one on the left (the fifth, clockwise, from the bottom) is separated from the rest of nodes by one empty node, while the one on the right is separated by three empty nodes. Hence, we would move the one on the left to the node at which the other one is located, as shown Fig. 1(b), where the black node indicates a multiplicity. Assuming we use `-1` to denote multiplicities, we can use a rewrite rule to specify this transition. The source state would use (i) `0` to indicate that two robots are adjacent, (ii) variables `I1` and `I2` to denote the intervals next to the adjacent robots, and (iii) a variable `S` to denote the remaining sequences. Assuming `I2` is larger than `I1`, we will increment the smaller interval (`I1`) and replace `0` (robots are adjacent) by `-1` (two robots are in the same node):

```
crl 0 I2 S I1 => -1 I2 S (I1 + 1) if I2 > I1.
```

In the particular case depicted in Fig. 1(a) the state could be expressed, clockwise and starting from the fifth node clockwise from the bottom, as 0 3 1 2 1. This

configuration matches (the left-hand side of) the rule[1] by substituting I2 with 3, I1 with 1, and S with 1 2. The state is rewritten to −1 3 1 2 2, which expresses the configuration in Fig. 1(b).

However, the state shown in Fig. 1(a) could be also expressed, clockwise from the top, as 3 1 2 1 0. In this case, there is no substitution such that the sequence can match the rule. For this reason we need another rule to handle it:

```
crl I2 S I1 0 => -1 I2 S (I1 + 1) if I2 > I1.
```

The configuration 3 1 2 1 0 matches this rule by substituting I2 with 3, I1 with 1, and S with 1 2.

Splitting Problem. The state in Fig. 1(a) could be also expressed, clockwise from the bottom, as 2 1 0 3 1, but it is impossible to apply any of the rules above to this configuration. The rest of the sequence is split into two sub-sequences at both sides of the whole sequence. Thus, it is necessary to split the variable S into two variables S1 and S2 that denote the remaining sequences at the left side and the right side, respectively.

```
crl S2 I1 0 I2 S1 => -1 I2 S1 S2 (I1 + 1) if I2 > I1.
```

In our theoretical framework we need to formally define and work on splitting and joining (which puts together two sub-sequences that substitute two sequence variables obtained from the splitting before) functions that deal with these cases.

Reversing Problem. Let us take a look at the state in Fig. 1(a), which could be expressed, counter-clockwise from the fifth node (clockwise) from the bottom, as 1 2 1 3 0. We need the following rule for this case:

```
crl I1 S I2 0 => -1 I2 rev(S) (I1 + 1) if I2 > I1.
```

When the configuration matches the rule, what substitutes S is 2 1. We need to reverse 2 1, the sequence that substitutes S because otherwise what is obtained by applying the rule to the configuration is −1 3 2 1 2, which is different from −1 3 1 2 2. The function rev reverses a sequence, e.g rev(2 1) is 1 2. The configuration 1 2 1 3 0 matches this rule by substituting I2 with 3, I1 with 1, and S with 2 1. The state is rewritten to −1 3 1 2 2, the configuration in Fig. 1(b).

Hence, we need to have all the rules by rotating and reversing the left-hand side of the first rule to handle all possible sequences. We need 10 rules to specify the above-mentioned transition. Note that we name the rules RL1 to RL10.

```
crl[RL1]  0 I2 S I1     => -1 I2 S (I1 + 1)            if I2 > I1.
crl[RL2]  I2 S I1 0     => -1 I2 S (I1 + 1)            if I2 > I1.
crl[RL3]  S I1 0 I2     => -1 I2 S (I1 + 1)            if I2 > I1.
crl[RL4]  I1 0 I2 S     => -1 I2 S (I1 + 1)            if I2 > I1.
crl[RL5]  S2 I1 0 I2 S1 => -1 I2 S1 S2 (I1 + 1)        if I2 > I1.
```

[1] In the actual specification, we need an operator enclosing the sequence, such as {_} to avoid rewriting sub-sequences. However, to make the explanation as close as possible to mathematical description, we omit it here.

```
crl[RL6]  I1 S I2 0     => -1 I2 rev(S) (I1 + 1)          if I2 > I1.
crl[RL7]  0 I1 S I2     => -1 I2 rev(S) (I1 + 1)          if I2 > I1.
crl[RL8]  I2 0 I1 S     => -1 I2 rev(S) (I1 + 1)          if I2 > I1.
crl[RL9]  S I2 0 I1     => -1 I2 rev(S) (I1 + 1)          if I2 > I1.
crl[RL10] S1 I2 0 I1 S2 => -1 I2 rev(S1) rev(S2) (I1 + 1) if I2 > I1.
```

This makes the specification complicated and lengthy and specifiers exhausted. If a ring is not faithfully specified, the formal verification of a mobile ring robot algorithm may overlook cases.

