Parallel Self-reconfiguration for MEMS Microrobots

Hicham Lakhlef, Hakim Mabed, and Julien Bourgeois
FEMTO-ST/DISC, University of Franche Comte, 1 cours leprince Ringuet, 25201, Montbeliard, France
$\{hlakhlef, hmabed, julien.bourgeois\}$ $@femto-st.fr$

***Abstract*—MEMS microrobots are low-power and low-memory capacity devices that can sense and act. One of the most difficult challenges in MEMS microrobot applications is the self-reconfiguration, especially when the efficiency, parallelism and the scalability of the algorithm are required. Self-reconfiguration with shared map does not scale. Because the map (predefined position of the target shape) consists of P positions, and each node must have a memory capacity, at least, of P positions. Therefore, if P is very high, the self-reconfiguration will be not feasible. In this paper, we present an efficient reconfiguration algorithm, without predefined positions of the target shape, which reduces memory usage to $O(1)$. This algorithm ensures the networks connectivity throughout all its execution time. This solution improves the execution time and the number of movements by using movement of different microrobots at the same time that we call parallelism of movement. Our algorithm is implemented in Meld, a declarative language, and executed in a real environment simulator called DPRSim.**

***Index Terms*—MEMS microrobot; distributed algorithm; self-reconfiguration; parallelism; logical topology; energy;**

I. Introduction

Micro electro mechanical system (MEMS) is a technology that enables the batch fabrication of miniature mechanical structures, devices, and systems. MEMS are miniaturized and low-power devices that can sense and act. It is expected that these small devices, referred to as MEMS nodes, will be mass-produced, making their production cost almost negligible. Their applications require a massive deployment of nodes, thousands or even millions [24], [7] which will give birth to the concept of Distributed Intelligent MEMS (DiMEMS) [3]. The size of MEMS nodes can vary from well below one micron on the lower end of the dimensional spectrum, all the way to several millimeters. A DiMEMS device is composed of typically thousands or even millions of MEMS nodes. Some DiMEMS devices are composed of mobile MEMS nodes [1], some others are partially mobile [29] whereas other are not mobile at all [3]. Due to their small size and the batch-fabrication process, MEMS microrobots are potentially very cheap, particularly through their use in many areas in our lifetime [27], [11]. One of the major challenges in developing a microrobot is to achieve a precise movement to reach the destination position while using a very limited power supply. Many different solutions have been studied for example, within the $Claytronics$ project [1], [2], [8], [18], [12] each microrobot can only turn around its neighbor which introduce the idea of a collaborative way of moving. But, even if the power requested for moving has been lowered, it still costs a lot regarding the communication and computation requirements. Optimizing the number of movements of microrobots is therefore crucial in order to save energy.

In the literature, the self-reconfiguration can be seen from two different points of view. First, it can be defined as a protocol, centralized or distributed, which transforms a set of nodes to reach the optimal logical topology from a physical topology [10]. On the other hand, the self-reconfiguration is built from modules which are autonomously able to change the way they are connected, thus changing the overall shape of the network [8], [22]. This process is difficult to control, because it involves the distributed coordination of a large numbers of identical modules connected in time-varying ways. The range of exchanged information and the amount of displacement, determine the communication and energy complexity of the distributed algorithm. When the information exchange involves close neighbors, 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.

II. Related Works

Many terms refer to the concept of self-reconfiguration. In several works on wireless networks the term used is $self-organization$. This term is also used to express the partitioning and clustering of ad-hoc networks or wireless networks to groups called cliques or clusters. Also, the self-organization term can be found in protocols for sensors networks to form a sphere or a polygon from a center node [17], [26]. The term *redeployment* is also a new term to address self-reconfiguration for sensor networks [14], [21]. In [22], a protocol of self-configuration where the desired configuration is grown from an initial seed module, after a generator uses a 3D CAD model of the target configuration and outputs a set of overlapping bricks which represent this configuration.

A growing number of research on self-reconfiguration for microrobots using centralized algorithms have been done, among them we find centralized self-assembly algorithms [20]. Other approaches give each node a unique ID and a predefined position in the final structure; see for instance [25]. The drawback of these methods is the centralized paradigm and

the need for nodes identification. More distributed approaches include [9], [13], [5], [23]. 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, is the name of a project led by Carnegie Mellon University and Intel corporation. Many works have already been done within the Claytronics project. In [6], [8], the authors propose a metamodel for the reconfiguration of catoms starting from an initial configuration to achieve a desired configuration using *creation* and *destruction* primitives. The authors use these two functions to simplify the movement of each catom. Another scalable algorithm can be found in [18]. In [2], a scalable protocol for Catoms self-reconfiguration is proposed, written with the MELD language [1], [19] and using the creation and destruction primitives. In all these works, the authors assume that all Catoms know the correct positions composing the target shape at the beginning of the algorithm and each node is aware of its current position. The first self-reconfiguration without predefined positions of the target shape appears in[15]. However, this solution is not parallelized and takes longer to achieve self-reconfiguration.

