J Supercomput (2014) 69:908–929
DOI 10.1007/s11227-014-1196-8

An energy and memory-efficient distributed self-reconfiguration for modular sensor/robot networks

Hicham Lakhlef · Hakim Mabed · Julien Bourgeois

Published online: 9 May 2014

Abstract Self-reconfiguration for mobile microrobots currently needs a positioning system and a map of the target shape. Traditional positioning solutions, such as GPS or multilateration are not applicable in the micro-world, and maps sharing does not scale. In the literature, if we want a self-reconfiguration of microrobots to a target shape that consists of millions of positions, each microrobot should have a memory capacity of at least million positions. Therefore, this is not scalable. In this paper, nodes do not record any position. We present self-reconfiguration methods where nodes are unaware of their positions and where they do not have the final coordinates of each microrobot. In other words, nodes do not store the coordinates that build the target shape. Therefore, memory usage for each node is hugely reduced to $O(1)$ and communications are limited to neighboring nodes. These algorithms aim to improve the logical topology of a set of microrobots by restructuring their physical topology. To that end, we consider here the case of restructuring a set of microrobots from a chain to a square and we study two algorithms: the first algorithm ensures the connectivity of the network at the end of the algorithm, where the second guarantees the connectivity of the network through the execution time. The paper presents both analytical and experimental assessments of the algorithms performances using the declarative language *Meld* and executed under the Dynamic Physical Rendering Simulator (DPRSim).

H. Lakhlef (✉) · H. Mabed · J. Bourgeois
UFC/FEMTO-ST, UMR CNRS 6174, 1 cours Leprince-Ringuet, Montbeliard, France
e-mail: hlakhlef@femto-st.fr

H. Mabed
e-mail: hakim.mabed@femto-st.fr

J. Bourgeois
e-mail: julien.bourgeois@femto-st.fr

Keywords Distributed algorithm · Self-reconfiguration · Energy-efficiency · Mobility · MEMS microrobot

1 Introduction

Recent advances in micro- and nano-technologies made possible the design and the development of a large variety of micro-electro-mechanical systems (MEMS) which are miniaturized and low-power devices that can sense and act (http://www.standford.edu/class/mems/ee321, http://www.xs4all.nl/ganswijk/chipdir/m/sensor.htm). It is expected that these small devices, referred to as MEMS nodes, will be mass-produced, making their production cost almost negligible. Their applications will require a massive deployment of nodes, thousands or even millions [11,40], which will give birth to the concept of Distributed Intelligent MEMS [3].

These last years, impressive progresses have been reported in producing miniaturized autonomous robots [22,26] that even scaled down to less than 1 mm [12], this last category will be referred as microrobots or node in the rest of the paper. A microrobot is defined as a system which can sense, act and take decisions by itself, i.e., without relying on a central processing and actuation unit [22].

The potential applications of such microrobots are tremendous, whether working at an individual micro-scale or grouped to act at a macro-scale. Considering the batch-fabrication process of MEMS components, microrobots will be cheap and, once ready, will be part of our daily lives. The most promising application is the *Claytronics project* [1,2,7,8,27] where the aim is the design of a programmable matter. The building blocks of Claytronics are millimeter-scale microrobots that stick together and rotate around each other.

Modular microrobots topic is gaining an increasing attention since large-scale swarms of robots will be able to perform several missions and tasks in a wide range of applications such as odor localization, firefighting, medical service, surveillance, search, rescue, and security. To achieve these tasks the microrobots have to do the self-reconfiguration. In the literature, the self-reconfiguration can be seen from two points of view. On the one hand, it can be defined as a protocol, centralized or distributed, which transforms the physical topology created by the interconnected nodes to another physical topology which is more optimal for communication and exchange [10]. On the other hand, the physical topology can be changed by moving the network nodes, changing the way they are connected and, thus, changing the overall shape of the network [8,31]. Both definitions have one thing in common which is the change of the physical form, but they differ in the purpose of the reconfiguration. The objective of the first definition is the optimization of logical topology for better communication complexity, and the purpose of the second definition is the reconfiguration to any desired shape. The self-reconfiguration is difficult to control as it involves the distributed coordination of a large number of identical modules connected in time-varying ways. The range of exchanged information and the amount of displacement, determine the communication and the energy complexity of the distributed algorithm. When the information exchange involves close neighbors and the algorithm does not

need a map of the target shape, the complexity is moderate and the resulting distributed self-reconfiguration scales gracefully with network size.

An open issue is whether distributed self-reconfiguration would result in an optimal configuration with a moderate complexity in message, execution time, number of movements and memory usage.

This work takes place within the Claytronics project and aims at optimizing the logical topology of the network through rearrangement of the physical topology, as we will see in the next sections. While part of the Claytronics project, this system could be applied to any work which has the same constraints and objectives.

2 Related works

The term *self-organization* is often used interchangeably with self-reconfiguration, though it is also used to express the partitioning and clustering of ad hoc or wireless networks to groups called *cliques* or *clusters*. The term self-organization can be found in protocols for sensor networks to form polygon or sphere from a central node [23, 42]. Redeployment is also a new term to address the self-reconfiguration for sensor networks [14,20,21,24,30].

For the self-reconfiguration with robots or microrobots there are the protocols [31, 33] where the desired configuration is grown from an initial seed module, and a generator inputs from a 3D CAD model of a desired configuration and outputs a set of overlapping blocks. In the second step this representation is combined with a control algorithm to produce the final configurations. These solutions are using the predefined positions of the target shape. In [36–38], the authors present protocols of self-reconfiguration for hexagonal metamorphic robots for specific shape using the map of the target shape.

A growing number of researches with centralized algorithms on this subject has been done in the literature, among them we find the proposed control algorithms for self-assembly and/or reconfiguration in [13,29]. Other approaches give each unit a unique ID and the predefined positions of the target shape; see [41]. The disadvantages of these methods are the centralized computing and the need to individually identify nodes. More distributed approaches that use the predefined positions of the final shape include [6,9,32,34]. The authors of [33,35] have demonstrated algorithms for self-reconfiguration and directed growth of cubic units based on gradients and cellular automata. The authors in [4] have shown how a simulated modular robot (Proteo) can self-configure into useful and emergent morphologies when the individual modules use local sensing and local control rules.

Claytronics, which stands for *clay-electronics* is the name of a robotics project by Carnegie Mellon University. Within Claytronics micro-scale robots are called *Catoms*. The idea of having hundreds of thousands of Catoms that assemble together to create new solid objects of any shape or size. The challenges are vast, but once a reality it will have many applications. Much like the cells in a body or complex organism, each small member of the ensemble engages to do its own part and communication between parts results in a unified form.

A growing body of research is done in Claytronics project. In [7], the authors show a metamodel for the configuration of Catoms beginning from an initial configuration to achieve a second desired configuration using *creation* and *destruction* of Catoms, the authors use these two functions due to the inability of motion of Catoms in the presence of neighbors that can be considered as barriers. In [8,27], the authors present a scalable distributed reconfiguration algorithm with the Hierarchical Median Decomposition, to achieve arbitrary target configurations without a global communication. In [2], the authors present a scalable protocol for Catoms self-reconfiguration with the MELD language [1,28] using the creation and destruction of nodes. In these works the positions of the target shape are predefined, and each Catom knows all correct positions at the beginning, consequently each node needs a large memory space to save these positions. Also, authors assume each node is aware of its current position. In contrast to existing solutions, in our paper the self-reconfiguration is without predefined positions of the target shape. Furthermore, the protocol is distributed and works with anonymous networks (without ID for nodes). The first not deeply studied idea of self-reconfiguration without map appears in [15–18]. This paper details the idea and studies the impact of the connectivity. We presented in [19] an algorithm of reconfiguration from any starting physical topology to a square, this algorithm does not ensure the connectivity of the network during the reconfiguration and needs more memory compared to these algorithms.