3 Maude Ring Specification Environment (Maude RSE)

One possible way to solve these problems is to develop a specification environment in which rings are explicitly supported. It is reasonable, and saves time and effort, if the environment is built on top of an existing specification system. For this reason, Maude Ring Specification Environment (Maude RSE) is implemented on top of Maude, a rewriting logic-based programming and specification language, taking advantage of its meta-programming features. This section outlines a theory of pattern matching on rings ("ring-pattern matching") that guarantees that our way of dealing with ring-pattern matching makes sense and briefly describes how Maude RSE is built, its architecture, and how to define a ring topology in it.

3.1 Ring Pattern Match Theory

Sequences. Let sequence patterns be in the form $ES_1\ ES_2\ \ldots\ ES_n$, where each ES_i is an element, an element variable, or a sequence variable. We suppose that the juxtaposition operator used as the constructor in sequence patterns is associative and the empty sequence, denoted ϵ, is its identity. Sequence instances are sequence patterns that do not contain variables. Let **SP** and **Seq** be the sets of sequence patterns and sequence instances, respectively. Let **Elt** be the set of (concrete) elements, **EV** be the set of element variables, and **SV** be the set of sequence variables.

Definition 1 (Sequence pattern match). *Pattern match between* $sp \in \mathbf{SP}$ *&* $seq \in \mathbf{Seq}$ *is to find all substitutions* σ *such that* $\sigma(sp) = seq$. *Let* $sp =?= seq$ *be the set of all such substitutions.*

Definition 2 (Split sequence patterns). *For* $sp \in \mathbf{SP}$, $\mathrm{split}(sp)$ *is a sequence pattern such that each sequence variable* S *in* sp *is replaced with* $\mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)$. *Then, the inductive definition of split is* $\mathrm{split}(\varepsilon) = \varepsilon$, $\mathrm{split}(e) = e$ *for* $e \in \mathbf{Elt}$, $\mathrm{split}(E) = E$ *for* $E \in \mathbf{EV}$, $\mathrm{split}(S) = \mathrm{sv}(S, 0)\ \mathrm{sv}(S, 1)$ *for* $S \in \mathbf{SV}$ *and* $\mathrm{split}(SP_1\ SP_2) = \mathrm{split}(SP_1)\ \mathrm{split}(SP_2)$ *for* $SP_1, SP_2 \in \mathbf{SP}$.

Definition 3 (Joining split sequence variables). *For* $sp \in \mathbf{SP}$ *and* $seq \in \mathbf{Seq}$, *let* σ *be in* $(\mathrm{split}(sp) =?= seq)$. $\mathrm{join}(\sigma)$ *is the substitution* σ' *such that for each sequence variable* S *in* sp $\sigma'(S) = \sigma(\mathrm{sv}(S, 0))\ \sigma(\mathrm{sv}(S, 1))$ *and for any*

other variables X $\sigma'(X) = \sigma(X)$*. The domain of* join *can be naturally extended to the set of substitutions such that* $\mathrm{join}(\mathrm{split}(sp) =?= seq)$ *is* $\{\mathrm{join}(\sigma) \mid \sigma \in (\mathrm{split}(sp) =?= seq)\}$*.*

Rings

Definition 4 (Rings). *For* $sp \in \mathbf{SP}$*,* $[sp]$ *is called a ring pattern and satisfies (1) the rotative law (*$[sp] = [\mathrm{rtt}(sp)]$*) and (2) the reversible law (*$[sp] = [\mathrm{rev}(sp)]$*). When* sp *is a sequence* $seq \in \mathbf{Seq}$*,* $[seq]$ *is called a ring.* $\mathrm{rtt}(sp)$ *rotates* sp *rightward;* $\mathrm{rev}(sp)$ *reverses* sp*.*