III. Contributions

In this paper, we propose a new distributed approach for parallelized self-reconfiguration of MEMS microrobots, where the target form is built in parallel incrementally, and each node in the current increment acts as a reference for other nodes to form the next increment, which will belong to the form.

We introduce a state model where each node can see the state of its physical neighbors to achieve the self-reconfiguration for distributed MEMS microrobots, using the states the nodes collaborate and help each other.

In the proposed algorithm, message exchange is limited to the construction of the spanning tree. The spanning tree is used to ensure the connectivity of the network and dynamically manage the nodes that can move. Contrary to existing works, in our algorithm each node has no information on the correct positions (predefined positions) of the target shape, and movement of microrobots is fully implemented.

We propose here an efficient, distributed and parallelized algorithm for nodes self-reconfiguration where each node moves by rotation around their physical neighbors. As an example, we study the case of a self-reconfiguration from a chain of microrobots to a square. The performance of the self-organization algorithm is evaluated according to the number of rotations and the time taken. In this paper the MEMS network is organized initially as a chain. 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 microrobots. Also, a chain of microrobots represents the worst case for message broadcasting complexity with $O(n)$, the reconfiguring to a square the complexity will be $O(\sqrt{n})$ in the worst case.

To assess the distributed algorithm performance, we present our results of simulations compared to the results in [15]. The simulations made with Meld [1] and the DPRSim simulator [28].

Outline of the paper. The rest of the paper is organized as follows: Section 4 discusses the model and some definitions. Section 5 discuss the proposed algorithm, analyzes the number of sent messages and the number of movements, it discuss memory space required and shows the generalization of the algorithm. Section 6 details the simulation results. Finally, section 7 summarizes our conclusions and illustrates our suggestions for future work.

IV. Model, definitions and tools

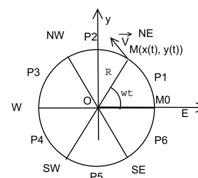

Figure 1. Node modeling, in each movement the node travels the same distance

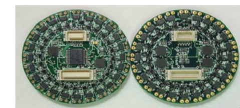

Figure 2. Two catoms.

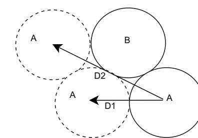

Figure 3. Traveled distance in one movement = 2R, the node A travels 2R in one movement

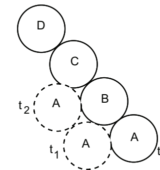

Figure 4. Message transmission, there will be message exchange if the node needs to know the state of a non-neighbor node

Within Claytronics, a Catom (figure 2) that we call in this paper a node is modeled as a sphere which can have at most six 2D-neighbors without overlapping (See figure 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)). In this work, the starting physical topology is a chain of n nodes linked together. A chain corresponds to a connected set of nodes where all nodes have two neighbors excepting the two extremities representing only one neighbor. We will take the example of nodes that have neighbors in NW and SE directions and we will show after how to generalize. A node A is in neighbor's list of node B if A touch physically B (figure 3). Communications are only possible through contact, which means that only neighbors can have a direct communication.

Consider the connected undirected graph *G = (V, E)* modeling the network, where $v \in V$, is a node that belongs to the network and, $e \in E$ a bidirectional edge of communication

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

between two physical neighbors. For each node $v \in V$, we denote the set of neighbors of v as $N(v)$. Each node $v \in V$ knows the set of its neighbors in G, denoted $N(v)$.
We define the following terminology:
$Connectivity$: in a graph $G = (V, E)$, if $\forall v \in V$, $\forall u \in V, \exists C_{v,u} \subseteq$E : $C_{v,u} = (e_{v,-}, ..., e_{-,-}, ..., e_{-,u})$, with $e_{x,y}$ 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 $t1..., tm$ are the time slots of execution of DA. There is a Snap-Connectivity in DA with the dynamic graph $G_t(V_ti, E_ti)$ the network state at the instant ti, if $\forall t_i, i \in \{1, ..., m\}$, $G_{ti}(V_{ti}, E_{ti})$ maintains the connectivity.
Spanning tree: is a tree composed of all $v \in V$ without any cycle. In the spanning tree, a node is either a child or a parent. The leaf is a node without any children.
We call the *highest number of movements* the highest number of movements was performed by a node belongs to the network.
To calculate the highest number of movements we define the following:
Consider the figure 1 which represents a microrobot. We say that a microrobot has done a single movement if the distance between its former position and its new position is exactly twice the radius $D1 = 2R$. For example, if the node is in a position at a distance $D2$ (see the figure 1) from the former position it has done two movements. We have 360° can be divided to six equal angles each one has 60°, since the perimeter at an angle a is $P_a = \pi\ Ra/180\ and\ P = 2\pi R$ we find $P1 = P2 = P3 = P4 = P5 = P6$, this means that the node can have without overlapping at most six neighbors and in each movement the node travels Ra (wih $a = 60°$) from $m0$ to m. In this paper, we assume that the change of message (consultation) between two physical neighbors is carried without complexity (0 message), while the distance between two physical neighbors is zero. If a node *to decide* needs to know the state of node which is not its physical neighbor message exchange is required, for example in the figure 4:

- At t_0: the node A needs to know the state of B to move to the new position, this movement is done without message exchange.
- At t_2: if A is in the new position and it needs to know the state of D to move then D sends a message to C informing its state to C that forwards the message to A. So, in this case there is a message exchange and A must wait two rounds to decide.
- But if at t_0 or at t_1 a message has been sent from D to C, so A at t_2 can have the state of D with a simple consultation of C's state.

It is important to minimize the number of movements vis-a-vis the energy and time of execution, the space of memory used, therefore the number of states per node.

Lemma 4.1: [1] Let x be an integer number. It is well-known that if x is odd/even, then x^2 is an odd/even number.

Proof: As x is odd/even , we can write $x = 2n + 1/x = 2n$. Therefore, $x^2 = (2n + 1)^2/x^2 = (2n)^2$. So, $x^2 = 4n + 4 + 1 = 2(2n^2 + 2) + 1/x^2 = 2(2x^2)$; which is an odd/even number. ∎

Theorem 4.2: Let y be an odd/even square number (y is an integer that is the square of an integer), then the next odd/even square number is $y + 4\sqrt{y} + 4$

Proof: As y is an odd/even square number, from lemma 1 we have $\sqrt{y} = \rho$, with ρ is odd/even integer number. So, as ρ is odd/even, the next odd/even number is $r = \rho + 2$, and because $r^2 = (\rho+2)^2 = \rho^2+4\rho+4$, we find $r^2 = y+4\sqrt{y}+4$. ∎

Theorem 4.3: Let y be a square number (y is an integer that is the square of an integer), then if y is odd/even the next even/odd square number is $y + 2\sqrt{y} + 1$.

Proof: As y is an odd/even square number, from lemma 1 and lemma 2 we have $\sqrt{y} = \rho$, with ρ is odd/even integer number. So, as ρ is odd/even, the next even/odd number is $r = \rho + 1$, and because $r^2 = (\rho + 1)^2 = \rho^2 + 2\rho + 1$, we find $r^2 = y + 2\sqrt{y} + 1$ ∎

V. PROPOSED PROTOCOL

A. Parallel Algorithm with Safe Connectivity (PASC)

As mentioned before, 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.

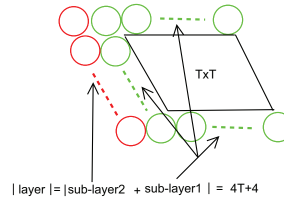

Figure 5. Represents how many nodes added to reach the next square when n is odd

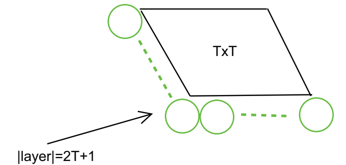

Figure 6. Represents how many nodes added in the last layer to reach the last square when n is even

To form the matrix of our square with $\sqrt{N} \times \sqrt{N}$ nodes, we begin with an incremental process with a correct square(for example 1x1). After, we add each time a new sub-layer contains $3T + 2$ nodes, with $T \times T$ is the last square. After, we add another sub-layer with $T + 2$ nodes taking positions at the W direction relative to nodes of the last shape. If N is even, at the last layer we add $2T + 1$ nodes, with $T \times T$ is the last square, figures 5 and 6 show an example. The choice of the middle node depends on the optimality of parallelism.
Let N is the network size, and $n = \left\lceil\sqrt{N}\right\rceil \left\lfloor\sqrt{N}\right\rfloor$, if n

[1]The character "/" in lemmas and theorems does not mean the division operation

is odd the middle node will be $mi = (n+1)/2$, as the case in figure 8. If n is even the middle node will be $mi = n/2 - ((\sqrt{n}/2) - 1)$, as the case in figure 7.
The middle node mi can be found by knowing the size of the network, an end node of the chain initializes a counter and broadcasts it, each node receives this message increments the counter until its arrives to the concerned node mi, that will have the satisfied predicate $medChain(v)$.

