International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 13, Number 1, pages 18–47, January 2023

Dynamic Line Maintenance by Hybrid Programmable Matter[1]

Nooshin Nokhanji
School of Computer Science, Carleton University
Ottawa, Canada

Paola Flocchini
School of Electrical Engineering and Computer Science, University of Ottawa
Ottawa, Canada

Nicola Santoro
School of Computer Science, Carleton University
Ottawa, Canada

Received: July 25, 2022
Revised: October 20, 2022
Accepted: November 23, 2022
Communicated by Susumu Matsumae

Abstract

Motivated by the manipulation of nanoscale materials, recent investigations have focused on hybrid systems where passive elements incapable of movement, called tiles, are manipulated by one or more mobile entities, called robots, with limited computational capabilities. Like in most self-organizing systems, the fundamental concern is with the (geometric) shapes created by the position of the tiles; among them, the line is perhaps the most important. The existing investigations have focused on formation of the shape, but not on its reconfiguration following the failure of some of the tiles. In this paper, we study the problem of maintaining a line formation in presence of dynamic failures: any tile can stop functioning at any time. We show how this problem can be solved by a group of very simple robots, with the computational power of deterministic finite automata.

Keywords: Shape formation, fault-tolerance, hybrid programmable matter, line maintenance, distributed algorithms, self-assembly

[1]Some of these results have been announced in [25, 26]

1 Introduction

1.1 Framework and Background

The research goal of the interdisciplinary field of *programmable matter* is to formalize and investigate the design and manipulation of micro- and nano-scale materials. Programmable matter consists of a large collection of micro- or nano-sized elements (variously called particles, tiles, robots, etc.) which are limited in terms of computation, communication, and motorial abilities; they might have the ability to change their appearance and physical properties such as shape, density, or color based on autonomous sensing or user input [33]. Although individual entities do not require any form of central or external control or direction, they are collectively capable of performing global tasks such as autonomous monitoring and repair, minimal invasive surgeries, and smart materials. A wide range of theoretical models have been proposed, ranging from DNA self-assembly systems (e.g., [29, 30, 31]), to metamorphic robots (e.g., [5, 34]), including models inspired by natural phenomena like insects or micro-organisms (e.g., [14]).

The computational universe, or *system*, defined by these models varies greatly; however, with respect to how they define the nature and capabilities of the entities, there are basically two types of systems: *passive* and *active*.

In *passive* systems the individual entity has either no intelligence while it is able to move and bond based on its structural properties as well as the chemical interactions with the environment, or has limited computational capabilities but is not capable of controlling its locomotion. DNA computing [1, 4, 10], population protocols [2], molecular computing and tile self-assembly models [9, 15, 29], as well as slime molds [3, 23] are the well-known instances of *passive* systems.

On the other hand, *active* programmable matter systems are composed of a multitude of identical autonomous entities, usually called particles or robots, provided with simple computational capabilities (usually, finite-state machines), strictly local interaction and communication capabilities (only with neighboring particles), and limited motorial capabilities. Typically, the entities are located on a tassellation of the Euclidean space. Instances of this type of systems are modular robotic systems [22], metamorphic robots [5, 32], *Nubot* model [35], and the more recent *Amoebot* model [7, 11, 13].

Since active elements are presumably more difficult to build, these systems might be far more costly to realize than the systems based on passive elements [8]. This has led to the recent proposal of a model of *hybrid* programmable matter that combines *active* and *passive* entities [8, 19, 20]. Specifically, the system is composed of a (large) set of passive hexagonal tiles and a (small) set of active robots. Each tile occupies only one node of a finite triangular lattice and it is not capable of performing any computation, communication, nor movement. The robots are able to lift, carry, or place the tiles, and are responsible for managing the relative position of tiles so to achieve a desired task; each robot has the computational power of a finite automaton, and is able to communicate with neighbouring robots. The system is subject to a connectivity requirements of the tiles. On a node there can be at most one robot at any time. It should be pointed out that similar models, where a set of finite automata manipulates passive materials in a square grid, have been considered in [6, 17, 24] under connectivity preservation requirements, and in [16] without connectivity constraints.

1.2 Shapes, Concurrency, and Faults

Like in most self-organizing systems, the fundamental concern in the *Hybrid Programmable Matter* model is with the (geometric) shapes created by the position of the tiles. The existing investigations have focused on the *formation* of a given shape by the robots (e.g., [8, 20, 21]) or the *recognition* of the shape (e.g., [8, 19]). Among the basic shapes, the *line* is especially important, as it is utilized as a foundation for constructing more complicated shapes or to perform more complex tasks (e.g., [20, 21]).

In these investigations, two important computational aspects are somehow neglected.

The first aspect is with regards to *faults*. In fact, in the existing investigations, it is assumed that the system elements never fail [8, 19, 20, 21]. Removing this not realistic assumption brings to light some (theoretically and practically) important problems that need to be addressed and solved. This has been the case in some active models of programmable matter, where the possibility of

faults has been considered. The focus of those investigations has been on the basic problem of *line reconfiguration*: following the failure of some entities, all the non-faulty entities must self-organize in a line that does not contain faulty entities. This task has been examined in the *Amoebot* [12, 27] and *Metamorphic Robot* [28] models. These investigations have been however limited to the case when all the faults have occurred before the execution of the algorithm (the *static* case) [12, 27, 28]; the *dynamic* case, where faults can occur at any time, has not yet been examined in the literature for those models.

The other aspect, neglected in the existing investigations on hybrid programmable matter, is with regards to handling the *concurrency* problems (e.g., collisions, deadlock) arising by the interaction between the $r \geq 1$ autonomous robots in the system. Indeed, with a single exception, these problems are sidestepped by assuming that $r = 1$; i.e., there is only one robot and, hence, no concurrency [8, 19, 20, 21, 25]. The only exception is [20], where the case of more than one robots is also considered; however, in that paper the concurrency problems due to simultaneous interaction between entities are removed by assuming a *sequential* activation scheduler[2]: only one robot is active at any time. This situation must be contrasted with those of the active models of robots where much more complex levels of concurrent interaction are allowed, including the so-called *semi-synchronous* activation scheduler[3]: an arbitrary number of robots is activated at each round, and those active in a round operates concurrently and simultaneously.

1.3 Problem and Contributions

In this paper, we start the investigation of computing in the *Hybrid Programmable Matter* model in presence of concurrency of the robots and of dynamic failures of the tiles.

We study the Dynamic Line Maintenance (DLM) problem in presence of dynamic failures: any tile can stop functioning at any time. Robots can detect whether or not a tile present in the same node or in a neighbouring node, as well as a tile they are carrying, is faulty.

Let us stress that there is no constraint on the number of tiles that will become faulty, nor on the location and time of the occurrence of a fault; furthermore, more than one tile might become faulty at the same time.

The DLM problem requires the r robots to collaboratively remove any faulty tile from the line within finite time from it becoming faulty, subject to three constraints: (1) (*Line Maintenance*) all non-faulty tiles eventually are on the nodes of the same line segment; (2) (*Basic Connectivity*) at any time, in the sub-grid of the locations occupied by all robots and tiles, all non-faulty tiles are connected, and (3) (*Collisionless*) at no time two robots occupy the same tile. In particular, the *Basic Connectivity* restriction is an integral part of the *Hybrid Programmable Matter* model [8]; it arises from applications where the overall structure formed by the robots and the tiles floats in a liquid: it prevents the robot or parts of the tile structure from floating apart.

In this paper, we investigate the DLM problem when an arbitrary number r of robots cooperate to maintain the line of non-faulty tiles in spite of dynamic failures of tiles; that is, $1 \leq r \leq n$, where n is the number of tiles, and faults can occur at any time and place. We study and solve the problem under the difficult semi-synchronous activation scheduler: an arbitrary subset of the robots is activated and concurrently operates in the same round. Our algorithm requires the robots to have only minimal computational capabilities, those of finite state machines, and limited communication power, sufficient to interact with neighbouring robots. At the basis of our solution are two algorithms, designed for the special cases of $r = 1$ (i.e., when there is a single robot in the system) and $r = n$ (i.e., when all the initial tiles are occupied by a robot). We then devise a strategy that combines elements of the solutions for $r = 1$ and for $r = n$, and show that it solves DLM in the general case, in presence of dynamic failures, and regardless of the number of robots.

[2] In [20], the sequential scheduler is called *asynchronous*.

[3] Note that the semi-synchronous scheduler includes the sequential and fully synchronous ones as special cases.

2 Model and Basic Limitations

2.1 Model and Terminology

In the *Hybrid Programmable Matter* model (see [8, 20]), the space is an infinite triangular lattice $\mathcal{G} = (V, E)$. In this space, a set $\mathcal{R} = \{R_1, ..., R_r\}$ of active agents, called *robots*, operate on a set $\mathcal{T} = \{b_1, ..., b_n\}$ of $n \geq r$ passive hexagonal elements, called *tiles*.

Tiles are not able to move, communicate, or perform any computation.

Robots are identical mobile entities with limited computational and communication capabilities. For our results, robots with the computational power of a finite automaton suffice; they can detect only the status of their immediate neighbourhood, communicate only with robots in that neighbourhood at that time, and move only to a neighboring node. Communication is (modeled as) occurring through the transmission of constant size messages.

A robot is able to pick up a single tile; carry it; and put it on a node that does not contain a tile already; it can also pass a tile to a neighbouring robot. A tile will be said to be *lifted* when picked up or carried by a robot, *posed* otherwise; at any time, on a node there can be at most one posed tile.

Each robot has a *local* sense of orientation (up/down, left/right) that allows it to distinguish between the six adjacent neighbours of the node where it is located.

The system works in synchronized rounds. In each round, one or more robots become active and execute an atomic sequence of operations, called `Look-Compute-Move` (`LCM`) cycle. The `LCM` of activated robot R is composed of the three phases:

- `Look`: R observes its status and that of its immediate neighbourhood. It identifies the nodes where there is a tile, those with a robot, the state of those robots and whether they are carrying a tile. It also receives the message (if any) sent by robots on neighbouring nodes in a previous round. It also accepts the tile (if any) passed by a robot from a neighbouring node in a previous round.

- `Compute`: Using what gathered in the previous phase as input, R executes the algorithm (the same for all robots) to determine its action in the next phase, as well as if and what to communicate to each of its neighbouring robots. If so determined, R prepares and sends the messages to its neighbouring robots.

- `Move`: R performs the action determined in the previous phase. The set of possible actions, in addition to *nil* (no action) are as follows. If R is holding a tile, it can: (i) place it on the node on which it is located, if no tile is posed there already; or (ii) pass it to a neighbouring robot; or (iii) move (with the tile) to a neighbouring node that does not contain another robot. If R is not holding a tile, it can: (iv) pick up the tile from the node it occupies (if any is there); or (v) move to a neighbouring node that does not contain another robot.

Within the round, the execution of each phase by the activated robots is synchronized.

The decision of which robots are active in a given round is assumed to be under the control of an adversary, called activation *scheduler*, whose goal it to make the protocol fail; the only restriction to its power is basic fairness: every robot must be activated infinitely often. Such an adversary is called *semi-synchronous* ($\mathcal{SSYNC}$) in the literature on autonomous mobile robots [18]. Weaker adversaries are the *sequential* scheduler ($\mathcal{SEQNL}$), obtained by constraining $\mathcal{SSYNC}$ to activate only one robot in each round (e.g., [20]), and the *fully synchronous* scheduler ($\mathcal{FSYNC}$), where all robots are activated in every round.

Observe that some basic operations require multiple cycles to be completed. For example, a message transmitted at round t by robot R to a robot R' on neighbouring node v will be (considered received and) processed only in the `Look` phase of the first round $t' > t$ when R' is active next.

At all times, a robot must stand on a node occupied by a tile or adjacent to one such a node. It is further required that, at all times, in the sub-graph $\mathcal{G}'$ of $\mathcal{G}$ induced by the posed tiles and the robots, the non-faulty tiles are connected; this requirement is called *Basic Connectivity Constraint*.