Definition 5 (Ring pattern match). *For* $sp \in \mathbf{SP}$ *and* $seq \in \mathbf{Seq}$*, pattern match between* $[sp]$ *and* $[seq]$ *is to find all substitutions* σ *such that* $[\sigma(sp)] = [seq]$*. Let* $[sp] =?= [seq]$ *be the set of all such substitutions.*

Definition 6 (Sequences rotated and/or reversed). *For* $sp \in \mathbf{SP}$*,* $[[sp]]$ *is the set of sequences inductively defined as follows: (1)* $sp \in [[sp]]$ *and (2) if* $sp' \in [[sp]]$*, then* $\mathrm{rtt}(sp') \in [[sp]]$ *and* $\mathrm{rev}(sp') \in [[sp]]$*.*

The intuitive idea is that $[sp]$ is an implicit notation indicating that sp behaves as a ring while $[[sp]]$ is a explicit notation that lists all possible combinations after applying rotation and reverse to sp. In this way, given a particular sequence seq it is possible to use standard pattern matching between the elements in $[[sp]]$ and seq, so $([[sp]] =?= seq) = \{\sigma \mid \sigma = (sp' =?= seq), \text{for } sp' \in [[sp]]\}$. The following theorem shows that it is possible to use $[[sp]]$ as an effective implementation of $[sp]$ (see Ring pattern match theory in [29] for details):

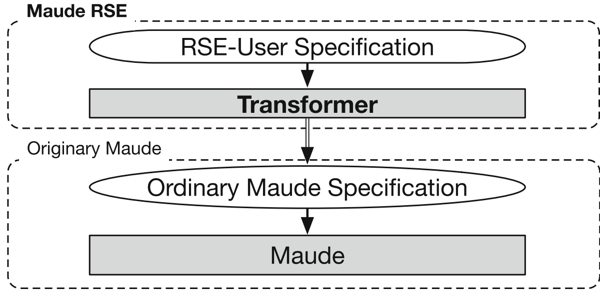

Fig. 2. Architecture of Maude RSE.

Theorem 1. *For any sequence pattern* $sp \in \mathbf{SP}$ *and any sequence* $seq \in \mathbf{Seq}$*,* $\mathrm{join}([[\mathrm{split}(sp)]] =?= seq) = ([sp] =?= [seq])$*.*

3.2 Extending Maude with Ring Attributes

It has been demonstrated in [16,17,21,28] that Maude allows programmers to specify distributed algorithms/systems more succinctly than others programming languages. In particular, we extend Full-Maude [7], which is an extension of Maude written in Maude itself that provides extra features to extend Maude. The specification environment is built as depicted in Fig. 2. A specification in Maude RSE is considered as a user specification, which may contain specifications of a ring topology that would not be supported by the standard Maude engine. The main player in the system is *Transformer* that takes a user specification and transforms it into an ordinary Maude specification. Technically, a user specification is represented as a term at the meta-level. Transformer, then, analyzes and modifies it by adding extra equations/rules that handle rings. Pattern matching is a key functionality in Maude. Because pattern matching between ring patterns and ring instances is not supported by Maude and any other existing specification languages, we need to simulate it. There are two possible ways to simulate ring pattern matching. For a sequence pattern SP and a sequence instance SI, (1) we generate all sequence instances that denote the ring instance denoted by SI and model check each sequence instance with SP, and (2) we generate all sequence patterns that denote the ring pattern denoted by SP and model check SI with each ring pattern. We have adopted (2) because Maude automatically matches one sequence instance with many sequence patterns, while (1) would force us to manually handle a collection of sequence instances. It is non-trivial, however, to decide whether a matching between a ring pattern and a ring instance can be simulated by pattern matching between a collection of sequence patterns and a sequence instance. We have formally proved that the former can be simulated by the latter, see Sect. 3.1. The main idea is that given a user ring specification as a ring pattern, Maude RSE generates all corresponding sequence patterns to deal with the "ring" characteristic. Intuitively, given a ring pattern $[ES_1 \ldots ES_i \ldots ES_n]$, Transformer generates as the left-hand side of a rule: n rotative patterns $[ES_1 \ldots ES_i \ldots ES_n]$, ..., $[ES_i \ldots ES_n\ ES_1]$, ..., $[ES_n\ ES_1 \ldots ES_i]$ and n reversible patterns $[ES_1\ ES_n \ldots ES_i]$, ..., $[ES_i \ldots ES_1 \ldots ES_n]$, ..., $[ES_n \ldots ES_i \ldots ES_1]$. When ES_i (i = 1, 2, ..., n) is a variable, it is split and jointed afterwards. We can basically use the right-hand side of the given rule as the right-hand side for the other $2n-1$ patterns generated as the left-hand side. We, however, need to reverse sequences that substitute sequence variables occurring in the right-hand side for the n reversible patterns. For example, in the problem in Sect. 2, users only need to specify the first rule $RL1$ while all other rules are automatically generated by Transformer by **1. Splitting:** All sequence variables are splitted. e.g. variable S is splitted into $S1$ and $S2$.

```
crl 0 I2 S1 S2 I1 => -1 I2 S1 S2 (I1 + 1)  if I2 > I1.
```