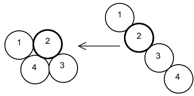

Figure 7. Represents an example of initiator finding when n is even, in this example the initiator is the node 2

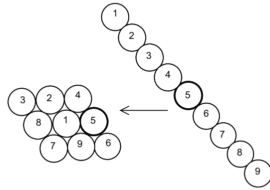

Figure 8. Represents an example of initiator finding when n odd, in this example the initiator is the node 5

B. Description and analysis

The algorithm runs in rounds, in each round satisfied predicates are chosen to run. The distributed algorithm seeks the desired form by using an incrementally process. In a completed increment, the nodes that build it belong already to the form; these nodes will help neighbor nodes and future neighbor nodes to get correct positions.
The middle node of the chain declares itself as an initiator with the predicate (1). The initiator which is the root of the tree initializes the tree and becomes a parent of itself (3). A node if it does not have a parent becomes a child of one the neighbor parents (8), and a node is a leaf if all its neighbors are parents (9). Nodes that are above the initiator take the state top with predicate (6), the other nodes that are under the initiator take the state $bottom$ (7). Initially, all nodes are initialized with state bad except the initiator (2), the initiator takes the states $well$ and $nper$ with (4) and (5). Nodes having the state $well$ or int are nodes already in the target shape and cannot move, they became steady.

To make optimal parallelism and correct square, the number of nodes having the state top to be in the same line as the initiator must be equal to the number of nodes having state $bottom$ to be in the same line as the initiator if N is odd. If N is even, another node is added to the nodes having state top. The state $nper$ is used to achieve this purpose. The initiator takes the state $nper$ (5), by taking this state the initiator and each node has this state does not allow neighbors to to move around it in order to join the line of the initiator. This is done with guard $(\neg state_v(nper))$. The state $mnper$ is an intermediate state used to propagate the state $nper$ to the other nodes, that will keep the parallelism optimal, as well the node that has a neighbor in the E direction having the state $nper$ takes the state $mnper$ (11). The node that has the initiator as neighbor node in the SE direction takes the state $mnper$ (10). The other (next) nodes that will take the state $nprem$ are nodes having in the E direction a neighbor that has the state $mnper$ (12). Therefore, the node having the state $nprem$ does not allow neighbor nodes to join the line of the initiator, as these nodes are checking the predicates (21), (22), (23), (26), (27), (28), (29) and (30).

The state int is an intermediate state used to add a non-complete layer to the square shape. Thus, the nodes that have neighbors having the state $well$ take the state int with predicate (13). The first node that changes its state to int is the one in the line of the initiator. After, the state int is propagated to nodes that have neighbors having the $well$ state. Notice that, nodes with the state $well$ and nodes with state int together do not composites a square, it well be a square if all nodes having the state int have in the W direction a neighbor node, this neighbor node has the state $well$. Therefore, the wave of state changing to $well$ begins with predicates (15), (16) and (17).

With predicates (18) and (19) the leaf nodes having the state top descend to the center of the chain. As well as, leaf nodes having the state $bottom$ mount to the center of the chain with the predicates (24) and (25). With predicate (20) / (21), leaf node v that has the bad state moves around a node u having the states top and $int/well$, node u becomes a neighbor in SE direction relative to v. With predicate (22) / (23), leaf node v that has the bad state moves around a node u having the states top and $int/well$, node u becomes v's neighbor in the E direction. With the predicate (26) / (27) / (28), leaf node v with bad state moves around a node u having $well$ and $bottom$ states, node u becomes v's neighbor in the NE / SE / E direction.

Theorem 5.1: If N is the network size and $n = \lfloor\sqrt{N}\rfloor \lfloor\sqrt{N}\rfloor$ is odd, the highest number of movements will be $((n+1)/2) + N - n$.

Theorem 5.2: If N is the network size and $n = \lfloor\sqrt{N}\rfloor \lfloor\sqrt{N}\rfloor$ is even, with $N \geq 5$, the highest number of movements will be $(\sqrt{n}/2) + N - (n/2) - 1$.

Example: Figure 9 shows an example with explanation, and figure 10 shows another example without explanation. In figure 9:

- At $t0$: with predicate (2) each node takes the state b (bad), with (6) nodes (node 1) which are above the initiator (node 2) take the state t (top), with (7) nodes (nodes 3 and 4) located under the initiator take the state B $(bottom)$, with (4) the initiator takes the state w $(well)$, and with (5) it takes state n $(nperm)$.
- To arrive at the next step $t1$, node 1 moves around node 2 using the predicate (18), and node 4 moves around node 3 using (26). The node 1 takes the state m with (10). The node 1 cannot move around the node 2 with (19) since the node 2 has the state n. Node 4 moves to the new position with (24) and it cannot get any other state in this step.

Variables and Predicates

–initiator$_v$(): node v that initializes the algorithm. –state$_v(X)$: v takes the state $X \in \{well, bad, int, nper, mnper, top, bottom\}$, v cannot take the states $well$ and bad in the same time. – moveAround$state_v(u, P_x)$: move around neighbor u that has the state $state$ in such a way u becomes v's neighbor in the direction x relative to v. – Parent(v, u): v is u's parent in the tree. –isLeaf(v): v is a leaf in the tree.