3 Contributions

In this paper, we propose a new approach for asynchronous self-reconfiguration of modular microrobots where the target form is built incrementally, and each node in the current increment acts as a landmark point for other nodes to form the next increment. Our algorithms are asynchronous in the sense that each node runs the instructions independently and it is responsible for counting the rounds itself, without dependence on other nodes rounds. The proposed model makes the assumption that each node can obtain the state of all its neighbors to achieve self-reconfiguration. Using these states the nodes collaborate and help each other, without global information from the network. Our algorithms are scalable and do not depend on the network size. Contrary to existing works, our algorithms do not need information on the correct positions of the target shape. Consequently, memory usage is reduced to $O(1)$. The nodes are unaware of their positions on the plan, knowing that to have the localization for nodes, we should go either with using GPS that provides accurate positions. However, GPS consume a significant amount of energy, which is leading researchers to find famous protocols using the multilateration method [5,25,39] where the error is always a problem. In all applications of self-reconfiguration knowing the exact location is an important factor to receiving expected programmed behavior. This paper provides a self-reconfiguration standalone and portable, because it is independent of the map that builds the target shape.

We propose scalable algorithms for nodes self-reconfiguration differentiated by the type of motion considered for nodes. In our protocols, the network is modeled as a chain that we will be converted to a square that represents the best physical topology

for message exchange. We analyze the complexity of the number of messages sent and the number of movements (energy). In our paper, the number of movements for each node is predicted, so each node can make sure that it has correctly followed the algorithm and this makes the algorithm energy-aware. We make a comparative study between algorithms depending on the type of motion. At the end, we present the results of simulation made with the declarative language MELD [1] and the open source simulator DPRSim [43]. By choosing a straight chain as the initial shape, we aim to study the performance of our approach in extreme case. Indeed, the chain form represents the worst physical topology for many distributed algorithms in terms of fault tolerance, propagation procedures and convergence. First, the number of direct contacts between microrobots is minimal and secondly the average distance between two robots (in terms of number of hops) is of $(n + 1)/3$, where n is the number of robots. Also, a chain of microrobots represents the worst case for message broadcasting complexity with $O(n)$. The redeployment into a square organization allows to obtain the best messages broadcasting complexity with $O(\sqrt{n})$.

The remainder of this paper is organized as follows: Sect. 4 discusses the model and terminology. Section 5 discusses the proposed algorithms. Section 6 analyzes the number of messages sent and the number of movements, shows how to generalize and extend the algorithms, and discusses the simulation results. Finally, Sect. 7 summarizes our conclusions and illustrates our suggestions for future work.

4 Model and definitions

Within Claytronics, a Catom (Fig. 1) referred throughout this paper as a *node* is modeled as a sphere which can have at most six 2D-neighbors without overlapping (see Fig. 3). Each node is able to sense the direction of its physical neighbors [east (E), west (W), north-east (NE), south-east (SE), south-west (SW) and north-west (NW)]. The starting physical topology is a chain of n nodes linked together. Until now, the node can have at the beginning neighbors in directions SE or NW or in the both directions in the same time, we show in Sect. 6.3 how to generalize the algorithm. A node A is in neighbor's list of node B if A touches physically B, see Fig. 2. In the Claytronics project, communications are only possible through contact, meaning only neighbors can have direct communication.

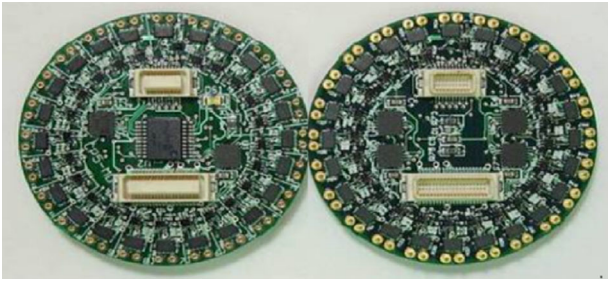

Fig. 1 Two Catoms

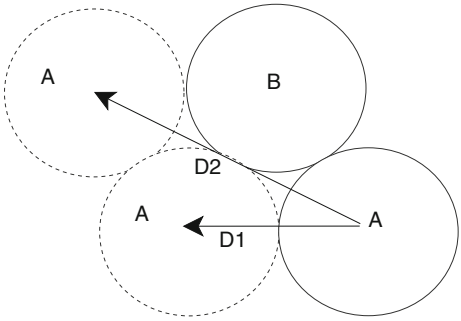

Fig. 2 Traveled distance in one movement = $2R$, the node A travels $2x\,R$ in one movement

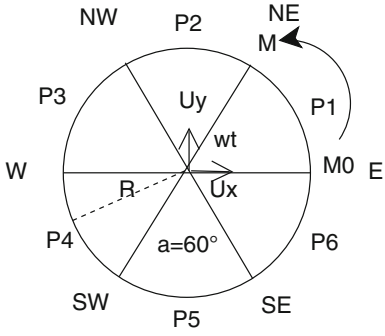

Fig. 3 Node modeling, in each movement the node travels the same distance

Consider the connected undirected graph $G = (V, E)$ modeling the network, $v \in V$, is a node belongs to the network and, $e \in E$ is a bidirectional edge of communication between two physical neighbors. For each node $v \in V$, we denote the set of neighbors of v as $N(v) = \{u, (u, v) \in E\}$. Each node $v \in V$ knows the set of its neighbors in G, denoted $N(v)$. We assemble our terminology as such:
Connectivity: in a graph $G = (V, E)$, if $\forall v \in V, \forall u \in V, \exists C_{v,u} \subseteq E : C_{v,u} = (e_{v,-}, \ldots, e_{-,-}, \ldots, e_{-,u})$, where $e_{x,y}$ is an edge from x to y and $C_{v,u}$ represents a path from v to u.
Snap-connectivity: let T be the total execution time of our distributed algorithm DA and $t_0, \ldots, t_m$ are the time slots of execution of DA. There is a snap-connectivity in DA with the dynamic graph $G_t(V_{t_i}, E_{t_i})$ the network state at the instant t_i, if $\forall t_i, i \in \{1, \ldots, m\}$, $G_{t_i}(V_{t_i}, E_{t_i})$ maintains the connectivity.
B-non-snap-connectivity: we say that DA guarantees a B-non-snap-connectivity if it ensures non-snap-connectivity at ti or at $t = t_i + \cdots + t_{m-1}$ and verifies the following conditions:

Let N_0 be the node's v set of neighbors at t_0, and let N_s be the node's v set of neighbors at t_s with $N_s = N_0$, and let $N_{0/s}$ be the node's v set of neighbors at t_0 or at ts and N_i its neighbors set at $t_i, i \neq 0$.

- If $r \in N_{0/s}$, and $r \notin N_i$ and $\forall e \in N_i, e \in N_{0/s}$ then $r \in N_{i+B}$ at $t_{(i+B)}$. Or
- if $r \in N_{0/s}$, and $r \notin N_i$ and $\exists e \in N_i, e \notin N_{0/s}$ then v is snap-connected with all its neighbors of $t_{(i+B+j)}$, $j = \{1, \ldots, m - i - B\}$. $Tree$: the tree is any connected

graph without cycles. In the tree, a node is a child or a parent. The leaf is a node without child

We call the *own number movements* of a given node the number of movements performed by it. In this paper, we present how the node can precalculate (predict) its own number movements, so it can make sure that it has correctly executed the algorithm.

To calculate the number of movements we agree on these proposals.

Consider Fig. 3 which represents a node microrobot. We say that a node has done a single movement if the distance between its old position and the new position is exactly twice the radius $D1 = 2R$. For example, if the node is in a position at a distance $D2$ from the old position it has done two movements (Fig. 2). Since 360° is divisible into six equal angles at 60° each, as the perimeter at an angle a is $P_a = \pi Ra/180$ and $P = 2\pi R$ and $P1 = P2 = P3 = P4 = P5 = P6$, the node can have without overlapping at most six neighbors and in each round the node travels the same distance.