2. Rotating and **3. Joining:** All elements in the sequence of the left-hand side are rotated. After that, some pairs of sequence variables that are splitted from one sequence variable and appeared in the splitted order are joined. We get the rules `RL2` to `RL5`. **4. Reversing:** All sequences on the left-hand sides are reversed

(rules RL6 to RL10). As the result, we get all 10 rules. Users do not need to deal with the "ring" characteristic, which is handled transparently by Maude RSE.

In fact, Transformer needs to handle more complicated user specification rules/equations, all of them guarateed correct by Theorem 1. Because the result of the transformation is a standard Maude specification, we can guarantee that all Maude facilities, such as the LTL model checker, can be directly used.

3.3 Syntax Declaration

We consider two kinds of rings: *oriented* rings in which the orientation of the ring (clockwise and anti-clockwise order) is taken into account, and *disoriented* rings in which there is no orientation. In Maude, types are called *sorts*. A sort denotes the set of elements of the same type. For example, the sort `Nat` denotes 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 denoted by the latter, and the latter is called a supersort of the former. Keywords `sort` and `subsort` are used to declare sort and subsort relation, respectively. Elements of a given sort are built by constructors, with keyword `op`, together with the keyword `ctor`, given the arity and the coarity. Moreover, operators can have equational axioms, such as associativity (`assoc`) and identity (`id:`).

We first consider *disoriented* rings, implemented by the `ring` attribute. In particular, rings are constructed as a sequence of elements with this attribute. Let us assume `Elem` and `Seq` the sorts for elements and sequences, respectively. The configurations of a system as rings could be defined as:

```
subsort Elem < Seq.
op emp : -> Seq [ctor].
op __ : Seq Seq -> Seq [ctor assoc id: emp].
op [_] : Seq -> Config [ring ctor].
```

An operator without any argument is called a *constant*, such as `emp`, which stands for the empty sequence. Underscores are placeholders where arguments are placed. Similarly, `id: emp` indicates that operator `emp` is the identity element of the juxtaposition (empty syntax) operator `__`. `Seq` is a supersort of `Elem`, which means that each `Elem` is treated as the singleton sequence only consisting of this element. The operator `__` is used to construct sequences of elements: for s_1 and s_2 of sort *Seq*, s_1 s_2 has sort *Seq*. The structure `__` is presented just as an example; it could be replaced by any other structure that depends on the user's preferences, such as `_,_` and `_|_`. Likewise, the structure `[_]` is an optional preference. A configuration is defined as a ring structure that is specified based on a sequence of elements. Because a ring is disoriented, the mirror image of a ring represents the same state as the original state. When we use intervals as ring elements, we could use `Int` for `Elem`, where `Int` is the sort for integers. The system shown in Fig. 1(a) is expressed using this syntax as [0 3 1 2 1], [3 1 2 1 0], [1 2 1 0 3], [1 2 1 3 0], and so on.

For *oriented* rings, Maude RSE provides the `r-ring` attribute that could be considered as a sub-class attribute of the `ring` attribute. The *oriented* ring and its mirror image do not necessarily represent the same state.

3.4 Applications

We have specified and model checked three algorithms for exploration with stop, exploration, and gathering. The two last algorithms have been also specified in standard Maude. We compare our new specifications with existing ones (see Sect. 4). Due to page limitation, this section only presents how to formalize and specify the algorithm for exploration with stop in Maude RSE.