Initially, all the tiles are posed and form a line segment. Let L_0 denote the line of the grid where the segment lies. Initially each robot is on a distinct node occupied by a tile.

A tile may become *faulty* at any time. A robot is able to detect whether or not the tile it is on (or nearby, or carrying) is faulty. There is no constraint on the number f of tiles that will become faulty, nor on the location and time of the occurrence of a fault; in particular, a tile may become faulty while being carried by a robot, and more than one tile might become faulty at the same time.

The Dynamic Line Maintenance (DLM) problem requires the r robots to collaboratively remove any faulty tile from L_0 within finite time from it becoming faulty subject to three constraints: (1) (*Line Maintenance*) all non-faulty tiles eventually are in the same line segment on L_0; (2) (*Basic Connectivity*) at any time, in the sub-grid of the locations occupied by all robots and tiles, all non-faulty tiles are connected; and (3) (*Collisionless*) at no time two robots occupy the same tile. In other words, should no more failures occur, all and only the non-faulty tiles will form a line on L_0.

2.2 Basic Limitations

Under the stated conditions, the problem is unfortunately *unsolvable* even in very simple static settings. In fact, as we show, there exists no deterministic algorithm that would always allow an arbitrary number of robots to restore the line even if there is a single failure in the system (i.e., $f = 1$) and the scheduler is fully synchronous (i.e., $\mathcal{FSYNC}$).

The proof is actually quite simple as this impossibility follows from the impossibility of the robots to break the initial symmetry of the system.

Theorem 1. *There exists no deterministic algorithm that solves the* Dynamic Line Maintenance *problem for all $r > 1$, regardless of the initial position of the robots and of the location of the faulty tiles. This results holds even in $\mathcal{FSYNC}$ with $f = 1$.*

Proof. By contradiction, let $\mathcal{A}$ be a solution algorithm that allows the robots to solve the problem regardless of their number and initial position, and regardless of the number and location of the faulty tiles. Consider a line with an odd number of tiles, of which only the center one is faulty, and where an even number of robots are located symmetrically with respect to the faulty tile. For simplicity let $r = 2$, and let the two robots be located on the extreme tiles of the line. Consider now a fully synchronous execution of $\mathcal{A}$. In this execution, since the robots are identical (i.e., without identifiers or distinguished features) and any attempt by both robots to move on the same tile are forbidden (since collisions must be avoided by any correct algorithm), the two robots will always be on distinct locations. Moreover, in such an execution, at any time, the symmetry of the overall configuration in the grid (location of the tiles and the positions of the two robots) will be necessarily preserved. To solve the problem, however, the symmetry must be broken, because one of the robots must move on the faulty tile (to remove it); such a configuration necessarily creates an asymmetric configuration. A contradiction. □

From the proof of the above theorem, it follows that, to escape the impossibility, there must be some means in the system for the robots to break symmetry. Clearly, this goal can be achieved by endowing the robots with unique identifiers[4], or assuming the a-priori existence of a distinguished robot (the *leader*); these are however quite strong requirements. In reality, it is sufficient for the robots to have some basic level of orientation in the grid, specifically, to agree on the *left/right* direction of the initial line. In fact, with such an orientation, one robot (e.g., the rightmost) can be uniquely singled out and assume the role of leader.

Thus, in view of the impossibility result, in the rest of the paper we will assume the following:

Assumption. *The robots agree on the left/right direction of the initial line L_0.*

Observe that, although each robot might have a distinct orientation of the grid and thus not be able to agree on which parallel line to L_0 is *up* and which is *down*, the leader (once identified though the use of the direction of the line) can communicate its own orientation to the other robots so that, within finite time, they agree also on the *up/down* direction. In other words, they can construct

[4]However, in such a case the robots would not be finite state.

a globally consistent sense of orientation that allows each of them to distinguish between the six adjacent neighbours of the node where they are located; we will denote the neighbours of node u by $right(u)$, $rightup(u)$, $rightdown(u)$, $left(u)$, $leftup(u)$, $leftdown(u)$. Let u be the node of L_0 where a robot is initially located; then, by construction, the nodes $right(u)$ and $left(u)$ are also on L_0 and, without loss of generality, let the nodes $rightup(u)$ and $leftup(u)$ be on line L_1, and the nodes $rightdown(u)$ and $leftdown(u)$ be on line L_{-1}.

Another important observation is that, since faults are dynamic, some operations are necessary regardless of the solution algorithm. More precisely, let a tile $b \in \mathcal{T}$ be said to be *covered* at time t if either it is being carried by a robot or it is posed and there is a robot on it. Then, we have:

Theorem 2. *In any algorithm solving the* Dynamic Line Maintenance *problem, should no more failures occur after time t, all non-faulty tiles must be covered infinitely often.*

Proof. (sketch) By contradiction, let $\mathcal{A}$ be a solution algorithm that does not cover a non-faulty tile, say $b \in \mathcal{T}$, infinitely often in an execution $\mathcal{E}$ where no more failures occur after time t; this means that there exists a time $t' > t$ from which no robot will ever cover b. Consider now an execution $\mathcal{E}'$ which is identical to $\mathcal{E}$ except that b fails at time t'; clearly $\mathcal{A}$ will fail to remove b from the line in this execution, contradicting it correctness. □

As a consequence, as long as there are more posed non-faulty tiles than robots, it is necessary to continuously *patrol* the posed non-faulty tiles to determine whether a failure has occurred.

3 Dynamic Line Maintenance: Basic Modules

As a first step in the development of our solution protocol for the general Dynamic Line Maintenance problem, we consider the problem under two special settings: when there is a single robot ($r = 1$), and when the number of robots is equal to the number of tiles (i.e., $r = n > 1$). In this section, we design and present solution algorithms `DLR1` and `DLR*` for these cases. Both will be used in Section 4 as basic modules of our protocol solving the general problem.

3.1 Algorithm `DLR1`

Initially, all tiles form a line on L_0. Let *FirstNode* and *LastNode* denote the rigthmost and the leftmost node of L_0 containing tiles. Initially, the only robot (called the *Patroller*), is on *LastNode*.

3.1.1 Informal Description

Algorithm `DLR1` is composed of two procedures, `Patrolling` and `Reconfiguration`.

During `Patrolling`, the *Patroller* patrols the line traversing it in alternate direction ("forward" and "backward") looking for faulty tiles. Procedure `Patrolling` is rather straightforward; it uses a variable $direction \in \{right, left\}$, whose value, initialized at $right$, is switched after the patrolling of the line in the indicated direction has been completed. If the *Patroller* finds a faulty tile while patrolling, it starts executing the `Reconfiguration` procedure.

Informally, during `Reconfiguration`, the *Patroller* will remove a faulty tile from the line (moving it to L_1) and, unless that tile is on *FirstNode*, it will be replaced by a non-faulty tile. To replace a faulty tile without breaking the connectivity, the *Patroller* needs first to find and pick up a non-faulty tile, called the *substitute*, removing it from the line. To ensure that this removal does not violate the *connectivity requirement*, the target tile used for substitution is the tile on *FirstNode*. Once the substitute has been picked up, the *Patroller* carries it until it reaches the faulty tile and performs the replacement. Once the replacement is completed, the *Patroller* carries the faulty tile to the front of the line; once there, it places it on L_1, and restarts `Patrolling`.

Due to the fact that any tile might become faulty at any time and that the robot is a finite state machine, this simple high level description of the `Reconfiguration` procedure hides the presence of a variety of different situations that may occur and the *Patroller* must handle. In particular, the tile intended to be used as a substitute might be found to be faulty, or the substitute tile might become

faulty while being carried. Also observe that, when the *Patroller* reaches a faulty tile carrying the substitute tile, it might not be able to determine whether or not this is the one, b, that triggered this replacement operation.

The `Reconfiguration` procedure uses two simple auxiliary operations:

1. `Substitute`(x, y), which starts with the robot holding tile y while standing on the node u where tile x is posed; it ends with y posed at u, with the robot standing there.

2. `Return`(x) which starts with the *Patroller* holding a faulty tile x; it consists of the *Patroller* moving on L_0 until it reaches $FirstNode$; it then places x on $rightup(FirstNode)$ (which, as we will prove, does not contain a tile), and moves back on $FirstNode$.

In the following section, procedure `Reconfiguration` is described in more details. The pseudo-code of all procedures and operations is to be found in Section 3.1.3.

3.1.2 Reconfiguration

The `Reconfiguration` procedure always starts with the *Patroller* on a node u where the posed tile b is faulty; this is for example the case when the *Patroller*, while `Patrolling`, arrives at u and discovers that b is faulty. What happens depends first of all on the location u of the tile.

1. If $u = FirstNode$, then the tile is just removed from L_0 and placed on $leftup(u)$, which, as we will prove, does not contain a tile; the *Patroller* then sets $direction = left$ and restarts `Patrolling`.

2. If $u \neq FirstNode$, the `Reconfiguration` procedure is more complex. The *Patroller* first moves right until it reaches $FirstNode$, with the intention of using the tile c located there as a replacement tile.

   (a) If the tile c is faulty, it is removed from L_0 and placed on $leftup(FirstNode)$, which, as we will prove, does not contain a tile; the *Patroller* then sets $direction = left$ and restarts `Patrolling`.

   (b) If the tile c is not faulty, then the *Patroller* picks it up, sets $direction = left$, and moves on L_0 carrying it until either it arrives on a node u with a faulty tile, or c becomes faulty (recall: since faults can occur anywhere at any time, c might become faulty at any time during the movement).

      i. If c becomes faulty during the movement, as soon as detected this to be the case, the *Patroller* executes operation `Return`(c), and then restarts `Patrolling` with $direction = left$.

      ii. If c does not become faulty during the movement, when the *Patroller* arrives to a node v with a faulty tile d, the *Patroller* performs operation `Substitute`(d, c), moves to $rightdown(v)$, picks up d and moves to $right(v)$ (which, as we will prove, does not contain a tile), followed by operation `Return`(d); it then restarts `Patrolling` with $direction = left$. Note that the tile d that is being substituted in the `Reconfiguration` procedure is not necessarily the same tile b that was detected to be faulty in the most recent traversal by the *Patroller*. In fact, if in the meantime a closer tile has become faulty, it will be substituted first.

Note that, whenever the tile on $u = FirstNode$ is picked up by the robot, then obviously node $v = left(u)$ becomes the new $FirstNode$.

3.1.3 Rules of Algorithm DLR1

The pseudo-code for Algorithm DLR1 and procedures `Patrolling` and `Reconfiguration` is shown in Algorithm 1, Algorithm 2 and Algorithm 3, respectively. Additionally, the rules for the `Substitution` and `Return` operations are presented in Algorithm 4 and Algorithm 5, respectively. For an example of execution of the procedures and the operations, see Figure 1.