Now, in a Cartesian plane, consider the curve of the following Cartesian parametric equation:

$$\begin{cases} x(t) = R\cos(wt) \\ y(t) = R\sin(wt) \\ \text{where } wt \in [0..2\pi[\end{cases} \tag{1}$$

With w a constant and the M point represents the contact point between the node in movement and the node around which it moves. This is a closed curve as $x(0) = x(2\pi/w)$ et $y(0) = y(2\pi/w)$. Noting that $x(t)^2 + y(t)^2 = R^2$. This means that M describes a circle with center O and radius R (like Fig. 3). If t means the time, the circle is described after a period $T = 2\pi/w$ which means the period of revolution. So, the velocity vector is written:

$$\vec{V} = \begin{pmatrix} -R\sin\ wt \\ R\cos\ wt \end{pmatrix} \tag{2}$$

The arc length traveled by M is $l(t) = \int || \vec{V} || \cdot \mathrm{d}t = Rwt$. So in one round the microrobot of radius R travels to Ra.

Message exchange between physical neighbors is carried without complexity, because the node can see directly the state of its physical neighbor. On the other side, if a node *to decide* needs to know the state of a non-physical neighbor this is carried through exchange of message since the node will wait to decide. For example, in Fig. 4:

- At t_0: Node A needs to know the state of B to move to the new position. This motion is done without exchanging of messages.
- At t_2: If A is in the new position and needs to know the state of D to move, D sends a message to C informing it of its state, after C forwards the message to A. There is a message exchange; A must wait two rounds for the input to decide.
- If at t_0 or at t_1 a message has been sent from D to C, A at t_2 can have the state of D with simple consultation of C's state, without message exchange.

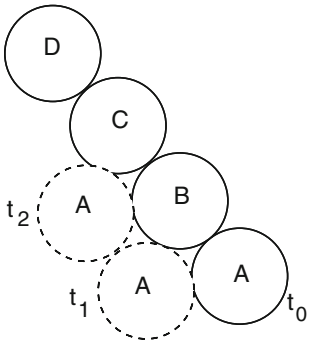

Fig. 4 Message transmission, there will be message exchange if the node needs to know the state of a non-neighbor node

5 Proposed protocols

In this section, we present the proposed self-reconfiguration algorithms. We start by presenting an algorithm named algorithm with unsafe connectivity (AUC), which ensures 1-non-snap-connectivity. After, we present an algorithm named algorithm with safe connectivity (ASC), which ensures a snap-connectivity.

5.1 Algorithm with unsafe connectivity (AUC)

In AUC, the node can move around its physical neighbor or move to a position with a distance equaling twice the radius from the original (old) position. These two kinds of motions are carried with the help of neighbors to learn the direction and the new position. The node that does not have a neighbor at the beginning cannot move, because it does not know where it is and where the other nodes are. Therefore, the algorithm should make sure not to lose nodes.

Description of the algorithm

The algorithm runs in rounds. In each round the demon (scheduler) chooses the verifiable predicates to run. We introduce in the algorithm a priority mechanism between predicates: the demon chooses predicate with the best priority and ignore in the current round the predicates with the lowest priority. The predicates labeled with P1 are predicates having the highest priority than those labeled by P2.

The distributed algorithm uses an incremental process. In a completed increment the nodes that build it belong already to the target shape. At the beginning, the initiator belongs to the target shape, so it helps its neighbors to take corrects positions. The nodes already in the form act as a landmark to the neighbor nodes to complete a new layer. The nodes already in the target shape change their states with predicates (1) and (5) and become constants. At the beginning, all nodes are initialized with the *bad* state (2) except the initiator (1), the node can check if its neighbor has the state *good* using predicate (5). With the predicate (12) if the node has *bad* state and it had in the NW direction a neighbor in the previous round, then it moves to the old position that was occupied by this former neighbor, it travels exactly $2R$.

The nodes of the current layer may move either at left directly or NW directly with the last three predicates. The node can change its state to *good* if it cannot move left or NW using the predefined movement predicates. With the instruction (13) the node moves to left around neighbors having the state *good*. It will have the neighbor that used it to move in the NE direction. This node repeats the same motion until it arrives at the diagonal node that has the state *spe*. It can move around this last only if the diagonal node does not have a node in the E direction. All diagonal nodes have the state *spe* with the predicate (7). And with (14) the node moves to NW until it takes a correct position.

Variables and predicates:

- $v, u1, u2$: variables denote a node belongs to the network.
- $\{u\}$: a set of nodes.
- $good, bad, spe$: states, a node can take one or two states at the same time, knowing that it cannot take the two states *spe* and *bad* or *good* and *bad* at the same time.
- $N_x(v)$: the neighbor in the direction x of the node v, with $x \in \{(N), (E), (W), (NE), (SE), or(NW)\}$.
- $thisRound$: integer denotes the current round.
- $connected_v$: $true$ if the node v is connected to the network, $false$ else (Boolean).
- $State_v(k)$: the state of the node v, taking one or two of these states $k \in \{good, bad, spe\}$.
- $N_x lastRound_v()$: the node v had a neighbor in the direction x in the previous round.
- $State_v(n, good)$: the node v has n neighbors that have the state $good$ $State(good)$.
- $cannotMove_v()$: the node v has one neighbor having the state $good$.
- $moveTo_v(P_{N_{nw}})$: move to the position $P_{V_{nw}}$ where was a neighbor N_{nw} in the direction nw in the previous round.
- $moveAround\ good\ _v(u, P_x)$: the node v moves around the neighbor u in such a way that u shifts in the direction x for v.

Predicates checked only in the first round

1: Initiator$(v) \equiv N_{nw}(v) = \oslash \wedge connected_v$.
2: State$_v(bad) \equiv connected_v \wedge \neg Initiator(v)$.
3: State$_v(good) \equiv Initiator(v)$.
4: State$_v(spe) \equiv Initiator(v)$.

Predicates checked in each round

5:(P1): State$_v(good) \equiv (N_e(v) = u1 \wedge State_{u1}(good) \wedge N_{ne}(u1) = \oslash) \vee State_v(3, good)$
$\vee (State_v(2, good) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(spe)) \vee (N_w(v) = u1 \wedge State_{u1}(good)))$
$\vee State_v(spe)$.
6: State$_v(n, good) \equiv (N_x(v) = \{u\}, |u| = n \wedge State_{\{u\}}(good))$.
7: (P1): State$_v(spe) \equiv (N_{nw}(v) = u1)) \wedge (N_{ne}(v) = u2, State_{u2}(spe))$.
8: thisRound$\equiv GetCurrentRound()$.
9: has$N_{nw}v(thisRound) \equiv N_{nw}(v) = u1 \wedge State_{u1}(bad)$.
10: N_{nw}lastRound$_v(LastRound) \equiv has N_{nw}v(thisRound - 1)$.
11: (P2): moveTo$_v(P_{N_{nw}}) \equiv State_v(bad) \wedge N_{nw} lastRound_v$
$(LastRound) \wedge \neg has N_{nw}v(thisRound)$.
12: cannotMove$_v() \equiv (N_x(v) = \{u\}, |u| = 1 \wedge State_u(good))$.
13: (P2): moveAround$good_v(u1, P_{ne}) \equiv \neg(cannotMove_v()) \wedge (State_v(bad)) \wedge (N_{nw}(v)$
$= u1 \wedge State_{u1}(good))$.
14:(P2): moveAround$good_v(u1, P_e) \equiv \neg(cannotMove_v()) \wedge (State_v(bad)) \wedge (N_{ne}(v)$
$= u1 \wedge State_{u1}(good))$.

Algorithm 1: the AUC Algorithm.

State change has priority as the moving actions to avoid bad motion, because when the node is in a good position (can change its state to *good*) it should ignore the predicate of motion. For this purpose, we use a priority mechanism in our algorithm. To avoid message exchange the node can change its state to *good* if it has 3 *good* neighbors or one neighbor has *spe* state and has neighbors in both directions, NE and NW (6).