Predicates checked only in the first round

1) initiator$_v() \equiv medChain(v)$.
2) state$_v(bad) \equiv connected_v \wedge \neg initiator_v()$.
3) parent$(v, v) \equiv initiator(v)$.
4) state$_v(well) \equiv initiator_v()$.
5) state$_v(nper) \equiv initiator_v()$.

Predicates checked in each round

6) state$_v(top) \equiv (N_{se}(v) = u, initiator_u()) \vee (N_{se}(v) = u, state_u(top))$.
7) state$_v(bottom) \equiv (N_{nw}(v) = u, initiator_u()) \vee (N_{nw}(v) = u, state_u(top))$.
8) parent$(v, u) \equiv (parent(w, v), u \neq w) \wedge (u \in N(v)) \wedge (state_u(bad)) \wedge (\nexists z \in N(v), parent(v, z))$.
9) isLeaf$(v) \equiv ((\forall u \in N(v), \neg parent(v, u)) \wedge \neg parent(v, v))$.
10) state$_v(mnper) \equiv (((N_{se}(v) = u, state_u(nper)) \vee (N_e(v) = u, state_u(nper))) \wedge initiator_u())$.
11) state$_v(mnper) \equiv (N_e(v) = u, state_u(nper)) \wedge (\neg state_v(nper)) \wedge (state_v(int) \vee state_v(well))$.
12) state$_v(nper) \equiv (N_e(v) = u, state_u(mnper)) \wedge (\neg state_v(mnper)) \wedge (N_{se}(v))$.
13) state$_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee (N_e(v) = u1, N_{se}(v) = u2, state_{u1}(int), state_{u2}(int)) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
14) state$_v(well) \equiv ((N_e(v) = u, state_u(int)) \wedge (N_{nw}(v))) \vee ((N_e(v) = u, state_u(int)) \wedge (N_{se}(v)))$.
15) state$_v(well) \equiv (N_w(v) = u, state_u(well)))$.
16) state$_v(well) \equiv state_v(bad) \wedge (N_{se}(v) = \oslash) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well))$.
17) state$_v(well) \equiv state_v(bad) \wedge (N_{se}(v) = \oslash) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well)) \wedge ((N_e(v) = u1, state_{u1}(well)))$.
18) moveAround$bad_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(bad), state_u(top))$.
19) moveAround$bad_v(u, P_{ne}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(bad), state_u(top))$
20) moveAround$int_v(u, P_{se}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.
21) moveAround$well_v(u, P_{se}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(well), state_u(top)) \wedge (\neg state_u(nper))$.
22) moveAround$int_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.
23) moveAround$well_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(well), state_u(top)) \wedge (\neg state_u(nper))$.
24) moveAround$bad_v(u, P_{ne}) \equiv isLeaf(v) \wedge state_v(bad) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge (N_{nw}(v) = u, state_u(bad)) \wedge state_u(bottom)$.
25) moveAround$bad_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(bad), state_u(bottom))$.
26) moveAround$well_v(u, P_{ne}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(well), state_u(bottom), (\neg state_u(nper))$.
27) moveAround$well_v(u, P_{se}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(well), state_u(bottom), (\neg state_u(nper))$.
28) moveAround$well_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(well), (\neg state_u(nper))$.
29) moveAround$int_v(u, P_{ne}) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(int), state_u(bottom), (\neg state_u(nper))$.
30) moveAround$int_v(u, P_e) \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(int), state_u(bottom), (\neg state_u(nper))$.

The PASC algorithm

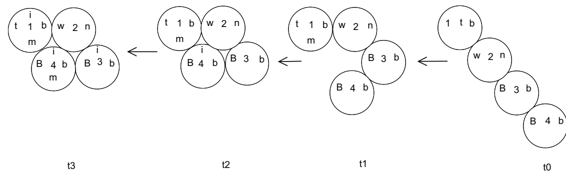

Figure 9. Represents an example of execution of PASC with four nodes

- To arrive at the next step $t2$, node 4 moves around node 3 using the predicate (25). After, node 4 takes the state i (int) with predicate (13).
- At $t3$, the target shape is obtained. Nodes 1 and 4 take the state i with (13).

C. The ten states minimum

In this section we prove that eight states are minimum to obtain the algorithm convergence. Obviously, with a single state, nodes have no way to distinguish whether they are in a good position or not and therefore if the node should move or not. Let us suppose a variant of PASC with two states bad and $well$, with these two states, we can say that the node that the node that has $well$ state is a steady node and is belonging to the target shape, and the node with bad state moves around nodes having $well$ state, thereby with these two states, nodes collaborate between them to make a next layer and change the state from bad to $well$. Suppose a set S of nodes having the $well$ state are correctly in the target shape. Depends on some conditions C the set B of nodes with bad state will change their state to $well$ in order to make a new layer, however as we have only two states the other nodes that are B's neighbors have likewise these C conditions and they will change their states to $well$ and became steady, although they are not even at the layer being built. So, PASC is executed and the target shape is lost.
Two additional states are required, the state top and $bottom$, these two states are indispensable to avoid deadlock in PASC. Indeed, in PASC there are predicates (18) and (19) executed by nodes to descend to the middle of the chain, and others executed to rise to the middle of the chain (24) and (25), if we remove the states top and $bottom$ from (18), (19), (24) and (25) the nodes will remain in their position by running (18)/(25) after (25)/(18), or (19)/(24) and (24)/(19) cyclically. Therefore, PASC will not get finished. A variant of PASC with four states bad and $well$ and top and $bottom$ is impossible, the reasons are the same used to prove the impossibility with the two states $well$ and bad.