Algorithm 1 DLR1

```
1: /* Initially: the Patroller is at node u = LastNode, and direction = right */
2: while !(u = FirstNode and left(u) is without tiles) do
3:     Patrolling()
4: end while
```

Algorithm 2 DLR1: Procedure `Patrolling`

```
1:  /* The Patroller is at node u and b is the posed tile on u */
2: if b is faulty then
3:     execute Reconfiguration
4: else
5:     if direction = left then
6:         if u ≠ LastNode then
7:             move to left(u)
8:         else
9:             set direction := right
10:            move to right(u)
11:        end if
12:    else
13:        if u ≠ FirstNode then
14:            move to right(u)
15:        else
16:            direction := left
17:            move to left(u)
18:        end if
19:    end if
20: end if
```

Algorithm 3 DLR1: Procedure `Reconfiguration`

```
1:  /* The Patroller is at u and the posed tile b on u is faulty */
2: if u = FirstNode then
3:     pick up b
4:     move to leftup(FirstNode)
5:     place b there
6:     move to FirstNode
7:     direction = left
8: else
9:     move to FirstNode; let c be the tile there
10:    pick up c
11:    if c is faulty then
12:        drop c on leftup(FirstNode)
13:        move to FirstNode
14:        direction = left
15:    else
16:        while c and the tile d posed on the current node are not faulty do
17:            move left on L_0
18:        end while
19:        if c is faulty then
20:            execute Return(c)
21:            direction = left
22:        else
23:            execute Substitute(d, c)
24:            move to rightdown(u)
25:            pick up d
26:            move to right(u)
27:            execute Return(d)
28:            direction = left
29:        end if
30:    end if
31: end if
```

Algorithm 4 Operation `Substitute`(x, y)

```
1:  /* The robot is on tile x at node u holding tile y; nodes leftdown(u) and rightdown(u) are empty */
2: move with y to leftdown(u) and place y there
3: move to u and pick up x
4: move with x to rightdown(u) and place x there;
5: move to leftdown(u) and pick up y
6: move to u and place y there
```

Algorithm 5 Operation `Return`(x)

```
1: /* Robot holds x at node u */;
2: move with x to rightup(u) (i.e., reaching L_0)
3: while currentnode ≠ FirstNode do
4:     move right with x on L_0
5: end while
6: move with x to rightup(FirstNode)
7: place x
8: move to FirstNode
```

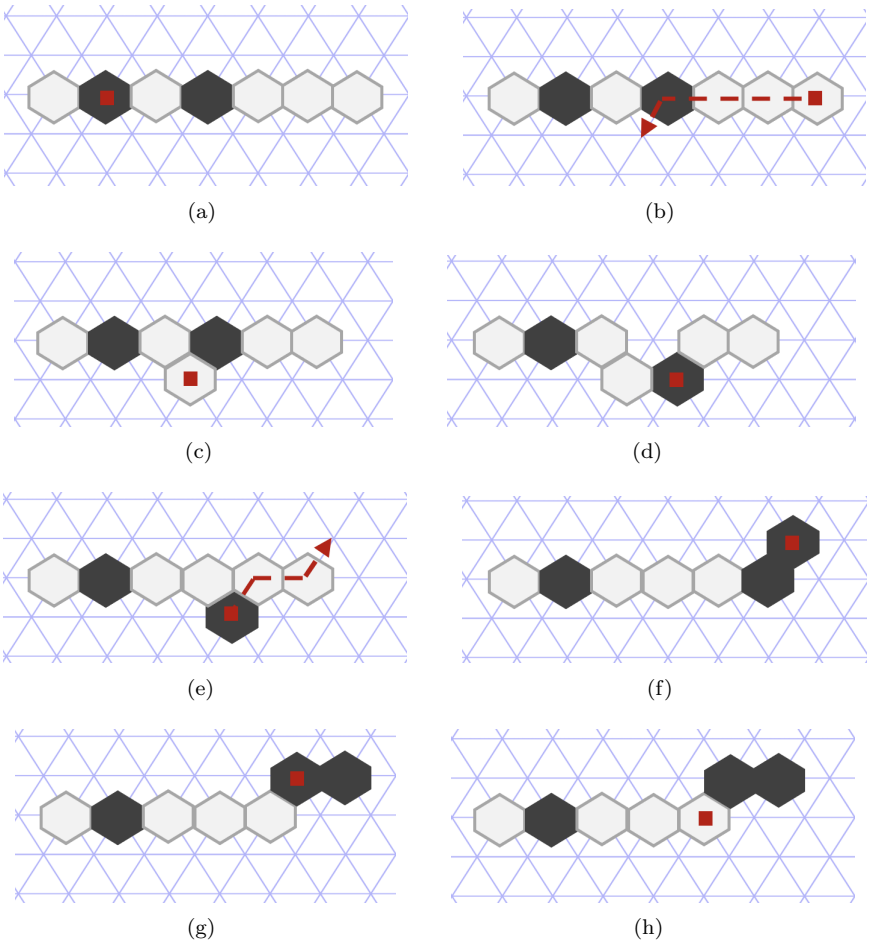


Figure 1: (a) During patrolling, the *Patroller* detects a fault; (b) it goes to $FirstNode$ to get a replacement tile and moves with it towards the faulty tile; (c)-(e) it reaches the closest faulty tile and substitutes it; (e)-(f) it returns to $FirstNode$ to dispose of the faulty tile; (f) during the disposal, $FirstNode$ becomes faulty; (g)-(h) the *Patroller* removes that faulty tile from L_0 and restarts patrolling.

3.1.4 Correctness of Algorithm DLR1

The following two properties trivially hold:

Property 1. *At any time. the* Patroller *can unambiguously detect* FirstNode *and* LastNode.

Property 2. *At any time, if no* `Substitute` *is in progress, all posed non-faulty tiles are in a connected line on* L_0.

Correctness of `DLM1` is based on the following lemmas.

Lemma 1. *1. Any* `Substitute` *operation correctly terminates in finite time.*

2. *Any* `Return` *operation correctly terminates (a faulty tile is placed on* L_1*, connectivity maintained) in finite time.*

3. *Any execution of the* `Reconfiguration` *procedure correctly terminates (at least one faulty tile is removed, connectivity is maintained) in finite time.*

Proof. 1. Procedure $Substitute(x, y)$ starts with the robot holding tile y on the node u where the faulty tile x is located; by construction, $leftdown(u)$ and $rightdown(u)$ are empty. The robot moves to $leftdown(u)$ temporarily placing y there; it then moves back to u to pick up x, and then it places it on $rightdown(u)$; it finally goes to $leftdown(u)$ to take y and places it on u (which is now empty). In the entire process, the the line is never disconnected and, since the procedure cannot be interrupted by other events, it obviously terminates.

2. $Return(x)$ consists of the simple process of the robot moving right along L_0 until reaching *FirstNode*. Since the new tile that substituted x has been taken from $right(FirstNode)$, by construction $rightup(FirstNode)$ is empty (the disposed tiles start from $rightup(right(FirstNode))$, so x can be dropped there, completing the operation.

3. Since there is a single robot continuously executing the algorithm, it proceeds with the substitutions sequentially and it is not interrupted by other events. It then trivial follows that any execution of the `Reconfiguration` operation correctly terminates. □

Since any execution of the `Reconfiguration` operation correctly terminates, for any number of faults f, it is easy to see that:

Lemma 2. *Let* $n > f$*. There exists a time* t *such that:*

1. $\forall t' \geq t$*, all non-faulty tiles form a line on* L_0*, all faulty ones form a line on* L_1*;*

2. *if* $n - f > 1$ *from time* t *the* Patroller *executes* `Patrolling` *perpetually. If* $n - f \leq 1$*, the* Patroller *stops executing the algorithm.*

In conclusion, from the above lemmas we have the following:

Theorem 3. *Algorithm* `DLR1` *solves the* Dynamic Line Maintenance *problem when* $r = 1$.

3.2 Algorithm DLR*

In this section, we present an algorithm, called DLR*, that solves the problem when there are more than one robot, and the number of robots and the number of tiles is the same (i.e., $r = n > 1$).

Initially, all tiles form a compact line on L_0. Let *FirstNodeMonitor* and *LastNodeMonitor* denote the leftmost and the rightmost node of L_0 containing tiles[5]. Initially, all robots are active and each robot is on a different tile; let robot A_i be on tile b_i, where b_1 is the tile on *FirstNodeMonitor* and b_n is the tile on *LastNodeMonitor*. Robot A_1 is in state *FMonitor*, A_n in state *LMonitor* and all others are in state *Monitor*.

The robots continuously monitor the tile they are located on. When faults happen, the robots cooperatively replace the faulty tiles with non-faulty ones to recover the line.

The algorithm is composed of a single procedure which uses two operations: `Substitute` and `Dispose`.

3.2.1 Informal Description

Informally, the procedure works as follows. All the active robots monitor their tile continuously to detect whether it becomes faulty.

If the *FMonitor* robot finds that its tile is faulty, it lifts that tile from the node u where it currently is, and then executes the `Dispose` operation, consisting on moving with the tile to $rightup(u)$. This move generates a change of state of both the *FMonitor* robot, that becomes *Redundant* and inactive, and of its neighbouring robot on L_0, that becomes *FMonitor*; furthermore, the tile where the new *FMonitor* is located becomes the new *FirstNodeMonitor*.

If a robot in state *Monitor* or *LMonitor* finds that its tile is faulty, it originates a *request* for the *FMonitor* to provide a replacement tile. Upon notification of the existence of a request, the robot *FMonitor* picks up the non-faulty tile it is on; this tile is passed along the monitoring robots until it reaches the closest robot whose tile is faulty; that faulty tile is then replaced with the newly arrived tile (operation `Substitute`). After the replacement, the robots cooperate to pass the faulty tile just removed until it reaches the *FMonitor*, say on node u, that will execute the `Dispose` operation, become *Redundant* and notify the robot on $right(u)$ to become *FMonitor*.

This informal description hides the operations needed to correctly handle the totally unpredictable (i.e., adversarial) timing of the occurrence of failures and, as a consequence, of tile requests, of the movements of replacement tiles, of the substitution operations, and of the movements of faulty-tiles to be discarded. The design of the request management mechanism, the delivery of replacement tiles and the disposal of faulty tiles are described in more details next.

3.2.2 Algorithm Description

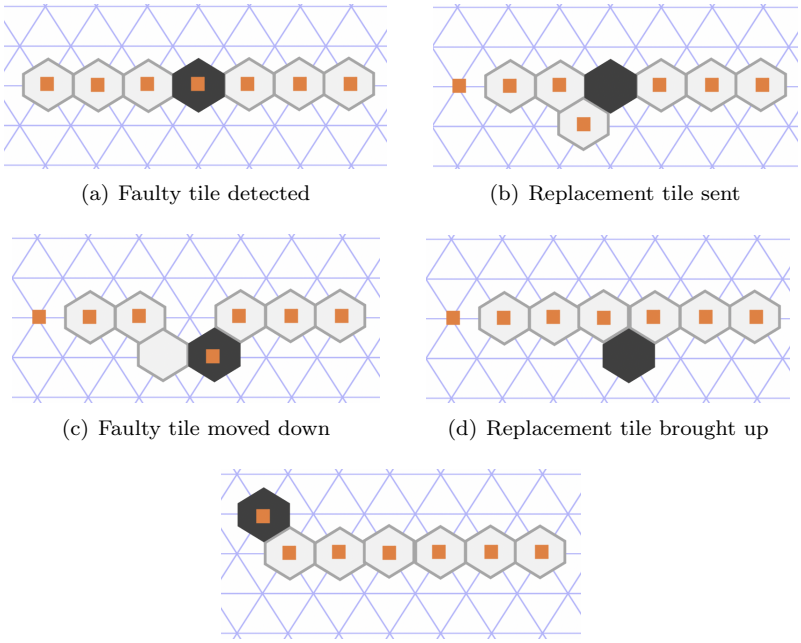
Let robot A on node u discover that the tile it is on has become faulty.

If A is the *FMonitor* robot, then it removes itself and its tile from L_0: it lifts the tile from the node u where it currently is, and executes the `Dispose` operation, moving to $rightup(u)$ and placing the carried tile there. At the end of this operation, A changes its state to *Redundant* and sends a message $\langle BecomeFMonitor \rangle$ to its neighbour on L_0 (i.e., the robot on $right(u)$), which then becomes *FMonitor*.

If A is in state *Monitor* or *LMonitor*, it originates a *request* and sends a $\langle TileRequest \rangle$ message to $left(u)$, its left neighbour on L_0; this request will be said to be *pending* until that faulty tile is replaced. The $\langle TileRequest \rangle$ message originated by A is forwarded by the robots along L_0, until it reaches either the *FMonitor* or a *Monitor* robot B that has already forwarded a still pending request or originated one; two local Boolean variables, *ReceiveRequestFlag* and *StartRequestFlag*, initially set to $false$, are used by each robot to record the reception and the origination of such a message, respectively.

The forwarding of a $\langle TileRequest \rangle$ message by a robot R is paused in two cases: (i) should the recipient robot be involved in the `Substitute` operation (described later), the forwarding resumes

[5]Note that this is the opposite of what happens with *FirstNode* and *LastNode* in DLR1, which are instead the leftmost and the rightmost node, respectively; the reason for this will become clear in Section 4



(a) Faulty tile detected (b) Replacement tile sent

(c) Faulty tile moved down (d) Replacement tile brought up

(e) Faulty tile passed and disposed

Figure 2: Substitution and disposal of a faulty tile after its detection.

after that operation is completed; (ii) should the recipient robot be the *FMonitor* executing the `Dispose` operation, when that operation terminates, no more forwarding is needed: R becomes the new *FMonitor*, and will act as if it just received the request, unless it is the originator of that request, in which case it executes the `Dispose` operation.