The algorithm guarantees 1-non-snap-connectivity

This means, there is no message following a path $C_{v,u}$ that it cannot be transmitted (must wait on the same node) to another new node for two or more than two consecutive rounds. An algorithm ensures 1-Non-snap-connectivity if it does not ensure a snap-connectivity and it is not a *B-non-snap-connectivity*, with $B > 1$.

Algorithm with unsafe connectivity does not ensure a snap-connectivity because, for instance, in the second round the neighbor node of the initiator moves around the initiator using the instruction $moveAroundgood_v(u1, P_e)$, it will have a single neighbor in the direction E leaving a gap where it had two neighbors. If we assume that the message has arrived at another former neighbor, the message will wait for another round or more so that it is transmittable to the initiator, the equivalent of which $\exists v \in V, \exists u \in V \exists t_i$, where $\nexists C_{v,u}$.

We show by induction that AUC does not guarantee a *2-non-snap-connectivity*. We assume that AUC ensures a *2-non-snap-connectivity* namely $\exists v \in V, \exists u \in V \exists t_i, \exists t_{i+1}$, where $\nexists C_{v,u}$, when the message should wait two consecutive rounds. Assume that the message is waiting at the end of t_i. At this time the predicate $moveTo_v(P_{N_{nw}})$ is available for the node having the message, and as the demon is equitable this node will run this predicate because the predicate $State_v(good)$ is not verifiable, so it moves at the position of its last neighbor and it will find a new neighbor because this last cannot move ($cannotMove_v() \equiv (N_x(v))$. Therefore, at t_{i+1} the message can be transmitted to another node. Since AUC is not a *2-non-snap-connectivity* it is not *B-non-snap-connectivity*, $B > 2$.

To complete the proof that AUC guarantees a *1-non-snap-connectivity*, it remains to show for the node where the message was blocked at the time t_i that this node is Snap-connected with its neighbors that had at t_{i+1+j}, $j = \{1, \ldots, n - i - 1\}$. This can be proven through reverse since the node will move with the predicate $moveAroundgood_v(u1, P_e)$ around nodes having a *good* state and have neighbors having a *good* state. Because these nodes cannot move ($State_v(good)$), the message can be transmitted at each time from or to neighboring nodes of t_{i+1+j}, $j = \{1, \ldots, n - i - 1\}$.

Predicting the number of movements

To form the matrix of our square of $N \times N$, we begin with an incremental process with a single node that we assume in a correct square 1×1. Then, we add each time a new layer that contains the number of nodes of the last column plus the number of nodes of the last line of the current square plus one node. For example, to reach the square 2×2 we have to add a new layer with three nodes. Consider Fig. 5, the node i will take a place $p + x$. Following the path from top to bottom the node i will always move before the other nodes. If node A is before node C, then A will take a place $p + c$, while C will take a place $p + k$, with $k > c$. Each time, we add a new layer

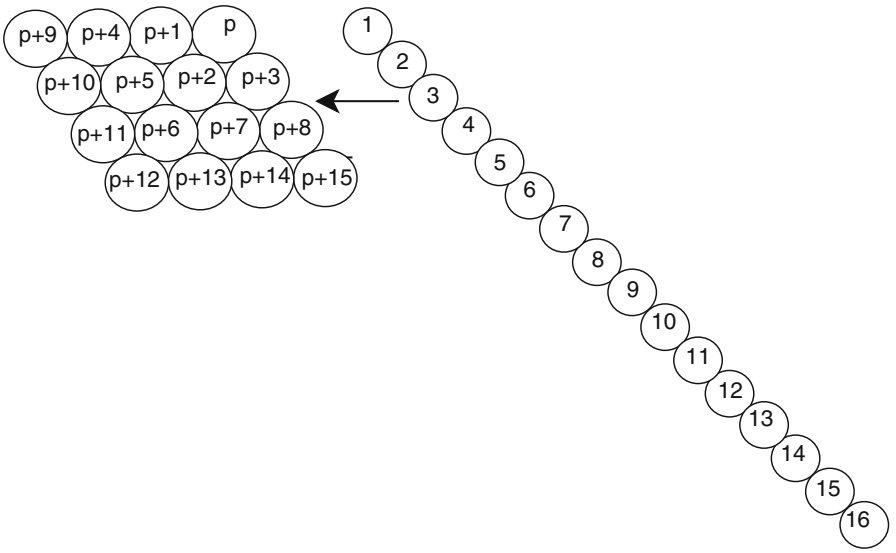

Fig. 5 Nodes positioning into the final square shape

with a number of nodes equal to the number of nodes in the previous layer plus two nodes; this is expressed on the form of this numerical arithmetic sequence:

$$U_j = U_{j-1} + 2. \tag{3}$$

where, U_j is the number of nodes in the layer j and U_{j-1} is the number of nodes in layer $j-1$.

In the chain we take a partitioning of nodes into levels. A level is associated to one or more nodes. The nodes take their levels through the following process: the root level (level 0) is associated to the first node (the root), for the other nodes, each level is associated to a number of nodes equal to the number of nodes of the previous level plus 2 (Fig. 6 shows an example). So each node i gets one level at the end. To calculate the number of movements, we use another sequence similar to Eq. (3) though with a different interpretation:

$$\begin{aligned} S_1 &= 2. \\ S_j &= S_{j-1} + 2. \end{aligned} \tag{4}$$

with S_j as a number associated to nodes that have the level j.

The number of movements for each node i having the level j can be given with the composition of two sequences $U_{i,j}$ and S_j,

$$\begin{aligned} U_1 &= 0. \\ U_{i,j} &= U_{j-1} + S_j. \end{aligned} \tag{5}$$

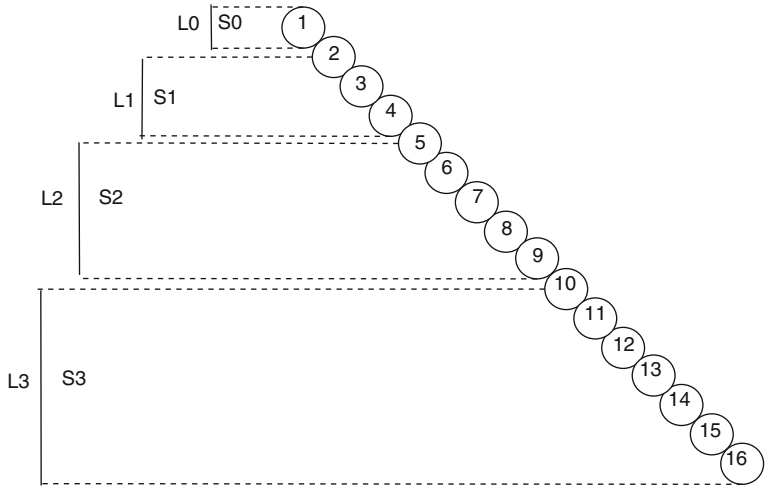

Fig. 6 Nodes partitioning into levels

where $U_{i,j}$ and $U_j$is the number of movements of node i of level j, or the number of movements of nodes that have the level j.

Theorem 1 *Let n be the network size.* $n - \sqrt{n}$ *is the highest number of movements in this algorithm, where* $\sqrt{n}$ *is an integer number. Special case: if* $\sqrt{n}$ *is not an integer number, the highest number of movements is* $\lceil\sqrt{n}\rceil\lceil\sqrt{n}\rceil - \lceil\sqrt{n}\rceil$*, and the number of the own movements is given with the same sequences.*

5.2 Algorithm with safe connectivity (ASC)

In this algorithm, each node can move only around its physical neighbor. To ensure a snap-connectivity only nodes that do not cause network disconnectivity can move around neighbors. For this purpose, we introduce the use of the tree to dynamically manage the leaf nodes that can move.

Description of the algorithm

Unlike AUC, to ensure a snap-connectivity in ASC, only the leaf nodes can move. For this purpose, we introduce the use of the tree to dynamically manage the leaf nodes that can move. The algorithm runs in rounds, in each round satisfied predicates are chosen to run. We introduce a priority mechanism between predicates, in a current round predicates with the best priority are chosen to run and others with lowest priority are ignored. We notice, in ASC predicates labeled with P1 are considered more prior than those labeled by P2.