The thing seen so far, is that we have to add an intermediate state int to separate neighboring nodes having the state bad and nodes having $well$ state. By adding this state, the node that has int state can change the C conditions that will be C'. Such a way, B's neighbors cannot change their state to $well$ with C', because they are not forming a new correct layer.

Let us suppose a variant of PASC with six states bad, $well$, top, $bottom$, and int, $\neg int$. With five states the deadlock is avoided, and the conditions to change the state to $well$ are managed. However, the node having the state int are making a new layer adjacent to the current correct square $\sqrt{Z} * \sqrt{Z}$, the number of node having int added is $3\sqrt{Z}+2$. Therefore, as $\sqrt{Z} * \sqrt{Z}$ + $3\sqrt{Z}+2$ is not a square root (from Theorem 4.2 and Theorem 4.3), the shape is not a square. To become a square we have to add $\gamma = \sqrt{Z}+2$ nodes, this nodes will be at the direction W relative to nodes having the state int. These γ can get the state $well$ beacause the shape is a square or an intermediate square. With six states bad, $well$, top, $bottom$, int, $\neg int$, the parallelism will not be optimal and the energy consumption will not be well balanced between nodes. To make an optimal parallelism, PASC makes two rectangles in parallel where the union gives a square. Also, to propagate the state $nperm$ to the concerned nodes of the middle, we have to use another state $mnper$, the states $\neg nper$ and $\neg mnper$ are used to check if the neighbor node has the state $nper$ and $mnper$ respectively. ■

D. Complexity of sent messages

PASC needs only $O(N)$ message. That is, the messages of tree construction $(O(N/2))$ and the messages of middle node finding $(O(N/2))$. The most interesting action for message exchange in the algorithm is the one activated by state changing predicates, from the int and bad states to $well$ with the predicates (15), (16) and (17), it is obvious that if a node changes its state before it is sure of the $well$ 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 and self-reconfiguration desired, the predicates (10), (11), 12), (13), (14), (15) and (16) ensures without exchanging of message that the node changes its state only if all nodes that have moved before it have changed their states, therefore the first node that begins the construction of the new layer does not need to wait for the message of the first node that began the previous layer. Since the node that is currently checking the predicates (10), (11), (12), (13), (14), (15) and (16) can have this information by simply consulting (message) the state of its former neighbors. In other words, the message was being sent before the node needs to know the state of its sender, when the node needs to know it, it will find the message at its physical neighbor. So we do not need to transmit information from the node blocked necessarily in a good position with the $well$ state to other nodes which are forming the new layer which explains that throughout the algorithm in any case we do not need to transmit information between two non-neighboring nodes of the new layer. This efficiency is explained by the fact that synchronization in state changing is not required for nodes that are in the same layer. As consequence, PASC needs only the messages of tree construction and the messages of middle node finding.

E. Generalization of the algorithm

Presented algorithm PASC is specific to a chain case where nodes form initially a straight line oriented toward SE-NW directions. In this section we describe how the algorithm can be generalized to any kind of initial straight chain with any direction as shown in figure 11. We start by explaining how

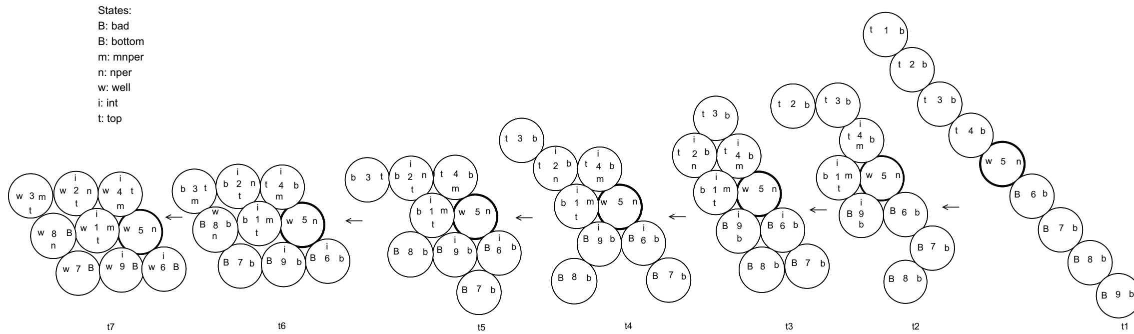

Figure 10. Represents an example of execution of PASC with nine nodes, the initiator is the node 5