The `Substitute` operation always starts with a robot R with a replacement tile in hand (call it c) on a node u where the posed tile b is faulty. The `Substitute` is done by R taking the replacement tile c to node $leftdown(u)$, moving back to u to pick up b, bring b to $rightdown(u)$ to momentarily drop it there, move again to $leftdown(u)$ to pick up c, move back to u to place c there (see Figure 2).

Since both the `Substitute` and the `Dispose` operations terminate in a constant number of rounds, we have:

Property 3. *Should any robot other than* FMonitor *detect that its tile has become faulty, the* FMonitor *will eventually receive a* ⟨TileRequest⟩ *message.*

When the *FMonitor* receives a ⟨*TileRequest*⟩ message, if its tile is non-faulty, the *FMonitor* uses its tile as a replacement and starts the process, we shall call `Replacement Transfer`, of transferring the replacement along the line; it does so by lifting the tile it is on, passing it to its neighbour on L_0, and waiting for a faulty tile to arrive.

This replacement tile, as long as it does not become faulty, is passed by the robots along the line until it reaches the closest robot C on L_0 whose tile is faulty. When this happens, C uses the received tile to replace its faulty tile by performing the `Substitute` operation.

After the substitution, the faulty tile is picked up and brought to u, then passed from robot to robot until it reaches *FMonitor* with the note ⟨*Return-Success*⟩.

Should instead the replacement tile become faulty while being passed, this tile is moved back to *FMonitor* with the note ⟨*Return-Failure*⟩.

Since no other `Substitute` or `Dispose` is performed during this process, we have:

Property 4. *Within finite time from sending a non-faulty replacement tile, the* FMonitor *will receive a faulty tile.*

Once the *FMonitor* robot receives the faulty tile, it disposes of the tile by executing the `Disposal` operation; it then notifies its neighbour on L_0 that becomes the new *FMonitor*. If the note had been $\langle$*Return-Success*$\rangle$, the substitution has been successful and no action is needed by the new *FMonitor*. On the other hand, if the note had been $\langle$*Return-Failure*$\rangle$, the substitution has not yet occurred; hence, a new `Replacement Transfer` process must be started: the new *FMonitor* will operate as if just receiving a $\langle$*TileRequest*$\rangle$ message.

Observe that, if the accompanying note is $\langle$*Return-Success*$\rangle$, a pending request has been satisfied. This means that, unless C is aware of the existence of another pending request (i.e., its $ReceiveRequestFlag = true$), the values of the *ReceiveRequestFlag* of every *Monitor* robot between C and *FMonitor* will be reset to $false$ upon reception of the $\langle$*Return-Success*$\rangle$ note.

3.2.3 Rules of Algorithm `DLR*`

The algorithm prescribing the robots' behavior is described, for each robot's state, by a corresponding set of rules: the behavior of a robot in state $Monitor$ or $LMonitor$ is described in Algorithm 8; the behavior of a robot in state $FMonitor$ is described in Algorithm 6. The `Substitute` and `Dispose` operations are shown in Algorithm 4 and Algorithm 7, respectively.

Algorithm 6 DLR* (State *FMonitor*)

```
1:  /* Robot R is on tile b at node u */ :
2:
3:  Case {b is faulty} do
4:  if u = LastMonitorNode then
5:      terminate the algorithm
6:  else
7:      pick up b
8:      execute Dispose(b)
9:      note ← Normal
10:     send ⟨BecomeFMonitor; note⟩ to robot on right(u)
11: end if
12:
13: Case {receive a ⟨TileRequest⟩ message from right(u)} do
14: pick up b
15: pass b to the robot on right(u)
16:
17: Case {receive a faulty tile c with message ⟨Return; note; flag⟩ from right(u)} do
18: execute Dispose(c)
19: send message ⟨BecomeFMonitor; note⟩ to robot on right(u)
```

Algorithm 7 Procedure `Dispose`(x)

```
1:  /* Robot R is on node u holding tile x */ :
2:  move to rightup(u)
3:  drop x
4:  state ← Redundant
```

Algorithm 8 DLR* (State *Monitor* or *LMonitor*)

```
1: /* Robot R is on tile b at node u */:
2:
3: Case {b is faulty} do
4: StartFlagRequest ← true
5: if ReceiveRequestFlag = false then
6:     send a ⟨TileRequest⟩ message to left(u)
7: end if
8:
9: Case {receive a ⟨TileRequest⟩ message from right(u)} do
10: if ReceiveRequestFlag = false then
11:     ReceiveRequestFlag ← true
12:     if StartRequestFlag = false then
13:         send ⟨TileRequest⟩ to left(u)
14:     end if
15: end if
16:
17: Case {receive a tile c from the robot on left(u)} do
18: if c is not faulty but b is faulty then
19:     execute Substitute(c, b)
20:     move to rightdown(u)
21:     pick up b and move to u
22:     StartRequestFlag ← false
23:     note ← Return.Succes
24:     flag ← ReceiveRequestFlag
25:     send b to the robot on left(u) with the message ⟨Return; note; flag⟩
26: else
27:     if c is not faulty then
28:         pass the tile to the robot on right(u)
29:     else
30:         return c to the robot on left(u) with the message ⟨Return; Return.Failure; true⟩
31:     end if
32: end if
33:
34: Case {receive a faulty tile from the robot on right(u) with ⟨Return; note; flag⟩ message} do
35: if note = Return.Sucess and flag = false then
36:     ReceiveRequestFlag ← false
37: end if
38: pass the tile and the message to the robot on left(u)
39:
40: Case {receive ⟨BecomeFMonitor; note⟩} do
41: state ← Fmonitor
42: if b is faulty then
43:     pick up b
44:     execute Dispose(b)
45:     forward the message to robot on right(u)
46: else/* b is not faulty */
47:     if ReceiveRequestFlag = true or note = Return.Failure then
48:         lift b
49:         pass it to the robot on right(u)
50:     end if
51: end if
```

3.2.4 Correctness

From Properties 3 and 4, the following lemma is immediate:

Lemma 3. *Let the* FMonitor *robot R start the* `Replacement Transfer` *process.*

1. *If the replacement tile a does not becomes faulty while being carried, a faulty tile b will be replaced by a and will be posed on L_1; R will become* Redundant *and its neighbour on L_0 will become* FMonitor*.*

2. *If the replacement tile a becomes faulty while being carried, a will be posed on L_1; R will become* Redundant*, and a new* Replacement Transfer *process will be started by the new* FMonitor*.*

3. *During the entire process, the* Basic Connectivity Constraint *is maintained.*

Observe that, in Lemma 3, while a faulty tile is always removed from L_0, a pending request is satisfied only in case (1). We now show that eventually, every pending request is satisfied.

Theorem 4. *Let $P(t)$ denote the set of requests still pending at time t. Then: $\forall t \geq 0$, if $P(t) \neq \emptyset$, then $\exists t' \geq t$ such that $|P(t') \cap P(t)| < |P(t)|$.*

Proof. Consider the time t_0 when a $\langle$*TileRequest*$\rangle$ arrives at the *FMonitor* robot that we shall denote by $R[t_0]$.

By contradiction, assume that, for all $t' > t_0$, $|P(t') \cap P(t_0)| \geq |P(t_0)|$. This means that, by Lemma 3, the result of the `Replacement Transfer` process started by $R[t_0]$ ends in the original replacement tile, we shall denote by $b[t_0]$, becoming faulty and returned to $R[t_0]$, that will dispose it on L_1. Upon $R[t_0]$ becoming *Redundant*, always according to by Lemma 3, another robot, say $R[t_1]$, becomes *FMonitor* and starts at time $t_1 > t_0$ a new `Replacement Transfer` process. By assumption, also this process fails; and so will fail any process, started by the *FMonitor* $R[t_{i+1}]$ at time $t_{i+1} > t_i$ after the failure of the one started by the *FMonitor* $R[t_i]$ at time t_i. In other words, $|P[t_i]| \geq |P[t_{i-1}]|$.

Let $k[t_0]$ be the number of consecutive non-faulty tiles posed starting from *FirstMonitorNode* at time t_0. All `Replacement Transfer` processes end in failures; however, since each failure results into the leftmost tile removed from L_0 and moved to L_1, after at most $k[t_0]$ failed processes, the faulty tile closest to $R[t_0]$ when it received the $\langle$*TileRequest*$\rangle$ at time t_0 is precisely the one on *FirstNodeMonitor*. According to the algorithm, in this case the *FMonitor* moves that tile directly to L_1 (without starting a new process); when this happens, the pending request associated to this tile has been resolved. A contradiction. □

Theorem 5. *Let $f(t)$ denote the number of faulty tiles at time t. Let no more failures occur after time $\hat{t} \geq 0$ and at least one tile be non-faulty. Then there is a time $t^* \geq \hat{t}$ such that $\forall t \geq t^*$:*

1. *all $n' = n - f(\hat{t})$ non-faulty tiles form a compact line on L_0, and all $f(\hat{t})$ faulty tiles form a compact line on L_1;*
2. *$r' = min\{r, n'\}$ robots are on distinct tiles on L_0 and the other $r - r'$ robots are on distinct tiles on L_1,*

Proof. By Lemma 3, it follows that each `Replacement Transfer` process, whether successful or not, results into the current *FMonitor* on node u moving with the faulty tile to $rightup(u)$ and staying there; similarly, each time the *FMonitor* on node u discovers that the tile on u is faulty, it moves with the faulty tile to $rightup(u)$ and stays there. By Theorem 4, every faulty tile is eventually replaced and removed (or removed directly if the *FMonitor* robot is on it). Thus, if no more failures occur after time $\hat{t} \geq 0$, there is a time $t' \geq \hat{t}$ when all the faulty tiles are placed on L_1 into consecutive nodes, each with a robot on it; on the other hand, since only the robots that in state *FMonitor* dispose of a faulty tile move and remain on L_1, there is a time $t'' \geq \hat{t}$ where all the non-faulty tiles form a line on L_0, each with a robot on it. Hence, at time $t^* = \max\{t', t''\}$, both statements of the theorem hold. Since in either situation no action is taken anymore by the robots, the theorem holds. □

In conclusion, from the lemmas above we have the following:

Theorem 6. *Algorithm* `DLR*` *solves the* Dynamic Line Maintenance *problem with with $n = r$ robots.*

4 The General Algorithm DLR

We now describe the general algorithm, `DLR`, that solves the problem regardless of the number r of robots, $1 \leq r \leq n$.

In the special cases of $r = 1$ and $r = n$, the algorithm is precisely `DLR1` and `DLR*`, respectively, without modifications. In all other cases (i.e., $1 < r < n$), the algorithm `DLR` is composed of a careful concurrent composition of algorithms `DLR1` and `DLR*`, appropriately modified and described below.

Observe that, although the robots have no knowledge of r or n, they can easily determine the relationship between these two values, and hence execute the correct module of the algorithm, through a preprocessing phase described later (Section 4.2.6).

For simplicity, the description will first assume $r > 2$; the algorithm will work with some simplifications for $r = 2$, as discussed later.

4.1 Overall Strategy

Informally, the idea of the algorithm is to dynamically integrate the operations of `DLR1` and `DLR*`. Initially the line segment is divided into two parts, the *monitored area* controlled by $r-1$ robots (the *Monitors*) placed on $r-1$ consecutive tiles, like in Algorithm `DLR*`, and the *patrolled area* explored by one robot (the *Patroller*), as in Algorithm `DLR1` (see Figure 3).