Algorithm with safe connectivity seeks the desired form using an incremental process. In a completed increment the nodes that build it belong already to the form. The initiator which is the root initializes the tree and becomes a parent of itself (5), a node if does not have a parent becomes a child of one of the neighbor parents (6). A node is a leaf if all its neighbors are parents (7). At the beginning all nodes are

Variables and predicates:

- $Parent(v, u)$: means the node v is parent of node u.
- $isLeaf(v)$: the node v is a leaf in the tree.

Predicates checked only in the first round

1: Initiator$(v) \equiv N_{nw}(v) = \oslash \wedge connected_v$.
2: State$_v(bad) \equiv connected_v \wedge \neg Initiator(v)$.
3: State$_v(good) \equiv Initiator(v)$.
4: State$_v(spe) \equiv Initiator(v)$.

Predicates checked in each round

5: Parent$(v, v) \equiv Initiator$(v).
6: Parent$(v, u) \equiv (Parent(w, v), u \neq w) \wedge neighbor(v, u) \wedge State_u(bad) \wedge (\nexists z \in N(v), Parent(v, z))$.
7: isLeaf$(v) \equiv (\forall u \in N(v), \neg Parent(v, u) \wedge \neg Parent(v, v))$.
8: (P1): State$_v(good) \equiv (N_e(v) = u \wedge State_u(good) \wedge N_{ne}(u) = \oslash) \vee State_v(3, good) \vee (State_v(2, good) \wedge (N_{ne}(v) = u \wedge State_u(spe)) \vee (N_w(v) = u \wedge State_u(good))) \vee State_v(spe)$.
9: State$_v(n, good) \equiv (N_x(v) = \{u\}, |u| = n \wedge State_{\{u\}}(good))$.
10: (P1): State$_v(spe) \equiv (N_{nw}(v) = u1)) \wedge (N_{ne}(v) = u2, State_{u2}(spe))$.
11: (P2): moveAround$bad_v(u1, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u1 \wedge State_{u1}(bad))$.
12: (P2): moveAround$bad_v(u1, P_{se}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(bad))$.
13: (P2): moveAround$good_v(u1, P_{ne}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u1 \wedge State_{u1}(good))$.
14: (P2): moveAround$good_v(u1, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(good))$.

Algorithm 2: the ASC Algorithm.

initialized with the *bad* sate with predicate (2). The initiator belongs to the target shape, so it changes its state to *good* (3), it will help its neighbors or future neighbors to take correct positions. The nodes already in the target shape act as a landmark to neighbor or future neighbor nodes to complete a new layer. The nodes already in the form change their states with predicates (3) and (8) and become constants. The node can check if its neighbors have the *good* state with predicates (3) and (8). The node that starts the move is the lowest node in the chain, which is the leaf of the first tree built, it rises until the root using motion around other nodes with predicates (11) and (12). The nodes of the current layer (layer being built) may make motion either at left directly or NW directly with the last three predicates. The node can change its state to *good* with (3) if it cannot move to left or in NW. With the predicate (13) the node moves at left, it will have the neighbor that used it to move at NE direction, it repeats the same motion until it arrives at the diagonal node that have the state *spe*, it cannot move around this last only if the diagonal node has not a neighbor node in the E direction. Diagonal nodes take the state *spe* with predicates (4) and (10). With (14) the node moves until it takes a correct position. The state change has a priority as the moving actions to avoid bad motion, this why we introduce the priority in our algorithm. To avoid message exchange the node can change its state to *good* if it

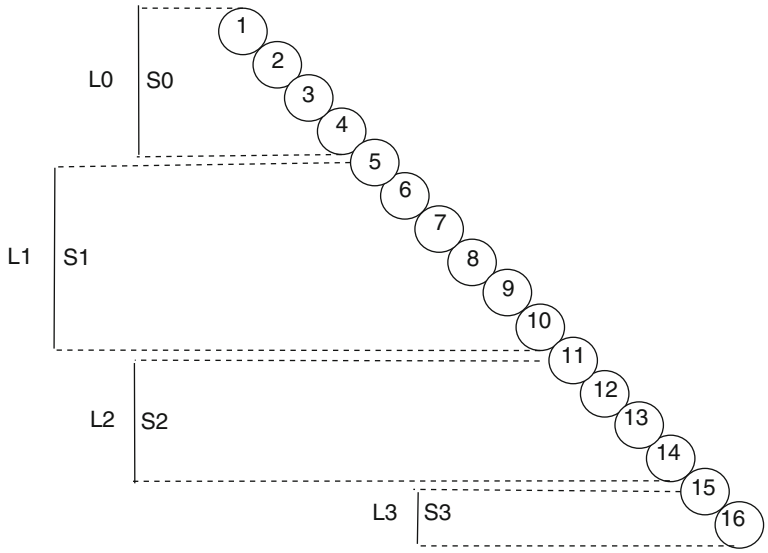

Fig. 7 Nodes partitioning into levels

has three neighbors having the state *good* (9) or one neighbor has *spe* state and has neighbors in the both NE and NW directions with predicate (10).

The algorithm guarantees a snap-connectivity

This algorithm guarantees a snap-connectivity because it uses a tree where only the leaves can move. The mobile leaves in the algorithm are unable to gap in the structure since they are surrounded by the neighbors which allow an interlocking network of safeties. We divide the leaves movements into two families: (1) a node moves around another without having a new neighbor to replace it. In this case there is no ti since the message, exchanged through the movement of the neighboring node, cannot be sent. (2) A node that has moved receives a neighbor after moving position to replace it and becomes a neighboring node with another. This means another route for the message of $C_{v,u}$ which will not be blocked for all ti, $i \in \{1, \ldots, n\}$.

Predicting the number of movements

As in AUC, to form the shape we use the same principle of increments. Examining the Fig. 5, the node i will take a position $p + x$. Unlike AUC, the node i will either remain stationary or move only after all nodes succeeding it. This means if A precedes B in the same layer, A takes the position $p + c$, while B takes the position $p + k$, with $c > k$. As in Eq. 3, the number of nodes added in each layer can be expressed with an arithmetic sequence.

In the chain we take a partitioning of nodes into levels. A level is associated to one or more nodes: the first nodes that have $i \leq \sqrt{n}$ take the level 0 (level 0), the first $x = (2\sqrt{n} - 2)$ nodes after the node $i = \sqrt{n}$ take the level 1, and the second $x - 2$ nodes take the next level and so on by subtracting each time 2 from the last x, this is well illustrated in Fig. 7. So each node i gets one level at the end.

If the number of nodes n is a square of an integer number, the number of movements for each node i of level j can be given with the composition of two sequences $U_{i,j}$ and S_j.

$$S_j = \begin{cases} 0, & \text{if } j = 0 \\ 2\sqrt{n} - 5, & \text{if } j = 1 \\ S_{j-1} - 2, & \text{otherwise} \end{cases} \tag{6}$$

with S_j as a number associated to nodes that have the level j.

$$U_{i,j} = \begin{cases} 0, & \text{if } j = O. \\ 2, & \text{if } i = \sqrt{n} + 1,\ j = 1. \\ U_{i-1} - S_j, & \text{if } l(i-1) \neq j. \\ U_{i-1} + 2, & \text{otherwise,} \end{cases} \tag{7}$$

where $U_{i,j}$ and U_j is the number movements of node i of level j or the number of movements of nodes that have the level j.

Theorem 2 *Let n be the network size. n is the highest number of movements in this algorithm.*

Special case If the number of nodes n is not a square of an integer number. To calculate the number of movements of a given node, we consider a similar partitioning system as before. The highest number of movements to reach the final form remains n.