the two end nodes are selected whatever are the directions of the straight chain. The node that has only one neighbor situated either in SW, SE or E direction is the first end node. The second end node has only one neighbor situated either in NW, NE or W. For the other nodes, every node in the chain can deduce the orientation of the chain (one of the three cases represented in figure 11) by analyzing the orientation of its two neighbors, they use the orientation of their two neighbors to determine the orientation of the formed chain. Generally, every node after the detection of the chain orientation, noted D-$\overline{D}$, runs a variant of the PASC algorithm depending of the orientation $D \in \{W, NW, NE\}$. The variant of PASC algorithm, $PASC^D$, represents an adaptation of the the original PASC algorithm (corresponding to $PASC^{NW}$) to the two other possible orientations with changing the directions in predicates. For instance, if the initial chain is oriented NE-SW, the algorithm $PASC^{NE}$ is called, and the square form is realized using moves of type $moveAroundbad_v(u, P_w)$, $moveAroundwell_v(u, P_w)$ and $moveAroundwell_v(u, P_{nw})$. The usage of these three predicates is described in figure 12 that presents an example with nodes having the state *bottom*.

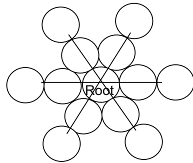

Figure 11. The three possible cases of a straight chain

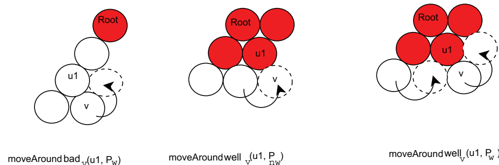

Figure 12. Moves adaptation in the case of NE-SW chain. Dark nodes represent the well-state node

VI. Simulation

We have done the simulation with the declarative language Meld that uses the DPRSim simulator. In our simulations the radius of the node is 1 mm. We simulated with a laptop with processor Intel(R) Core(Tm) i5, 2.53 Ghz with 4 Go of memory. We note in the figures of simulation, $PASC1$ for the values odd of $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$, and $PASC2$ for the values even of n, with N is the network size.

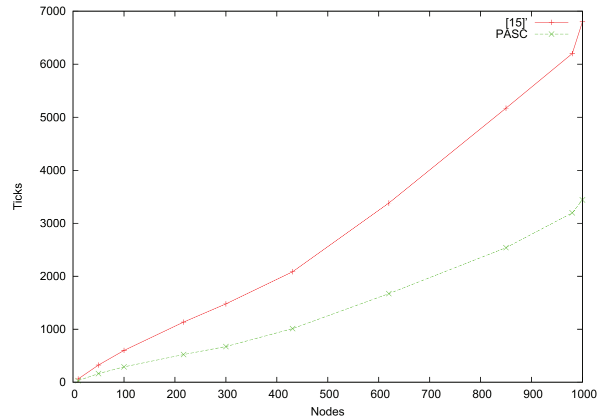

Figure 13. Execution time

The simulation results come to agree our theoretical results. The figure 13 represents the execution time in ticks by the number of nodes; this figure compares the execution time of the algorithm proposed in this paper to the one given in [15]. Figure 14 presents the highest number of movements found in this paper compared to the one in [15]. With, $g(N) = (\sqrt{n}/2) + N - (n/2) - 1$ and $f(N) = ((n+1)/2) + N - n$, where $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$.

The effects of parallelism appear well in the curve representing the execution time of PASC and [15], as PASC makes two rectangles in the same time. We see that whenever the network size increases the difference increases dramatically. We remark in figure 14 that the number of movements in PASC is much lower, which will increases the probability of lifetime of nodes, therefore, the probability that the node continues its task (its

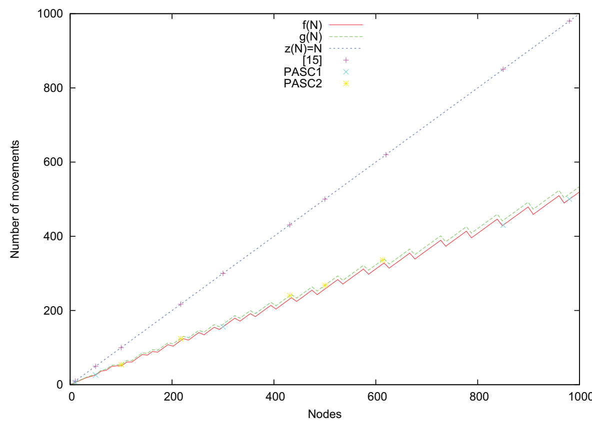

Figure 14. Highest number of movements

movements). However, PASC needs eight states per node and the algorithm in [15] needs three states per node.