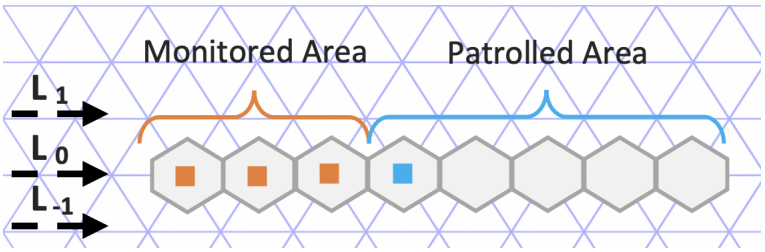


Figure 3: Initial configuration of the line segment.

Failures in the patrolled area are generally treated like in `DLR1`, while those in the monitored areas are generally treated like in `DLR*`. Exceptions are failures occurring at the "border" between the two areas, where more care has to be taken.

Various new operations are introduced involving the border between the two areas. The first is the reintegration of the patroller: if the *Patroller* realizes that there is only a single tile to be patrolled then, instead of terminating its execution (as in `DLR1`), it joins the execution of `DLR*` becoming *LMonitor*.

The most significant change is the *reintegration of monitors*. After the substitution of a monitored tile, when the *FMonitor* robot disposes of the faulty tile, it becomes *Redundant*; instead of being discarded, like done in `DLR*` (where it is no longer used), in this algorithm the *Redundant* robot will be reintegrated into active monitoring; it will do so by travelling on L_1 to reach the border between the two areas, and moving down to L_0 to monitor a new tile (see Figure 4). The reintegration of redundant robots allows the algorithm to make use of all the available robots. This operation will be possible at time t as long as the number $p(t)$ of nodes being patrolled at that time is more than one.

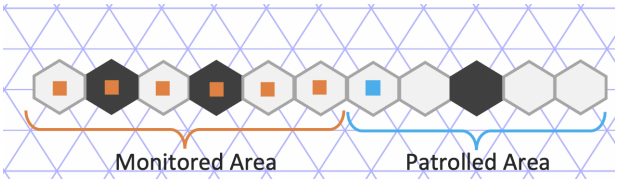
If faults continue to occur, eventually the *Patroller* is reintegrated and the line will contain only the monitored area (see Figure 7). When/if this occurs, there is no possibility for *Redundant* robots to be reintegrated, and the end of the reintegration process starts. At this point, the *Redundant* robots will move to occupy the faulty tiles on L_1 on the right (the ones disposed of by the *Patroller*, see Figure 7 (a)) and become *Done*. It is possible that all these tiles become occupied (this situation is easily detected by a redundant robot). In such a case, since there is no more room on the faulty tiles on the right, all current/future redundant robots will become *Superfluous* and will move to occupy the faulty tiles on the left side (see Figure 7 (b) and (c)).

The coordination between the *patroller* and the monitors, as well as the management of the reintegration process, involve handling several new situations originated by the concurrent execution of two algorithms. This requires various technicalities to avoid breaking the connectivity, creating deadlocks, and having robots collide (i.e., be on same cell at the same time).

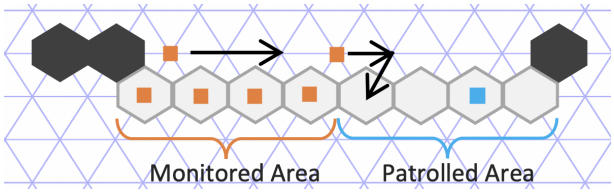
4.2 Algorithm Description

Initially, the tiles form a compact line on L_0 with tile b_i on node u_i, $1 \leq i \leq n$, where u_1 is the leftmost node of the line containing a tile. The r robots are on the r leftmost nodes of the line: robot R_j is on tile b_j at node u_j, $1 \leq j \leq r$. Recall that, since the robots are finite state machines, generally they cannot compute the values of r and n; they can however easily determine whether $r = 1$ or $r = n$, and hence whether $1 < r < n$.

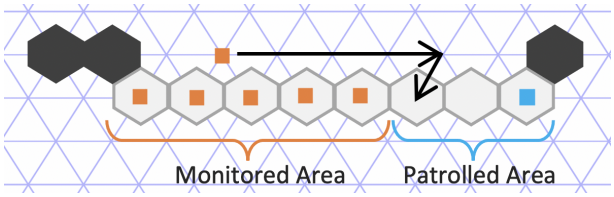
Let $1 < r < n$. In this case, initially: robot R_r is in state *Patroller*, and executes a modified `DLR1`; the other robots execute a modified `DLR*` with R_{r-1} in state *LMonitor*, R_1 in state *FMonitor*



(a) The line with some faults



(b) Two *Redundant* robots on their way to reintegration



(c) The monitored area changes as *Redundant* robots join.

Figure 4: Reintegration of monitors.

and all others in state *Monitor*. Initially, u_1 is the *FirstNodeMonitor*, u_{r-1} the *LastNodeMonitor*, u_r the *LastNodePatrol*, and u_n the *FirstNodePatrol*.

Details of modifications needed in both `DLR1` and `DLR*` to perform the reintegration, as well as to correctly perform the other parts of the algorithm, are described below.

4.2.1 Reintegration of the Patroller

Let us first describe a simple modification needed to be made to `DLR1`. If the *Patroller* P determines that there is only a single tile to be patrolled ($u = LastNodePatrol$ and there is no tile on $right(u)$) then, instead of terminating its execution of the algorithm (as would happen in `DLR1`), P sends a $\langle BecomeMonitor \rangle$ message to the robot on $left(u)$ (which is the current *LMonitor*), changes state to *LMonitor* and joins the execution of `DLR*`.

As part of this reintegration, when the *LMonitor* robot receives a $\langle BecomeMonitor \rangle$ message, it changes state to *Monitor*.

4.2.2 Reintegration of the Redundant robots

The following set of modifications to `DLR*` is more complex and crucial, and focus on state *Redundant*. Recall that, when a robot R becomes *Redundant*, it is at a node of line L_1 on a faulty tile that it has just placed there. After sending a message $\langle BecomeFMonitor \rangle$ to the robot on $rightdown(u)$ (that will change its state to *FMonitor*), instead of terminating its execution of the algorithm (as would happen in `DLR*`) R does as follows. It moves rightwise on L_1, pausing if another *Redundant* robot is in front of it, and continues until its leftdown neighbour is *LMonitor* (see Figure 4 (*b*)).

Let u be the node where R is when this occurs. The intention of R is to move to $v = rightdown(u)$ (if/when it is without robots) and rejoin the monitors. However, this operation has to be done carefully; in fact, a direct move by R to v might create a collision should the *Patroller* be at $right(v)$ at that time (and, thus, not visible by R) and about to move to v. To prevent this possibility, rather then moving directly to v, R performs a *double-step*: it moves first to $right(u)$, if without robots.

Then, if v is without robots and the tile is non-faulty, and if the *Patroller* is not at v, R moves from $right(u)$ to v (see Figure 4 (b) and (c)).

Note that, before the second step toward v, if R sees that v is occupied by a robot or does not contain a tile, or contains a faulty tile, a variety of situations are possible; for example, it is possible that a substitution is now ongoing involving v, or that the *Patroller* is now visiting v as part of normal operation. In all cases, to avoid a potential collision on v, R waits until the situation changes and can safely descend to v. Also note that, if $right(v)$ is occupied by the *Patroller*, R can move safely to v because the *Patroller* will refrain from going there as described in Subsection 4.2.3(b).

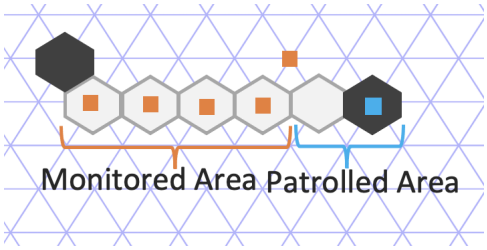
Once the double-step is completed, R must notify the robot on $left(v)$ (the old *LMonitor*) to become *Monitor*, and change state from *Redundant* to *LMonitor* to join the execution of `DLR*`. However, also this operation has to be done carefully. The reason is that, by changing its state to *LMonitor*, R would enable the next *Redundant* robot to descend to $right(v)$ creating the potential of a collision with the *Patroller* should it be now in $downright(v)$ (having just completed a substitution). For these reasons, should R see the *Patroller* in $downright(v)$, it waits for the *Patroller* to move away before assuming the role of *LMonitor*.

4.2.3 Other interactions at the border

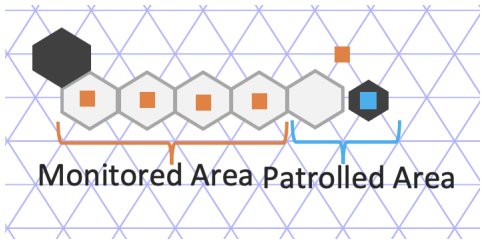
In addition to the reintegration of *Patroller* and *Redundant* robots, other actions performed at the border between the patrolled and the monitored areas require attention to avoid possibility of collisions.

a) **The Patroller disposing of a faulty tile on FirstNodePatrol:** A special rule is needed when the *Patroller* is on $FirstNodePatrol$ (call it u) and finds a faulty tile. According to `DLR1`, the *Patroller* should dispose of the faulty tile bringing it to $leftup(u)$. However, such an action is dangerous in this case, as it could potentially create a collision with a *Redundant* robot (this could happen if $FirstNodePatrol$ is adjacent to $LastNodePatrol$, in which case there could be a *Redundant* robot that the patroller cannot see, about to move precisely to $leftup(u)$). To avoid this issue, the final destination of the faulty tile is still $leftup(u)$, but, instead of moving directly there, the *Patroller* performs a double-step going first to $left(u)$ (which, by construction is surely empty) and then to $rightup(left(u))$ (if/when without robots). Should the *Patroller* holding the faulty tile on $left(u)$ see a *Redundant* robot R on $leftup(u)$, it passes the faulty tile to R and becomes *LMonitor* (telling the current *LMonitor* to become *Monitor*). Upon receiving the faulty tile, R deposits the tile and continues the algorithm still in state *Redundant*. In the similar case that R is on $leftup(u)$ and sees P on u holding the faulty tile, it waits for P to move to $left(u)$ and passes the faulty tile. It is also possible that the *Patroller* on u is in the second step of disposing the faulty tile found in *FirstNodePatrol* and the *Redundant* robot R is on $leftup(u)$ in the first step of its reintegration. In this case, the *Patroller* moves to $rightup(u)$, drops the faulty tile, and returns to u while R waits for this operation to be finalized. For an example of execution of these rules see Figure 5.

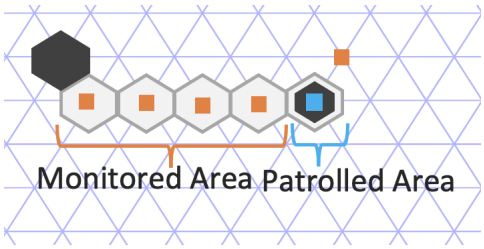
b) **The Patroller sees a Redundant robot about to descend:** If the *Patroller* P at u sees on $leftup(u)$ a *Redundant* robot R on a node without a tile, communication between R and P is needed to distinguish between two possible situations that require attention: either (i) R is in the second step of its reintegration about to descend to u and P is in the first step of disposing of a faulty *FirstNodetPatrol* is about to move to u (as described above in 4.2.3(a)) or (ii) R is in the second step of its reintegration about to descend to u and P is about to move to u for its regular patrolling. Case (i) has been already treated in a). In case (ii), priority is given to R or to P depending on the status of the tile on u. ($ii1$) As long as u does not contain a faulty tile P waits for R to descend before continuing the patrolling; should u be the last node of the segment when R descends on u, R becomes *Monitor*, P becomes *LMonitor* and both robots join algorithm `DLR*`. ($ii2$). If instead u contains a faulty tile, P proceeds with the substitution and R waits for the substitution to be concluded (see Figure 6).