Let $q = \lfloor \sqrt{n} \rfloor$, and $\text{diff} = n - q^2$.

$$U_{i,j} = \begin{cases} 0, & \text{if } i \leq q. \\ \text{diff} + 2q - 1, & \text{if } i = q + 1. \\ U_{i-1} - 2, & \text{if } q + \text{diff} \geq i > q \\ \text{diff} + 2, & \text{if } i = q + \text{diff} + 1. \\ U_{i-1} - S_j, & \text{if } i > q + \text{diff}, L(i+1) \neq j. \\ U_{i-1} + 2, & \text{otherwise.} \end{cases} \tag{8}$$

6 Analysis and evaluation

In this section, we discuss and analyze the number of states required, the generalization of the algorithms and the simulation results.

6.1 The three-state minimum

In this section, we prove that three states are necessary to achieve AUC and ASC.

With just a single state it is impossible for the nodes to self-configure since they cannot distinguish whether they are in a good position or not. Furthermore, there is no

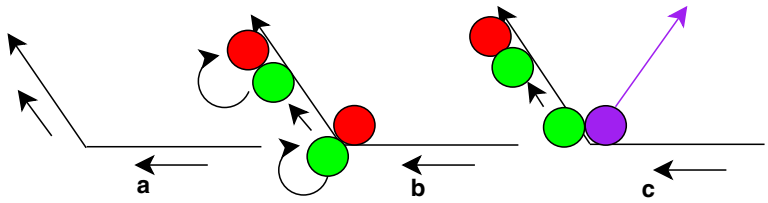

Fig. 8 States of nodes, three states are required for each node

mechanism to differentiate nodes that we wish to do the motion and those we wish to rest stable. This prevents us from knowing a correct movement from an incorrect one. Suppose we seek an algorithm with two states, knowing there are two types of motion possible; with two states the node knows whether it is in a good position or not and will adjust itself accordingly in the aforementioned left or NW around other nodes in correct positioning as well (Fig. 8a). However, in the early rounds the nodes that form in the diagonals of the square can move around nodes that converted themselves to the correct state through moving in the direction NW (Fig. 8b), thus creating another chain. While a node may change its state if it has three $good$ neighbors, but if a node sits west of the diagonal node cannot move NW and has only two neighbors it must change its state to $good$. There is no benchmark to distinguish these situations unless adding a special state to nodes forming the diagonal. The third state is essential to differentiate the node's ability to move around another having spe state if and only if it has no right neighbor (Fig. 8c).

6.2 Complexity of messages sent

The most interesting action for exchanging of messages in the algorithms is one that is activated by predicates of state-changing. If a node changes its state before it is sure of the appropriate state of other nodes that have moved before it in the current layer, the process will completely go in the opposite direction of the desired objective. The predicate $State_v(good)$ ensures without changing messages that the node changes its state only if all nodes that have moved before it have changed their state to $good$. The first node that begins the construction of the new layer does not need to wait for the message from the first node of the previous layer, since it has this information by simply consulting (message) the state of its former neighbors. In other words, the message was being sent before the node knew the state of its sender. The node will find the information of the state of its neighbors simply on a need-to-know basis. This means we do not need to transmit information between two non-neighboring nodes of different layers or between two neighboring nodes of the same layer. This efficiency is explained by the fact that AUC and ASC make the synchronization in state changing is not required for nodes that are in the same layer. We note that ASC needs $O(n)$ messages of the tree.

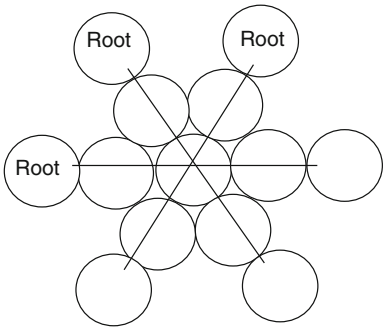

Fig. 9 The three possible cases of a straight chain

6.3 Generalization of the algorithms

We have presented two distributed algorithms that deal with the case where the chain is oriented in a NW–SE direction. To show how to generalize the algorithms in order to deal with any chain orientation, we start by explaining how the initiator is selected in the general case. The other nodes can determine the orientation of the initial chain by looking at the direction of their two initial neighbors. The root is designated as the node with only one neighbor in the direction SW, SE, or E. Whatever the orientation of the chain, only one node corresponds to this condition (Fig. 9). Every node in the chain can deduce the orientation of the chain (among the three cases represented in Fig. 9) by analyzing the orientation of its neighbors. For example, if a node corresponds to an extremity node (with one neighbor) where the direct neighbor is on the E side, the node deduces that the straight line is oriented E–W. The same thing happened on the middle nodes, which use the orientation of their two neighbors to determine the orientation of the formed chain.

After the detection of the chain orientation, noted $D - \overline{D}$, every node runs a variant of the ASC or AUC algorithm depending on the orientation $D \in \{\text{W,NW,NE}\}$. The variant of AUC or ASC algorithm, called AUC^D or ASC^D, represents an adaptation of the original AUC and ASC algorithms (corresponding to AUC^{NW} and ASC^{NW}) to the two other possible orientations (W and NE). For instance, if the initial chain is oriented NE–SW, the algorithm ASC^{NE} is called, and the square form is realized using moves of type $moveAroundbad_v(u1, P_w)$, $moveAroundwell_v(u1, P_w)$ and $moveAroundwell_v(u1, P_{nw})$. The usage of these three predicates is described in Fig. 10.

6.4 Simulation and comparison

We have done the simulation with both the Meld [1] and Dprsim [43] tools. Dprsim simulates real movement using magnet forces taking into the gravity. Meld is a declarative language which, using Dprsim, simulates with the predicates algorithms where the type of movement necessary can be completed only around the physical neighbors such as ASC. Figure 11 represents an instance of execution of AUC and Fig. 12 represents

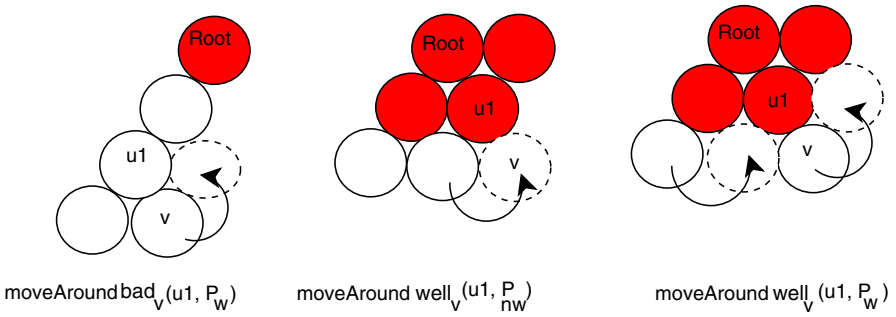

Fig. 10 Moves adaptation in the case of NE–SW chain. *Dark nodes* represent the well-state node

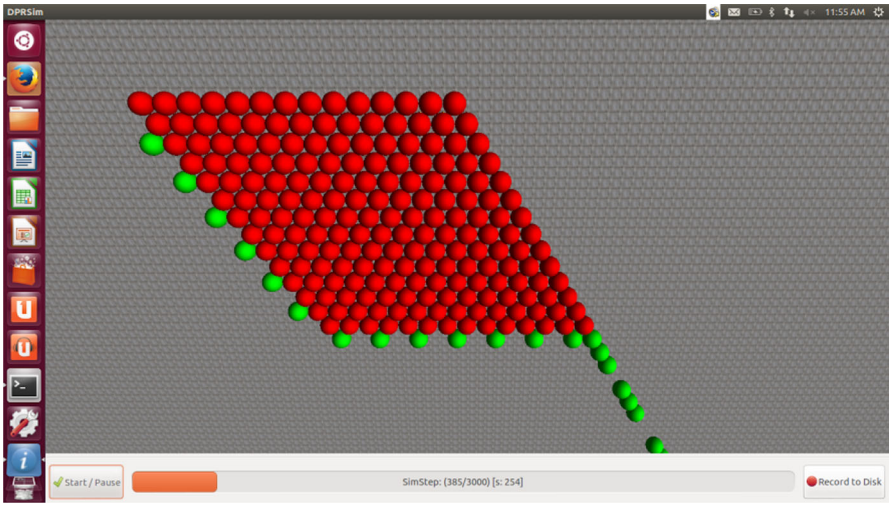

Fig. 11 An instance of execution of AUC, nodes *green* colored are nodes having the state *bad* and nodes *red* colored are nodes having the state *well* (color figure online)

an instance of execution of ASC. In our simulations the radius of the node is 1 mm[1]. We have used an Intel(R) Core(Tm)i5, 2.53 Ghz laptop with 4 GB of memory. The results of these simulations come to agree with the result obtained previously, in particular regarding the number of movements and the own movements. The nodes applied the procedure of nodes partitioning to levels and predicted with the sequences the number of movements for each node, at the end of the algorithm each node compares the results of prediction to the results calculated by it. Figure 14 represents the number of movements by number of nodes, with $f(n) = (\lceil\sqrt{(n)}\rceil\lceil\sqrt{(n)}\rceil) - \lceil\sqrt{(n)}\rceil$, $g(n) = n - \sqrt{(n)}$, and $h(n) = n$. Figure 13 represents the execution time in ticks by the number of nodes.