Robots Exploration with Stop on Ring Under ASYNC. The ASYNC (or asynchronous) model is considered. Each robot can distinguish whether a node is empty, occupied by one robot, or occupied by more than one robot. The problem of exploring with stop requires that, regardless of the initial placement of the robots, each node must be visited by at least one robot and the robots must be in a configuration in which they all remain idle.

Exploration Algorithm [18]**.** The algorithm works following a sequence of three distinct phases: Set-Up, Tower-Creation, (towers are the equivalent notion to multiplicities in our notation) and Exploration. The Set-Up phase transforms any initial configuration into one that is in a predetermined set of configurations (called *no-towers-final*) with special properties. After that, the Tower-Creation phase and then the Exploration phase are executed. Finally, all nodes are visited and no robot will make any further moves.

Formal Specification of Exploration Algorithm. Let us suppose that there are n nodes denoted u_0 u_1 ... u_{n-1} and each node may be occupied by more than one robot. The multiplicity of robots located at node u_i is denoted d_i: $d_i = 0$, $d_i = 1$, and $d_i = 2$ indicate that there are no robots, there is exactly one robot, and there are more than one robot, respectively.

State Expressions. So far the state of a system on a ring has been represented as a sequence of elements, e.g, for the system in Fig. 1(a), elements are intervals. For this system, each element is a node of the ring. Remember from Sect. 2 that robots move in two phases: first they decide where to move and then the movement is performed; when the movement has been decided but not applied yet it is called *pending move*. A pending move is represented as a snapshot of the ring from the node the robot will move; to avoid ambiguities due to the symmetries of the ring, the same snapshot is stored in the target node.

Once the pending move is completed a new movement can be computed, so we have at most one pending move at a time (this is true because of the particular structure of the algorithm given above, that first creates multiplicities and does not move them afterwards); on the other hand, many target-pending moves can be stored in each node, because robots from different nodes may want to move

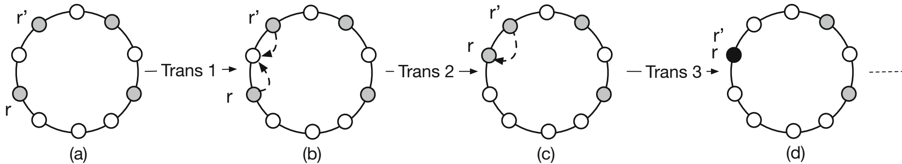

Fig. 3. One possible transitions from the initial configuration: a dashed arrow represents a pending move and a black node represents a tower.

to the same node. because robots from different nodes may want to move to the same node. In this way, executing a movement is as simple as finding two nodes, one with the pending move and another one with the same pending move as a target. Note that, in this stage, robots just know that they have to move, but they cannot access other robots' pending moves.

Hence, we denote a node as a tuple $\langle d_i, p_i, ps_i \rangle$, where d_i denotes the multiplicity of the node, p_i denotes a pending move, and ps_i denotes the set of pending moves that will be done by other robots. p_i is either one pending move or `nil` meaning that there are no pending moves. Given this definition of node, a snapshot (i.e., the pending move) for a robot at node u_i is the sequence d_i d_{i+1} ... d_{i-1} of the multiplicities taken from that node. ps_i is either a set of pending moves or `emp`, the empty set. The sort `Pending` denotes pending moves, `PendingSet` pending move sets, and `Node` nodes. A configuration, of sort `Config`, is expressed as a ring of nodes with the `ring` attribute:

```
subsort Node < Seq.
op <_,_,_> : Nat Pending PendingSet -> Node [ctor].
op emp : -> Seq [ctor].
op __ : Seq Seq -> Seq [ctor assoc id: emp].
op [_] : Seq -> Config [ctor ring].
```

The structure `<_,_,_>` is used to construct nodes. For $d \in Nat$, $p \in Pending$, $ps \in PendingSet$, we have $\langle d, p, ps \rangle \in Node$. A configuration is defined as a ring `[_]`, which takes as argument a sequence of nodes. For example, let v and v' be the pending moves 1 0 1 0 1 0 1 0 0 0 and 1 0 1 0 1 0 0 0 1 0 for r and r', respectively. Note that each snapshot is taken from the node making the movement in clockwise order, although the anti-clockwise order would be valid as well. The configuration of the system with two pending moves as shown in Fig. 3(b) could be expressed from robot r' clockwise as $[\langle 1, v', emp \rangle \langle 0, nil, emp \rangle \langle 1, nil, emp \rangle \langle 0, nil, emp \rangle\ \langle 1, nil, emp \rangle \langle 0, nil, emp \rangle \langle 0, nil, emp \rangle \langle 0, nil, emp \rangle \langle 1, v, emp \rangle \langle 0, nil, (v; v') \rangle]$. We see that two nodes (standing for r and r', respectively) have pending moves in the second component of the tuple while the target node has both pending moves in the third component.