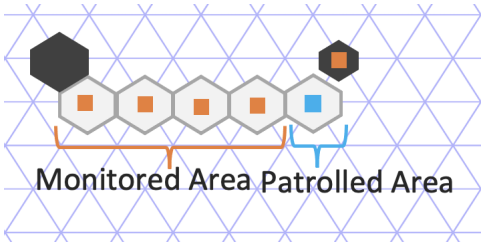
(a) The *Patroller* finds the tile on *FirstNodePatrol* faulty



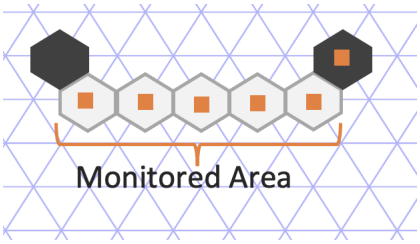
(b) The *Patroller* lifts the faulty tile and the *Redundant* robot is about to descend to L_0



(c) The *Patroller* moves with the faulty tile to *FirstNodePatrol*



(d) The *Patroller* passes the faulty tile to the *Redundant*

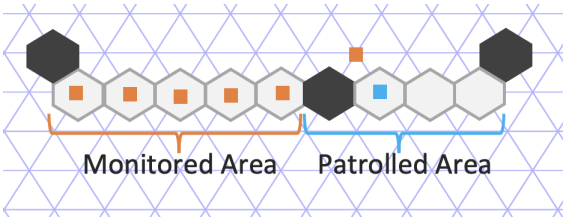


(e) The *Redundant* disposes the faulty tile and becomes *Done*. The *Patroller* becomes *LMonitor* and updates its state

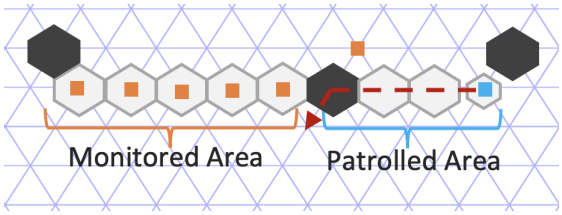
Figure 5: The *Patroller* disposes a faulty tile on *FirstNodePatrol*. A small hexagon around a robot indicates a carried tile.

c) **LastNodePatrol is faulty:** Communication between *LMonitor* and the *Patroller* is needed when the *Patroller* finds that *LastNodePatrol* is faulty (and is not the only node of the patrolled area). In this case, the *Patroller* notifies the *LMonitor* from *LastNodePatrol* and waits for acknowledgment before starting the substitution; it then notifies again when the substitution is completed.

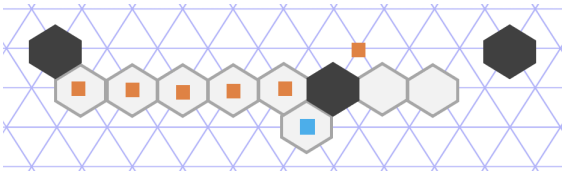
d) **LastNodeMonitor is faulty:** The substitution can proceed only if there is no substitution currently ongoing involving *LastNodePatrol* (note that *LMonitor* is always aware of ongoing substitutions involving *LastNodePatrol* because of Rule 4.2.3(c) above.



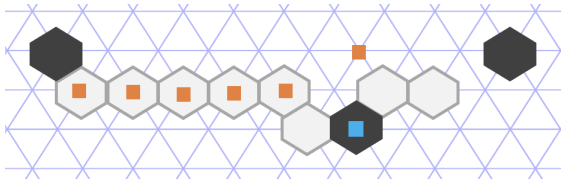
(a) *Redundant* waits for the *Patroller* to substitute the faulty tile



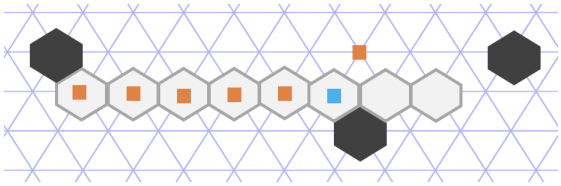
(b) *Patroller* lifts the non-faulty tle on the *FirstNodePatrol* as the replacement tile



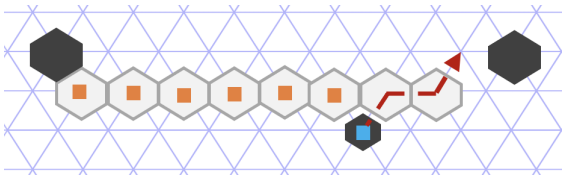
(c) *Patroller* starts the `substitute` operation



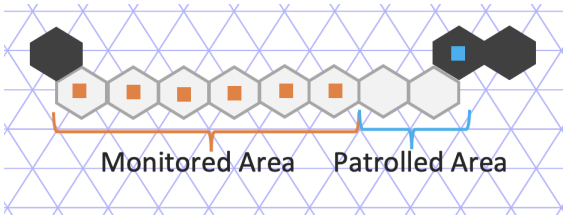
(d) *Patroller* removed the faulty tile from L_0



(e) *Patroller* replaces the faulty tile with non-faulty tile



(f) *Patroller* lifts the faulty tile and *Redundant* performs the second step of its double step



(g) *Patroller* disposes the faulty tile

Figure 6: The *Patroller* sees a *Redundant* robot about to descend. The small hexagon around a robot indicates a carried tile.

4.2.4 End of reintegration

When the patrolled area is composed of a single node (or two nodes, one of which faulty), there is no possibility for *Redundant* robots to be reintegrated. At this point, the *Redundant* robots will migrate to occupy the faulty tiles on L_1 which have been disposed of by the *Patroller* (i.e., the ones posed on the right of the segment, see Figure 7 (a)). Once a robot reaches the end of these faulty tiles, it becomes *Done* and any incoming *Redundant* robot seeing a *Done* robot on its right will also become *Done*. It is possible that all these tiles become occupied (this situation is easily detected by a *Redundant* robot). In such a case, since there is no more room on the faulty tiles on the right, all further *Redundant* robots will migrate to the left side to occupy the faulty tiles which have been posed there by the monitors.

The change of direction in the *Redundant* robots' migration requires proper coordination to avoid collisions with *Redundant* robots which could still be moving to the right. This will be achieved through communication among the robots and change of state.

This is started when a *Redundant* robot R on node u with no posed tile finds a *Done* robot on $right(u)$. At this point, R starts the "broadcast" of a $\langle NoMore \rangle$ message to be communicated (through local communication among neighbouring robots) to all robots on L_0, as well as to the ones

(if any) on L_1 on the left of u. This message will make any *Redundant* robot, including R, wait for an $\langle Ack \rangle$ message. This message will be broadcasted to the right by the *FMonitor* once it receives the $\langle NoMore \rangle$ message. All redundant robots receiving the $\langle Ack \rangle$ become *Superfluous*, except for R that becomes *FSuperfluous*. At this point, R (as well as any *Superfluous* robot, if any) moves to the left, pausing if there is another superfluous robot on its left, becoming *Done* if it reaches the end of the faulty tiles or it sees a *Done* robot on its left.

Note that an additional synchronization is necessary during this migration process to avoid the collision between the *FMonitor* (disposing of its faulty tile) and a *Superfluous* robot. This is accomplished by having the *FMonitor* wait, after the broadcast of $\langle Ack \rangle$, until the passage of the *FSuperfluous* robot R. From this moment on, when a *FMonitor* goes to L_1 to dispose of a faulty tile, it becomes *Superfluous* instead of *Redundant*.

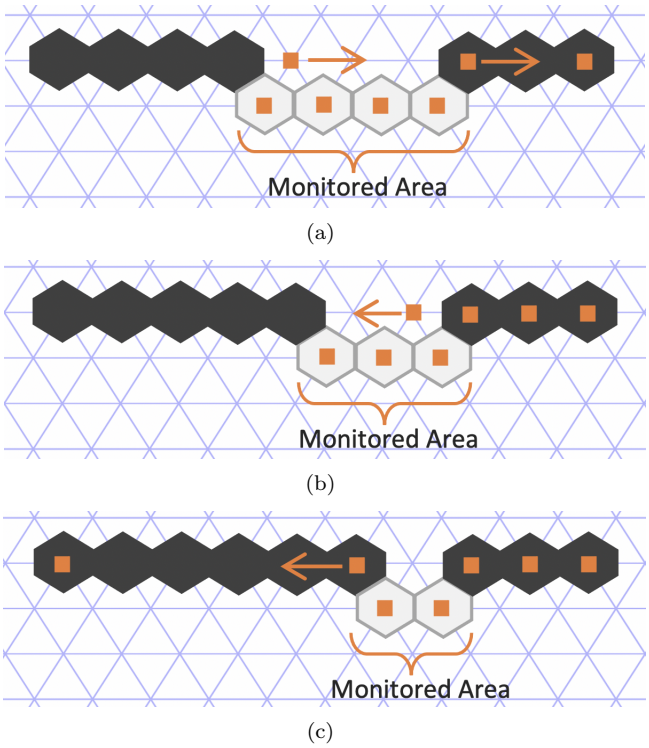


Figure 7: No more reintegration is possible: (a) *Redundant* robots migrating to the right; (b) a robot detects that no more room is available on the right side, after broadcasting a $\langle NoMore \rangle$ message and receiving acknowledgment, starts migrating to the left; (c) *Superfluous* robots move to the left side.

4.2.5 The case $n > r = 2$

When the system is composed of at least three tiles but of only two robots (a case easily detectable by the robots), the algorithm conceptually follows the same rules as the ones described above, with the *Patroller* visiting the patrolled area and the single monitor controlling a monitored area of size one. Clearly, in this situation, the single monitor has to act both as *LMonitor* and as *FMonitor*. In general, all the rules, while being conceptually the same as for $r > 2$, are subject to slight technical simplifications. In particular, the reintegration of monitor after becoming *Redundant* (Rule 4.2.2), as well as the end of reintegration (Rule 4.2.4) become straightforward.

4.2.6 Preprocessing

As mentioned before, although the robots have no knowledge of r or n, they can easily determine the relationship between these two values, and hence execute the correct module of the algorithm.

This can be achieved by the robots, initially all in the same state, say *Pre*, executing the following simple pre-processing phase.

Let u and v be the rightmost and leftmost node of the line containing a posed tile. Informally, the robots start the pre-processing phase by moving in the *right* direction. The robot R that reaches first (or is already on) u changes its state to *P-Patroller*, moves to $leftup(u)$ and moves left on L_1.

If the *P-Patroller* reaches $leftup(v)$ without detecting any robot (moving) on L_0, it correctly determines that $r = 1$; it then moves to v becomes *Patroller*, and executes `DLR1`.

If, instead while moving left on L_1, R encounters *Pre* robots (moving) on L_0, it understands that $r > 1$. It thus notifies them that $r > 1$, and instructs them to become *Hold* and to stop moving. R continues to move left on L_1 until it reaches $leftup(v)$. When this occurs, all other robots are on L_0, not moving, in state *Hold*. Importantly, by the time R reaches $leftup(v)$, it is able to determine whether or not $r = n$.

Indeed, $r = n$ if and only if R, during its move from $leftup(u)$ to $leftup(v)$, only sees on L_0 nodes with a robot on them. Should this be the case, R returns to u, becomes *Rmonitor*, and sends a notification to all robots Upon notification, the *Hold* robot on v becomes *Lmonitor*, any other *Hold* robot becomes *Monitor*, and can start the execution of `DLR*`.

In the case $r < n$, R returns to u, notifying on the way all the other robots, becomes *P-Patroller* and moves left on L_0. Upon notification, the *Hold* robots become *P-Monitor* and move left. Hence, eventually all robots move in the left direction on L_0 and, within finite time, they occupy the leftmost r tiles of the line, all in state *P-Monitor*, except for the rightmost robot, which is in state *P-Patroller*. The *P-Monitor* robot on v changes its state to *Lmonitor* and notifies its neighbour; the notification transforms the robot in *Monitor*, and is forwarded until it is received by the *P-Monitor* next to the *P-Patroller*, which become *Rmonitor* and *Patroller*, respectively. A final notification is sent by the *Patroller* allowing the robots to start the execution of `DLR`.

Further observe that, if the *Patroller* and *Lmonitor* are neighbours, they both detect that $r = 2$.