For the curves that represent the movements, we denote some value of the network size as m. If m is a square root than the number of movements of AUC is always less

[1] The time of one movement depends on the size (the diameter) of the microrobot, as shown in Sect. 4.

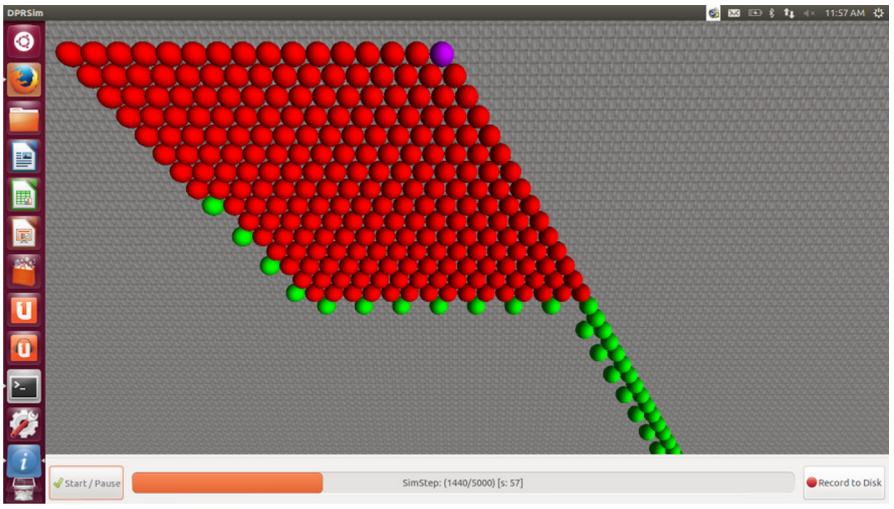

Fig. 12 An instance of execution of ASC, nodes *green* colored are nodes having the state *bad* and nodes *red* colored are nodes having the state *well*, the node *pink* colored is the initiator (color figure online)

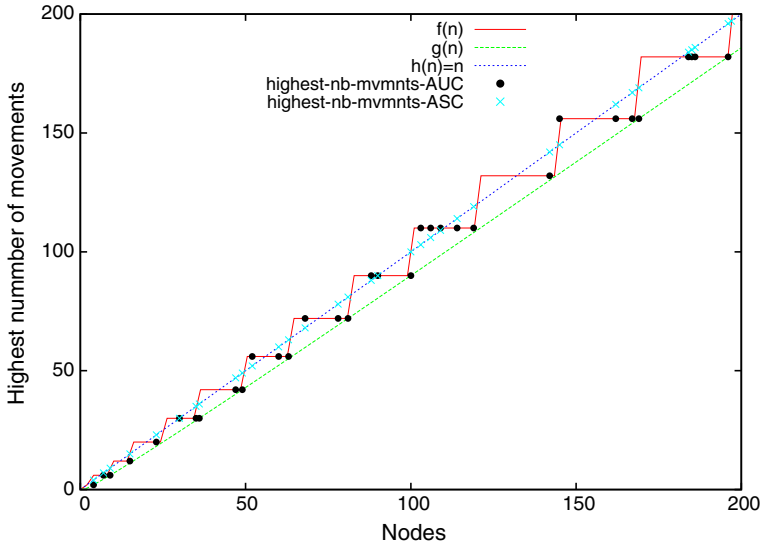

Fig. 13 The highest number of movements of AUC and ASC

than that of ASC, but if m is not a square root then we distinguish two cases: if m is not a square root then there is a number Ms that is the minimum number superior to m with Ms as square root. There is also a number Mi, the number maximum inferior to m with Mi as square root. In this curve, until 50 nodes for some values of m, we see the number of movements of AUC are lower than that of ASC knowing that in these cases m is closer to Ms. However, if m is closer to Mi we see the number of movements of ASC are lower than that of AUC for the values from $m = 100$. Note

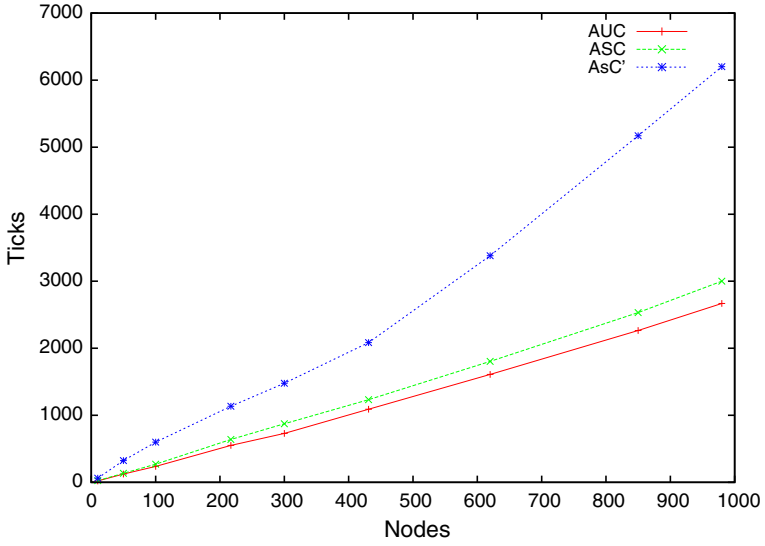

Fig. 14 Execution time comparison between ASC and AUC

also that if m is in the midst of Mi and Ms as the cases of 30 and 90 nodes, the two algorithms have the same number of movements. For the curves that represent the execution time, without counting the time of construction of the tree for ASC we see that if the number of nodes increases the difference between the two algorithms increases with almost constant acceleration, but if we count the time of the tree ($O(m)$ time) the difference will be very high.

7 Conclusion

In this paper, we have shown the self-reconfiguration possibility without a map of the target shape. We proposed an energy and memory-efficient distributed self-reconfiguration for modular microrobots. We presented a new method to complete the self-reconfiguration where the nodes do not know the fixed positions of the target form but only the aimed shape. We have studied two types of movements where the node can receive help to create the motion and have the exact positions desired.

Both proposed algorithms are characterized by a linear time complexity regarding the system size (number of microrobots) with constant memory needs. Message exchanges are limited to neighboring consultations; consequently, system reconfiguration is fast.

However, some open problems remain. The derivation of a fault-tolerant algorithm from the principle of this paper which guarantees the data items delivery to non-faulty nodes needs to be investigated. We also study the conception of an energy-efficient algorithm when the starting form may be any connected shape. We predict the loss of these previous characteristics found in this paper, in particular the number of states of each node and the message exchange. In our algorithm with safe connectivity, the

number of own movements is calculated (predicted) according to the network size. The question is then to know if it is possible to calculate the own movements without knowing the network size and if using messages would be required.

Acknowledgments This work is supported by the Labex ACTION program (contract ANR-11-LABX-01-01), ANR/RGC (contracts ANR-12-IS02-0004-01 and 3-ZG1F) and ANR (contract ANR-2011-BS03-005). The authors wish to express their appreciation to the anonymous reviewers for their constructive comments.