State Transitions. 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. Such changes, called (state) transitions, are

specified by rewrite rules. For example, one possible transition path is as shown in Fig. 3. The following rewrite rule describes the action when a robot performs its pending move (note that variable P appears twice, once as pending move and then as part of the set in the target node; it disappears once the rule is applied):

```
rl [S1 < 1, P, PS > < D, P', (P; PS') > S2] =>
   [S1 < 0, nil, PS > < D + 1, P', PS' > S2].
```

where S1 and S2 are variables of sort Seq, P and P' are variables of sort Pending, PS and PS' are variables of sort PendingSet, and D is a variable of sort Nat.

The configuration [S1 ⟨ 1, P, PS ⟩⟨ D, P', (P; PS') ⟩ S2] represents any state such that the robot ⟨ 1, P, PL ⟩ has a pending move P and the next node is ⟨ D, P', (P; PL') ⟩ in which P is in the set of pending moves. In addition to these two nodes, such a state may have some more nodes that are expressed as $S1$ and $S2$. The term $[\langle 1, v', emp\ \rangle\langle 0, nil, emp\ \rangle\langle 1, nil, emp\rangle\langle\ 0, nil, emp\ \rangle\langle 1, nil, emp\rangle\langle 0, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 1, v, emp\rangle\langle 0, nil, (v; v')\rangle]$ expresses the state of Fig. 3(b). The left-hand side of the above rewrite rule matches this term by using the substitution $S1 \mapsto \langle 1, v', emp\rangle\langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 0, nil, emp\rangle$, $P \mapsto v$, $PS \mapsto emp$, $D \mapsto 0$, $P' \mapsto nil$, $PS' \mapsto v'$ and $S2 \mapsto emp$, and the rewrite rule can be applied to the term, creating the state $[\langle 1, v', emp\rangle\langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 1, nil, emp\rangle\ \langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 0, nil, emp\rangle\langle 1, nil, v'\ \rangle]$, which corresponds to the state in Fig. 3(c). Note that the size of the ring is not fixed.

Model Checking the Algorithm. We use the Maude LTL model checker to verify that the algorithm enjoys desired properties. The authors in [18] give some important theorems, such as *Theorems 3.1* and *3.2*, that must hold to guarantee the correctness of the algorithm. For example, *Theorem 3.1* states a property that must be satisfied at the end of the Set-Up phase: any initial configuration is transformed into a *no-towers-final* configuration. We have formally expressed these theorems as LTL formulas [23]. For example, *Theorem 3.1* then is expressed as the LTL formula:

```
theorem3-1 = [] (endOf -> SetUp) /\ <> endOf.
```

where [] is the always operator and <> is the eventually operator. The proposition endOf holds if and only if the Set-Up phase has finished and the proposition SetUp holds if and only if the state does not have any towers and all robots are located adjacent to each other in one or two groups.

As the result of the model checking, no counterexamples are found for the LTL formula. This makes us more confident on the correctness of the algorithm.

4 Evaluation

We compare the sizes and performance of specifications in plain Maude [15,16] and in Maude RSE and report on the overheads (which is almost nothing) introduced by Maude RSE for model checking. We consider two algorithms solving

two main problems on ring: the perpetual exploration algorithm, which was defined in [4] and specified in [15], and the gathering algorithm, designed in [9] and specified in [16]. Note that Maude RSE successfully reproduces the model checking experiments reported in [15,16], finding the counterexamples demonstrating that the algorithms do not enjoy some properties.

4.1 A Perpetual Exploration Algorithm [4]

In [15], the ring is represented as the set of all non-empty nodes. The ring features, namely rotation and reversibility, are dealt with by using associative and commutative sets. However, the commutative attribute makes it impossible to keep the order in the ring. Specifiers, thus, are forced to use complex constraints to specify the algorithms because the order of elements might change. For this reason, several functions are defined to handle these constrains.