4.3 Rules of Algorithm `DLR`

The algorithm prescribing the robots' behavior is described, for each robot's state, by a corresponding set of rules as indicated below:

- State $Patroller$: Algorithm 9
- State $LMonitor$: Algorithm 8 with the addition of Algorithm 10
- State $FMonitor$: Algorithm 6 with the addition of Algorithm 11
- State $Monitor$: Algorithm 8 with the addition of Algorithm 12
- State *Redundant*: Algorithm 13
- State $FSuperfluous$: Algorithm 14
- State $Superfluous$: Algorithm 15

The `Substitute`, `Reconfiguration`, and `Dispose` operations are shown in Algorithm 4, 3, and 7, respectively.

Algorithm 9 DLR (State *Patroller*)

```
1: /* Rules for robot P in state Patroller on node u, with direction ∈ {right, left}; b is the posed tile on u (if any) */
2:
3: Case {u = LastNodePatrol, b is faulty, P holds a non-faulty tile c } do
4: send ⟨ Substitute-Start ⟩ to left(u)
5: wait to receive ⟨ Ack-Substitute ⟩
6: execute Substitute(u, c)
7: send ⟨ Substitute-Done ⟩ to left(u)
8:
9: Case {u = FirstNodePatrol, b is faulty} do
10: lift b
11: move to left(u)
12: /* first step of the double-step */
13: if rightup(u) = v is empty then
14:     /* u is the FirstNodePatrol */
15:     move to v
16:     drop b
17:     move to u
18: else /* v is occupied by a robot R */
19:     pass b to R
20: end if
21:
22: Case {leftup(u) contains a Redundant robot R without a tile} do
23: /* R is in its 2nd step of double-step */
24: communicate with R
25: if b and the tile on left(u) are non-faulty, and u is not FirstNodePatrol then
26:     wait for R to move to left(u)
27: else if b is faulty and u is FirstNodePatrol then
28:     lift b
29:     move to left(u)
30:     /* first step of double-step */
31: else if the tile of left(u) is faulty and u is not FirstNodePatrol then
32:     move to left(u)
33:     execute Reconfiguration
34: end if
35:
36: Case {u = LastNodePatrol and right(u) has no tile} do
37: send ⟨BecomeMonitor⟩ to left(u)
38: state ← LMonitor
39: execute DLR*
40:
41: Case {Otherwise} do
42: if b is faulty then
43:     execute Reconfiguration
44: else
45:     if u ∈ {LastNodePatrol,FirstNodePatrol} then
46:         change direction
47:     end if
48:     move to direction
49: end if
```

Algorithm 10 DLR (State *LMonitor*)

```
1: /* Additional rules for robot R on node u: */
2:
3: Case {receive ⟨BecomeMonitor⟩ from right(u)} do
4: state ← Monitor
5:
6: Case {receive ⟨Substitute-Start⟩ from right(u)} do
7: if no substitution is ongoing then
8:     send ⟨Ack − Substitute⟩ to right(u)
9: end if
10:
11: Case {u = lastNodeMonitor contains a faulty tile, and R holds a non-faulty tile b } do
12: if receive ⟨Substitute − Start⟩ from right(u) then
13:     wait for receiving ⟨Substitute − Done⟩ from right(u)
14: else
15:     Substitute(u, b)
16: end if
```

Algorithm 11 DLR (State *FMonitor*)

/* Additional and modified rules for robot R on node u: */

Case {receive $\langle NoMore \rangle$} **do**
broadcast $\langle Ack \rangle$ to the $right$ direction
wait to receive $\langle Resume \rangle$

Case {receive $\langle Ack \rangle$ and holding a faulty tile b} **do**
Dispose(b)

Algorithm 12 DLR (State *Monitor*)

/* Additional rules for robot R on node u: */

Case {receive $\langle NoMore \rangle$} **do**
forward $\langle Nomore \rangle$ to the $left$ direction

Case {receive $\langle Ack \rangle$} **do**
forward $\langle Ack \rangle$ to the $right$ direction

Algorithm 13 DLR (State *Redundant*)

/* Robot R on node u on L_1; let $v = rightdown(u)$ */:

Case {$leftdown(u)$ contains a $Monitor$ and $right(u)$ is empty} **do**
move to $right(u)$

Case {$leftdown(u)$ contains a *LMonitor*} **do**
if $right(u)$ contains an empty non-faulty tile **then** /* first step of the double-step */
    move to $right(u)$
    if v contains an empty non-faulty tile and *Patroller* is not at v **then** /* second step of the double-step */
        move to v
        if $rightdown(v)$ is empty **then**
            send $\langle BecomeMonitor \rangle$ to $left(v)$ /* the robot on $left(v)$ is the *LMonitor*; $right(v)$ is now the *LastNodePatrol* */
            $state \leftarrow LMonitor$
        end if
    end if
    if receive a faulty tile b from v **then**
        place b on u /**Patroller* disposing of a faulty tile at $FirstNodePatrol$*/
    end if
end if

Case {v and $left(v)$ are occupied by robots and $right(u)$ contains a faulty tile} **do** /*No possibility for *Redundant* robots to be reintegrated*/
while $right(u)$ contains an empty faulty tile **do**
    moves to $right(u)$
end while
if $right(u)$ contains no faulty tile or contains a robot in *Done* state **then**
    $state \leftarrow Done$
end if

Case {v and $left(v)$ are occupied by robots and $right(u)$ contains a *Done* robot on a faulty tile} **do** /* all of the faulty tiles are occupied*/
broadcast $\langle NoMore \rangle$

Case {originated a $\langle NoMore \rangle$ broadcast and receive $\langle Ack \rangle$} **do**
$state \leftarrow FSuperfluous$

Case {receive $\langle NoMore \rangle$ and $\langle Ack \rangle$} **do**
$state \leftarrow Superfluous$

Algorithm 14 DLR (State *FSuperfluous*)

```
1:  /* Robot R on node u:*/
2:
3:  Case {left(u) is empty and v = leftdown(u) contains robot} do
4:  if v is occupied by FMonitor then
5:      send a ⟨Resume⟩ message to it
6:  end if
7:  move to left(u)
8:
9:  Case {left(u) contains faulty tile without robot and v = leftdown(u) is empty} do
10: if left(u) contains no faulty tile or left(u) contains a robot in Done state then
11:     state ← Done
12: end if
13: move to left(u)
```

Algorithm 15 DLR (State *Superfluous*)

```
1:  /* Robot R on node u: */
2:
3:  Case {left(u) is empty and v = leftdown(u) contains robot} do
4:  move to left(u)
5:
6:  Case {left(u) contains faulty tile without robot and v = leftdown(u) is empty} do
7:  if left(u) contains no faulty tile or left(u) contains a robot in Done state then
8:      state ← Done
9:  end if
10: move to left(u)
```

Algorithm 16 Procedure `Dispose`(b)

```
1: /* Robot R is on node u holding tile b after receiving ⟨Ack⟩: */
2:
3: move to rightup(u)
4: drop b
5: state ← Superfluous
```

4.4 Correctness

The correctness of the individual modules `DLR1` and `DLR*` has been shown in Sections 3.1.4 and 3.2.2, respectively. We now show that the general algorithm `DLR`, created by the integration of these two modules correctly solves the problem. To do so we show that, with the modifications described and discussed above, any potential problem created by the concurrent execution of the modules does not occur; more precisely, there are no collisions nor deadlocks. For ease of presentation, we shall refer to the rule described in sub-Section 4.2.i simply as Rule 4.2.i.

We first show that no collisions are created during the reintegration of redundant robots.

Lemma 4. *No collision can occur between a* Redundant *robot and the* Patroller.

Proof. The *Patroller* P moves on L_0 (when performing the regular patrolling), on L_1 (when performing a `return` operation or disposing a faulty *FirstNodePatrol*), and on L_{-1} (when performing a `substitution` operation), while a *Redundant* robot R only moves on L_1 and L_0 to reintegrate among the monitors. A collision may occur between R and P in four cases.
Case 1) R is in the second step of reintegration (it wants to descend to node u) and P also wants to move to u for its regular patrolling. In this case, by construction (Rule 4.2.3(b)), if u contains a non-faulty tile, P waits until R has descended to node u before continuing the patrolling. If u contains a faulty tile, R waits for P to perform the substitution.
Case 2) R is in the second step of reintegration (it wants to descend to node u), another *Redundant* robot R' is on L_0 changing its state to *LMonitor*, while P is on L_{-1} terminating a substitution and is about to move up to L_0. Note that R cannot descend to L_0 until it sees R' in state *LMonitor*.

On the other hand, by construction (Rule 4.2.2), R' checks the *downright* tile and does not change state until P moves away. In doing so, no collision between R and P can occur.
Case 3) R (on $leftup(u)$) is in the first step of reintegrating and P (on u) is in the second step of disposing the faulty *tile* found in *FirstNodePatrol*. In this case, by construction (Rule 4.2.3(a)), P moves to $rightup(u)$, drops the faulty tile and returns to u while R waits for the operation to be completed.
Case 4) R is in the second step of its reintegration (about to descend to node u) and P is in the first step of disposing the faulty *FirstNodePatrol* (about to move to node u). In this case, by construction (Rule 4.2.3(b)) R at $rightup(u)$ waits for P to move to u to pass the faulty tile.

□

We now focus on the procedures `Substitute`, `Dispose`, and `Return` employed by the algorithm, and show that their execution always terminate without collisions.

Lemma 5.
(i) Procedure `Substitute`*, when executed, always terminates without collisions.*
(ii) Procedures `Dispose` *and* `Return`*, when executed, always terminate without collisions.*

Proof. (*i*) There are at most two concurrent executions of `Substitute`, one in the monitored area and one in patrolled area. If neither occurs at the border of the two areas, the claim trivially holds. Consider when there are executions of `Substitute` at the border: the *LMonitor* is replacing *LastNodeMonitor* and/or the *Patroller* is replacing *LastNodePatroller*.

- Only *LastMonitorNode* is faulty: *LMonitor* starts the `substitute` operation as there is no substitution currently ongoing involving the *LastNodePatrol* (Rule 4.2.3(d)).
- Only *LastPatrolNode* is faulty: the *Patroller* interacts only with *LMonitor*, exchanging a message and the rest of the substitution proceeds by construction without collisions (Rule 4.2.3(c)).
- *LastMonitorNode* and *LastPatrollerNode* are faulty: *Patroller* informs *LMonitor* from *LastNodePatrol* and waits for receiving acknowledgement before staring the substitution. Once the substitution is completed, it informs the *LastMonitor* (Rule 4.2.3(c)). Therefore, only one substitution is allowed to proceed at the border at a time with priority given to the substitution of *LastPatrollerNode* by the *Patroller*. While the substitution is ongoing, by construction it will proceed until completion without interference from other robots.

(*ii*) There is at most one execution of `Dispose` operation and/or one execution of the `Return` operation. The potential collision may occur between *FMonitor* (disposing of its faulty tile) and a *Superfluous* robot, as well as between the *Patroller* (disposing of the faulty tile) and a *Redundant* robot. By construction, *Fmonitor* waits after broadcasting the message until the passage of *FSuperfluous* robot. The potential collision between the *Patroller* and *Redundant* has been considered in Lemma 4. □

The following is easy to see:

Lemma 6.
(i) Every Superfluous *robot becomes* Done *within finite time.*
(ii) Every Redundant *robot becomes either* Done *or* LMonitor *within finite time.*

We now show that there are no deadlocks in the system:

Lemma 7. *During the execution of Algorithm* `DLR`*, no deadlocks occur.*

Proof. The following waiting situations happen during the execution of the algorithm:

1. The *Redundant* robot R is in the second step of reintegration at node u, wanting to descend to node $v = leftdown(u)$, and the *Patroller* P wants to move to v for its regular patrolling. Let v contain a non-faulty tile. In this case, R and P are neighbours and they can communicate:

P informs R that it wants to descend to v, and P waits for R to descend to v before continuing the patrolling (Rule 4.2.3(b)). P restarts the regular patrolling as soon as either R moves to v or the tile on v becomes faulty, in which case P proceeds with a substitution and R waits for the substitution to be completed. By Lemma 5 the substitution will eventually be completed and P will move away to dispose of the tile. Note that the tile might become faulty again and R might again be activated when that happens; also in this case, eventually the wait will be resolved. This process might be repeated several times, resolved every time. In the worst case, this will happen until there is no more room for reintegration; that is, when upon activation, R finds that the patrolled area contains one node and a faulty tile has been passed by P (Rule 4.2.3(a)).