References

1. Ashley-Rollman MP, Goldstein SC, Lee P, Mowry TC, Pillai P (2007) Meld: a declarative approach to programming ensembles. In: Proceedings of the IEEE international conference on intelligent robots and systems (IROS '07)
2. Ashley-Rollman MP, Lee P, Goldstein SC, Pillai P, Campbell JD (2009) A language for large ensembles of independently executing nodes. In: Proceedings of the international conference on logic programming (ICLP '09)
3. Bourgeois J, Goldstein SC (2012) Distributed intelligent MEMS: progresses and perspectives. Thirrd international conference on ICT innovations, Advances in Intelligent and Soft Computing, Ohrid, pp 15–25
4. Bojinov H, Casal A, Hogg T (2000) Emergent structures in modular self-reconfigurable robots. In: Proceedings of the IEEE international conference on robotics and automation, vol 2. IEEE Computer Society Press, Los Alamitos, pp 1734–1741
5. Bulusu N, Heidemann J, Estrin D (2000) GPS-less low-cost outdoor localization for very small devices. IEEE Pers Commun Mag 7:28–34
6. Butler ZJ, Kotay K, Rus D, Tomita K (2004) Generic decentralized control for lattice-based self-reconfigurable robots. Int J Robot Res 23(9):919–937
7. Dewey D, Srinivasa SS, Ashley-Rollman MP, Rosa MD, Pillai P, Mowry TC, Campbell JD, Goldstein SC (2008) Generalizing metamodules to simplify planning in modular robotic systems. In: Proceedings of IEEE/RSJ 2008 international conference on intelligent robots and systems (IROS '08)
8. Funiak S, Pillai P, Ashley-Rollman MP, Campbell JD, Goldstein SC (2008) Distributed localization of modular robot ensembles. In: Proceedings of robotics science and systems
9. Jones C, Mataric MJ (2003) From local to global behavior in intelligent self-assembly. In: Proceedings of the 2003 IEEE international conference on robotics and automation, ICRAM 2003, vol 1. IEEE Computer Society Press, Los Alamitos, pp 721–726
10. Jeon S, Ji C (2008) Randomized distributed configuration management of wireless networks: multi-layer Markov random fields and near-optimality CoRR abs/0809.1916
11. Hollar S, Flynn A, Bellew C, Pister KSJ (2003) Solar powered 10 mg silicon robot. In: MEMS, Kyoto
12. Karagozler ME, Thaker A, Goldstein SC, Ricketts DS (2011) Electrostatic actuation and control of micro robots using a post-processed high-voltage SOI CMOS chip. IEEE international symposium on circuits and systems (ISCAS)
13. Kotay K, Rus D, Vona M, McGray C (1998) The self-reconfiguring robotic molecule. In: Proceedings of the 1998 IEEE international conference on robotics and automation, Leuven
14. Kribi F, Minet P, Laouiti A (2009) Redeploying mobile wireless sensor networks with virtual forces. IFIP Wireless Days, Paris
15. Lakhlef H, Mabed H, Bourgeois J (2013) Distributed and efficient algorithm for self-reconfiguration of MEMS microrobots. In: The 28th ACM symposium on applied computing, Coimbra
16. Lakhlef H, Mabed H, Bourgeois J (2013) Distributed and dynamic map-less self-reconfiguration for microrobot networks. In: IEEE NCA 2013, 12th IEEE international symposium on network computing and applications, Cambridge, pp 55–60
17. Lakhlef H, Mabed H, Bourgeois J (2013) Parallel self-reconfiguration for MEMS Microrobot. In: Seventh IEEE international conference on computer as a tool, Zagreb, pp 283–290
18. Lakhlef H, Mabed H, Bourgeois J (2013) Dynamicity to save energy in microrobots reconfiguration. In: Tenth IEEE international conference on ubiquitous intelligence and computing (UIC-2013), Italy, pp 246–253

19. Lakhlef H et al (2014) Optimization of the logical topology for mobile MEMS networks. J Netw Comput Appl. doi:10.1016/j.jnca.2014.02.014
20. Liu L, Antonopoulos N, Mackin S (2008) Managing peer-to-peer networks with human tactics in social interactions. J Supercomput 44(3):217–236
21. Liu L, Xu J, Russell D, Antonopoulos N (2008) Self-organization of autonomous peers with human strategies. Third international conference on internet and web applications and services, pp 348–357
22. Microrobot design using fiber reinforced composites. J Mech Design 130(5)
23. Mamei M, Roli A, Zambonelli F (2004) Emergence and control of macro spatial structures in perturbed cellular automata, and implications for pervasive computing systems, IEEE transactions on systems, man, and cybernetics, vol 36, no 5; 2005. J Appl Artif Intell 8(9–10):903–919
24. Minet P, Mahfoudh S (December 2009) Energy, bandwidth and time efficiency in data gathering applications. IFIP Wireless Days, Paris
25. Moses R, Krishnamurthy D, Patterson R (2003) A self-localization method for wireless sensor networks. Eur J Appl Signal Process 4:348–358
26. Petrina AM (2011) Advances in robotics (review). Autom Doc Math Linguist 45(2):43
27. Ravichandran R, Gordon G, Goldstein SC (2007) A scalable distributed algorithm for shape transformation in multi-robot systems. In: Proceedings of the IEEE international conference on intelligent robots and systems (IROS '07)
28. Rosa MD, Goldstein SC, Lee P, Campbell JD, Pillai P (2008) Programming modular robots with locally distributed predicates. In: Proceedings of the IEEE international conference on robotics and automation (ICRA '08)
29. Rus D, Vona M (2001) Crystalline robots: self-reconfiguration with compressible unit modules. Auton Robots 10(1):107–124
30. Soua R, Saidane L, Minet P (April 2010) Sensors deployment enhancement by a mobile robot in wireless sensor networks. IEEE ICN 2010, Les Menuires
31. Stoy K, Nagpal R (2004) Self-reconfiguration using directed growth. Seventh international symposium on distributed autonomous robotic systems (DARs), France
32. Spears W, Spears D, Hamann J, Heil R (2004) Distributed, physics-based control of swarms of vehicles. Auton Robots 17(2–3):137–162
33. Stoy K, Nagpal R (2004) Self-repair through scale independent self-reconfiguration. In: Proceedings of 2004 IEEE/RSJ international conference on intelligent robots and systems, Sendai
34. Shen W, Will P, Galstyan A (2004) Hormone-inspired self-organization and distributed control of robotic swarms. Auton Robots 17(1):93–105
35. Sty K (2006) Using cellular automata and gradients to control self-reconfiguration. Robot Auton Syst 54(2):135–141
36. Walter J, Tsai B, Amato N (2005) Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. IEEE Trans Robot 21(4):621–631
37. Walter J, Welch J, Amato N (2004) Distributed reconfiguration of metamorphic robot chains. Springer Verlag J Distrib Comput 17:171–189
38. Wong S, Walter J (2013) Deterministic distributed algorithm for self-reconfiguration of modular robots from arbitrary to straight chain configurations. The IEEE international conference on robotics and automation (ICRA 2013)
39. Ward A, Jones A, Hopper A (2002) A new location technique for the active office. IEEE Pers Commun Mag 4:42–47
40. Warneke B, Last M, Leibowitz B, Pister KSJ (2001) Smart dust: communicating with a cubic-millimeter. Computer Magazine, pp 44–51
41. White P, Zykov V, Bongard JC, Lipson H (2005) Three dimensional stochastic reconfiguration of modular robots. In: Proceedings of robotics science and systems. MIT Press, Cambridge, pp 161–168
42. Zambonelli F, Gleizes MP, Mamei M, Tolksdorf R (2005) Spray computers: explorations in self-organization. J Pervas Mobile Comput Elsevier 1:1–20
43. The physical rendering simulator (dprsim). http://www.pittsburgh.intel-research.net/dprweb