By using Maude RSE, we do not need to handle ring characteristics and we do not need to use commutativity as the attribute of the constructor used to construct configurations. Our specification gets rid of all these extra functions. In total, we reduce over 50% of the code.

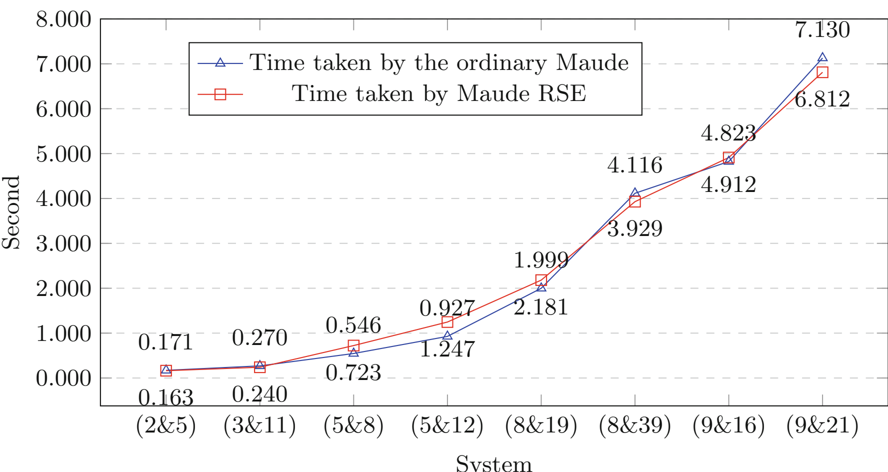

Fig. 4. Maude RSE preserves the performance of the ordinary Maude environment.

4.2 A Gathering Algorithm [9]

In [16], the authors use 44 rules to specify the system and 53 equations to handle some constraints about configurations. Many of these rules and equations are defined to handle rings. In Maude RSE, the specification requires 17 rules and 18 equations, that is, we obtain a code reduction of more than 60%.

Performance Analysis. We conducted model checking experiments for the gathering algorithm to compare the performances. 8 different systems in terms of the number of robots and the size of the ring, e.g. System 1 with 2 robots and 5 nodes denoted as (2&5), are taken. Experiments were conducted on a 4 GHz Intel Core i7 processor with 32 GB of RAM. The results are shown in Fig. 4. Based on these experiments, we can conclude that Maude RSE preserves the performance of the ordinary Maude environment; no extra time consuming.

5 Conclusion

Because mobile robot systems are a new form of distributed system, the existing specification methods (and tools) do not support these systems appropriately. In this case, a new language or an extension of an existing language is needed. This paper introduces an extension of Maude to mobile ring robot algorithms: Maude RSE. Maude RSE makes it possible to specify ring structures, which need to be used to specify mobile ring robot algorithms. As extensions of ring topology, recent research on rings consider dynamic rings where edges may appear and disappear unpredictably [6]. Furthermore, more kinds of robots, such as myopic luminous robots [31] are proposed to work on rings. As future work, we try to tackle other kinds of robots on rings, such as myopic luminous robots. We then consider extending Maude RSE in the following directions: (1) to support other features on ring, such that rings are dynamic, and (2) to support other topologies by making virtual rings over them.

References

1. Balabonski, T., Delga, A., Rieg, L., Tixeuil, S., Urbain, X.: Synchronous gathering without multiplicity detection: a certified algorithm. In: Bonakdarpour, B., Petit, F. (eds.) SSS 2016. LNCS, vol. 10083, pp. 7–19. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49259-9_2
2. Barnat, J., Brim, L., Češka, M., Ročkai, P.: Divine: parallel distributed model checker. In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology. IEEE (2010)
3. Bérard, B., Lafourcade, P., Millet, L., Potop-Butucaru, M., Thierry-Mieg, Y., Tixeuil, S.: Formal verification of mobile robot protocols. Distrib. Comput. **29**(6), 459–487 (2016)
4. Blin, L., Milani, A., Potop-Butucaru, M., Tixeuil, S.: Exclusive perpetual ring exploration without chirality. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 312–327. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15763-9_29
5. Bonnet, F., Potop-Butucaru, M., Tixeuil, S.: Asynchronous gathering in rings with 4 robots. In: Mitton, N., Loscri, V., Mouradian, A. (eds.) ADHOC-NOW 2016. LNCS, vol. 9724, pp. 311–324. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40509-4_22
6. Bournat, M., Dubois, S., Petit, F.: Computability of perpetual exploration in highly dynamic rings. In: IEEE 37th International Conference on Distributed Computing Systems, pp. 794–804 (2017)

7. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
8. Courtieu, P., Rieg, L., Tixeuil, S., Urbain, X.: Certified universal gathering in $\mathbb{R}^2$ for oblivious mobile robots. In: Gavoille, C., Ilcinkas, D. (eds.) DISC 2016. LNCS, vol. 9888, pp. 187–200. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53426-7_14
9. D'Angelo, G., Di Stefano, G., Navarra, A.: Gathering on rings under the look-compute-move model. Distrib. Comput. **27**, 255–285 (2014)
10. D'Angelo, G., Di Stefano, G., Navarra, A., Nisse, N., Suchan, K.: Computing on rings by oblivious robots: a unified approach for different tasks. Algorithmica **72**(4), 1055–1096 (2015)
11. D'Angelo, G., Navarra, A., Nisse, N.: A unified approach for gathering and exclusive searching on rings under weak assumptions. Distrib. Comput. **30**(1), 17–48 (2017)
12. Datta, A.K., Lamani, A., Larmore, L.L., Petit, F.: Enabling ring exploration with myopic oblivious robots. In: IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, pp. 490–499 (2015)
13. Devismes, S.: Optimal exploration of small rings. In: Proceedings of the Third International Workshop on Reliability, Availability, and Security, pp. 91–96 (2010)
14. Devismes, S., Petit, F., Tixeuil, S.: Optimal probabilistic ring exploration by semi-synchronous oblivious robots. Theor. Comput. Sci. **498**, 10–27 (2013). https://doi.org/10.1016/j.tcs.2013.05.031
15. Doan, H.T.T., Bonnet, F., Ogata, K.: Model checking of a mobile robots perpetual exploration algorithm. In: Liu, S., Duan, Z., Tian, C., Nagoya, F. (eds.) SOFL+MSVL 2016. LNCS, vol. 10189, pp. 201–219. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57708-1_12
16. Doan, H.T.T., Bonnet, F., Ogata, K.: Model checking of robot gathering. In: Proceedings of The 21th Conference on Principles of Distributed Systems, pp. 12:1–12:16 (2017)
17. Doan, H.T.T., Bonnet, F., Ogata, K.: 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, pp. 1586–1596 (2017)
18. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Computing without communicating: ring exploration by asynchronous oblivious robots. Algorithmica **65**(3), 562–583 (2013)
19. Flocchini, P., Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Multiple mobile agent rendezvous in a ring. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 599–608. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24698-5_62
20. Flocchini, P., Prencipe, G., Santoro, N.: Distributed Computing by Oblivious Mobile Robots. Morgan & Claypool Publishers (2012)
21. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's megastore in real-time maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_25
22. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2004)
23. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, Cambridge (2004)

24. Izumi, T., Izumi, T., Kamei, S., Ooshita, F.: Mobile robots gathering algorithm with local weak multiplicity in rings. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 101–113. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13284-1_9
25. Kawamura, A., Kobayashi, Y.: Fence patrolling by mobile agents with distinct speeds. Distrib. Comput. **28**(2), 147–154 (2015)
26. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theor. Comput. Sci. **390**(1), 27–39 (2008)
27. Lamani, A., Potop-Butucaru, M.G., Tixeuil, S.: Optimal deterministic ring exploration with oblivious asynchronous robots. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 183–196. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13284-1_15
28. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the walter transactional data store. In: Rusu, V. (ed.) WRLA 2018. LNCS, vol. 11152, pp. 136–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99840-4_8. https://sites.google.com/ site/siliunobi/walter
29. Our Maude source files. https://goo.gl/6AnwHE
30. Millet, L., Potop-Butucaru, M., Sznajder, N., Tixeuil, S.: On the synthesis of mobile robots algorithms: the case of ring gathering. In: Felber, P., Garg, V. (eds.) SSS 2014. LNCS, vol. 8756, pp. 237–251. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11764-5_17
31. Ooshita, F., Tixeuil, S.: Ring exploration with myopic luminous robots. In: Izumi, T., Kuznetsov, P. (eds.) SSS 2018. LNCS, vol. 11201, pp. 301–316. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03232-6_20