2. A *Redundant* robot R at node u is in the same situation as in the previous poit: in the second step of the reintegration, it wants to descend to $v = leftdown(u)$). If R sees that v is *i*) occupied by a robot, or *ii*) does not contain a tile, or *iii*) contains a faulty tile, then it has to wait until the situation changes. Let us show that in all cases, the waiting condition will be eventually resolved.
   i) Let v be occupied by a robot (necessarily *Patroller* P). If P is performing the patrolling, it will by construction move away to continue its regular patrolling and, when coming back, it will stop to wait for R to descend (Rule 4.2.3(b)). If P is substituting a faulty tile, by Lemma 5 the substitution will eventually be completed and P will move away to dispose of the faulty tile. Note that the tile might become faulty again and R might always be activated when that happens: following the same reasoning as in point 1), the waiting condition will be eventually resolved. *ii*) and *iii*) In the case that v does not contain any tile or contains a faulty tile, it means that the *Patroller* P is performing (or will perform) a `Substitute` operation; by Lemma 5 the substitution will eventually be completed and a non-faulty tile will be placed the on v. Following the same reasoning as in point 1), the waiting condition will be eventually resolved.

3. The *Redundant*, at node u, has completed the *double-step* and has to notify the *LMonitor* to become *Monitor*. In the case that it sees the *Patroller* in $downright(u)$, it has to wait for the *Patroller* to move away. By Lemma 5, the *Patroller* will conclude the `substitute` operation and will move away to dispose the faulty tile at $rightup(FirstNodePatrol)$.

4. The *Redundant* robot R is in the first step of its reintegration and *Patroller* is in the second step of disposing the faulty *FirstNodelPatrol*. In this case, R will wait for this operation to be completed; specifically, if *Patroller* is at *FirstNodePatrol* holding the faulty tile, R waits at $leftup$(*FirstNodePatrol*); if *Patroller* is at $right$(*FirstNodePatrol*) holding the faulty tile and R is at $rightup$(*FirstNodePatrol*), it waits for *Patroller* to move to *FirstNodePatrol* and pass the faulty tile. In the first situation, the *Patroller* will move $rightup$(*FirstNodePatrol*), drop the faulty tile, and return to the $FirstNodePatrol$. In the second situation the R will receive the faulty tile from the *Patroller* and place it on the $rightup$(*FirstNodePatrol*) (Rule 4.2.3(a)). In both situation the waiting condition is resolved.

5. If the *LastNodePatrol* is faulty, the *Patroller* notifies the *LMonitor* and waits for the acknowledgement before starting the substitution. Once the *Lmonitor* receives message, it sends an acknowledgement if there is no substitution involving *LastNodeMonitor* ongoing. Otherwise, it sends the acknowlegement once the `substitute` operation is completed and the faulty tile is passed for being disposed (Rule 4.2.3(c)).

6. If the *LastNodeMonitor* and *LastNodePatrol* are faulty, *LMonitor* waits until the substitution of *LastNodePatrol* is completed. The *Patroller* always notifies the *LMonitor* before starting the substitution, it then notifies again when the substitution is completed. Since the substitution will eventually terminate, the waiting condition will be resolved.

7. The *Redundant* robots that received the $\langle NoMore \rangle$ message, have to wait to receive the $\langle Ack \rangle$ message. As the $\langle NoMore \rangle$ is broadcasted, it is received by the *FMonitor*. Next, the *FMonitor* will broadcast the $\langle Ack \rangle$ message to the right, reaching all of the robots.

8. After the broadcast of the $\langle Ack \rangle$ message, *Fmonitor* waits until the passage of *FSuperfluous* robot. After receiving the $\langle Ack \rangle$ message, all the redundant robots become *Superfluous* except the one that started the broadcast of the $\langle NoMore \rangle$ message that becomes *FSuperfluous*. At this point all of them start moving to the left until they reach the end of the faulty tiles on the left. Eventually, *FSuperfluous* will pass by *Fmonitor*.

□

Termination of the reintegration process then follows:

Lemma 8. *The end of reintegration process terminates correctly.*

Let $f(t)$ denote the number of faulty tiles at time t. From the above sequence of lemmas, we have:

Lemma 9. *Let no more failures occur after time $\hat{t} \geq 0$ and at least one tile be non-faulty. Then there is a time $t^* \geq \hat{t}$ such that $\forall t \geq t^*$:*
(i) All $n' = n - f(\hat{t})$ non-faulty tiles form a compact line on L_0, and all $f(\hat{t})$ faulty tiles form a compact line or two compact lines on L_1.
(ii) $r' = min\{r, n'\}$ robots are on distinct tiles on L_0 and the other $r - r'$ robots are on distinct tiles on L_1 in state Done*.*
(iii) If $r' < n'$, then the rightmost robot on L_0 is in state Patroller *and the others are monitoring the leftmost $r' - 1$ tiles; if $r' = n'$, then each tile on L_0 is monitored by a robot.*

In conclusion:

Theorem 7. *Algorithm* `DLR` *solves the* Dynamic Line Maintenance *problem with an arbitrary number of robots.*

5 Conclusions

In this paper, we started the investigation of computing in the *Hybrid Programmable Matter* model in presence of multiple robots operating concurrently and of dynamic failures of the tiles. We considered the problem of maintaining a line formation of tiles when any tile can stop functioning at any time, and when the number of robots is arbitrary. In our solution, the robots, which have the power only of finite state machines and have only local communication capabilities, cooperate to rearrange the line as faulty tiles are discovered, substituted (to restore the line shape of the non-faulty tiles), and disposed of; all of this is done avoiding collisions, deadlocks, and disconnections.

This work opens several new research directions. These include, for example, the examination of possible speed-up vs the increased complexity due to coordination among the robots, and the study of the maintenance of more complex shapes.

6 Acknowledgments

We would like to thank the anonymous referees for their insightful comments, which allowed us to improve the presentation. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grant program.

References

[1] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[2] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.

[3] V. Bonifaci, K. Mehlhorn, and G. Varma. Physarum can compute shortest paths. *Journal of Theoretical Biology*, 309:121 – 133, 2012.

[4] K. C. Cheung, E. D. Demaine, J. R. Bachrach, and S. Griffith. Programmable assembly with universally foldable strings (moteins). *IEEE Transactions on Robotics*, 27(4):718–729, 2011.

[5] G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of the International Conference on Robotics and Automation*, pages 1:449–1:455, 1994.

[6] J. Czyzowicz, D. Dereniowski, and A. Pelc. Building a nest by an automaton. *Algorithmica*, pages 1–33, 2020.

[7] J. Daymude, K. Hinnenthal, A.W. Richa, and C. Scheideler. Computing by programmable particles. In P. Flocchini, G. Prencipe, N. Santoro (Eds.): *Distributed Computing by Mobile Entities*. Springer, 2019.

[8] J. Daymunde, R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, A. Richa, D. Rudolph, C. Scheideler, and T. Strothmann. Towards hybrid programmable matter : shape recognition, formation, and sealing algorithms for finite automaton robots. In *3rd Highlights of Algorithms conference (HALG)*, 2018.

[9] E. D. Demaine, S. P. Fekete, C. Scheffer, and A. Schmidt. New geometric algorithms for fully connected staged self-assembly. *Theoretical Computer Science*, 671:4–18, 2017.

[10] E. D. Demaine, M. J. Patitz, R. T. Schweller, and S. M. Summers. Self-assembly of arbitrary shapes using rnase enzymes: Meeting the Kolmogorov bound with small scale factor. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 201–212, 2011.

[11] Z. Derakhshandeh, S. Dolev, R. Gmyr, A. Richa, C. Scheideler, and T. Strothmann. Brief announcement: Amoebot – a new model for programmable matter. In *26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 220–222, 2014.

[12] G. A. Di Luna, P. Flocchini, G. Prencipe, N. Santoro, and G. Viglietta. Line recovery by programmable particles. In *19th International Conference on Distributed Computing and Networking (ICDCN)*, pages 1–10, 2018.

[13] G.A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33:69–101, 2020.

[14] S. Dolev, S. Frenkel, M. Rosenbli, P. Narayanan, and K. M. Venkateswarlu. In-vivo energy harvesting nano robots. In *Proc. of ICSEE*, pages 1–5, 2016.

[15] D. Doty. Theory of algorithmic self-assembly. *Commun. ACM*, 55(12):78–88, 2012.

[16] S. P. Fekete, R. Gmyr, S. Hugo, P. Keldenich, C. Scheffer, and A. Schmidt. Cadbots: algorithmic aspects of manipulating programmable matter with finite automata. *Algorithmica*, pages 1–26, 2020.

[17] S. P. Fekete, E. Niehs, C. Scheffer, and A. Schmidt. Connected reconfiguration of lattice-based cellular structures by finite-memory robots. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 60–75, 2020.

[18] P. Flocchini, G. Prencipe, and N. Santoro. Moving and computing models: Robots. In *Distributed Computing by Mobile Entities*, pages 3–14. Springer, 2019.

[19] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, and C. Scheideler. Shape recognition by a finite automaton robot. In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*, 2018.

[20] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, and T. Strothmann. Forming tile shapes with simple robots. *Natural Computing*, 19(2):375–390, 2020.

[21] R. Gmyr, I. Kostitsyna, F. Kuhn, C. Scheideler, and T. Strothmann. Forming tile shapes with a single robot. In *Abstr. European Workshop on Computational Geometry (EuroCG)*, pages 9–12, 2017.

[22] F. Hurtado, E. Molina, S. Ramaswami, and V. Sacristán. Distributed reconfiguration of 2d lattice-based modular robotic systems. *Autonomous Robots*, 38(4):383–413, 2015.

[23] K. Li, K. Thomas, C. Torres, L. Rossi, and C. Shen. Slime mold inspired path formation protocol for wireless sensor networks. In *International Conference on Swarm Intelligence*, pages 299–311. Springer, 2010.

[24] E. Niehs, A. Schmidt, C. Scheffer, D. E. Biediger, M. Yanuzzi, B. Jenett, A. Abdel-Rahman, K. C. Cheung, A. T. Becker, and S. P. Fekete. Recognition and reconfiguration of lattice-based cellular structures by simple robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[25] N. Nokhanji, P. Flocchini, and N. Santoro. Fully dynamic line maintenance by a simple robot. In *8th International Conference on Automation, Robotics and Applications (ICARA)*, pages 108–112, 2022.

[26] N. Nokhanji, P. Flocchini, and N. Santoro. Fully dynamic line maintenance by hybrid programmable matter. In *24th IPDPS Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, 2022.

[27] N. Nokhanji and N. Santoro. Line reconfiguration by programmable particles maintaining connectivity. In *9th International Conference on Theory and Practice of Natural Computing*, pages 157–169. Springer International Publishing, 2020.

[28] N. Nokhanji and N. Santoro. Self-repairing line of metamorphic robots. In *7th International Conference on Automation, Robotics and Applications (ICARA)*, pages 55–59, 2021.

[29] M. J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

[30] P. W. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[31] N. Schiefer and E. Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *Proc. of the International Conference on DNA Computing and Molecular Programming*, pages 34–54, 2015.

[32] T. Strothmann. *Self-* algorithms for distributed systems: programmable matter & overlay networks*. PhD thesis, Universität Paderborn, 2017.

[33] T. Toffoli and N. Margolus. Programmable matter: concepts and realization. *Physica D*, 47(1):263–272, 1991.

[34] J. E. Walter, J. L. Welch, and N. M. Amato. Distributed reconfiguration of metamorphic robot chains. *Distributed Computing*, 17(2):171–189, 2004.

[35] D. Woods, H. Chen, S. Goodfriend, N. Dabby, E. Winfree, and P. Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, pages 353–354, 2013.