VII. Conclusion

In this paper, we have shown the self-reconfiguration parallelized possibility without predefined positions of the target shape; we presented an algorithm where nodes help each other to achieve the self-reconfiguration using an incrementally process. The proposed algorithm ensures the connectivity of the network throughout the execution time of the algorithm, each node needs eight states to help and collaborate with neighbors, its execution time and highest number of movements is much better than that proposed in [15].

However, some open problems remain. We are studying the conception of an energy-efficient algorithm when the starting form may be any connected shape, in which we predict the loss of these previous characteristics described in this paper, in particular the number of states of each node and the message exchange. Another questions remain, the derivation of a fault tolerant algorithm for faulty MEMS nodes is to be investigated. The study of the effect of self-reconfiguration on the permutation routing [16] where the objective will be to optimize the path of a node to go to the correct position where it finds its correct data.

VIII. ACKNOWLEDGMENTS

This work is funded 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] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, *Meld: A Declarative Approach to Programming Ensembles,* In Proceedings of the IEEE Int. Con. on Intelligent Robots and Systems, October, 2007.

[2] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, Padmanabhan Pillai, and Jason D. Campbell, *A Language for Large Ensembles of Independently Executing Nodes,* In Proc. of the Int. Con. on Logic Programming, 2009.

[3] J. Bourgeois and S.C. Goldstein. *Distributed Intelligent MEMS: Progresses and perspectives*, In the 3-rd Int. Conf. ICT Innovations, Ohrid, Macedonia, September, 2011.

[4] H. Bojinov, A. Casal, T. Hogg, *Emergent structures in modular self-reconfigurable robots,* Proceedings of the IEEE Int. Con. on Robotics and Automation, vol. 2, pp. 1734-1741, Los Alamitos, 2000.

[5] Z. J. Butler, K. Kotay, D. Rus, K. Tomita, *Generic decentralized control for lattice-based self-reconfigurable robots*, International Journal of Robotics Research 23(9):919-937, 2004

[6] D. Dewey, S. S. Srinivasa, M. P. Ashley-Rollman, M. D. Rosa, P. Pillai, T. C. Mowry, J. D. Campbell, and S. C. Goldstein, *Generalizing Metamodules to Simplify Planning in Modular Robotic Systems,* In Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems, September, 2008

[7] T. Ebefors, J.U. Mattsson, E. K. lvesten, and G. Stemme, *A walking a silicon microrobot*, in The 10th Int. Conf. on Solid-State Sensors and Actuators (Transducers '99), pages 1202-1205, Sendai, Japan, June 1999.

[8] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, *Distributed Localization of Modular Robot Ensembles,* In Proceedings of Robotics: Science and Systems, June, 2008.

[9] C. Jones, M. J. Mataric, *From local to global behavior in intelligent self-assembly.In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 721-726, Los Alamitos, 2003.

[10] S. Jeon, C. Ji, *Randomized Distributed Configuration Management of Wireless Networks: Multi-layer Markov Random Fields and Near-Optimality* CoRR abs/0809.1916, 2008.

[11] S. Hollar, A. Flynn, C. Bellew, and K.S.J. Pister, *Solar powered 10mg silicon robot*, In MEMS, Kyoto, Japan, January 2003.

[12] M. E. Karagozler, A. Thaker, S. C. Goldstein, D. S. Ricketts, *Electrostatic Actuation and Control of Micro Robots Using a Post-Processed High-Voltage SOI CMOS Chip*, IEEE International Symposium on Circuits and Systems (ISCAS), May 2011.

[13] K. Katoy, D. Rus, M. Vona, and C. McGray, *The Self-reconfiguring Robotic Molecule*, in Proceedings of the 1998 IEEE International Conference on Robotics and Automation, Leuven, 1998.

[14] F. Kribi, P. Minet, A. Laouiti, *Redeploying mobile wireless sensor networks with virtual forces*, IFIP Wireless Days, Paris, France,2009.

[15] H. Lakhlef, H. Mabed, J. Bourgeois, *Distributed and Efficient Algorithm for Self-reconfiguration of MEMS Microrobots*, in the 28th ACM Symposium On Applied Computing, Coimbra, Portugal, March 2013.

[16] H. Lakhlef, J. F. Myoupo, *Secure permutation routing protocol in multihop wireess sensor networks,* International Conference on Security and Management (SAM'11), pp. 691-696, 2011.

[17] M. Mamei, M. Vasirani, F. Zambonelli, *Experiments of Morphogenesis in Swarms of Simple Mobile Robots*, Journal of Applied Artificial Intelligence, 8(9-10):903-919, Oct. 2004.

[18] R. Ravichandran, G. Gordon, and S. C. Goldstein: *A Scalable Distributed Algorithm for Shape Transformation in Multi-Robot Systems,* In Proceedings of the IEEE Int. Con. on Intelligent Robots and Systems, October, 2007.

[19] M. D. Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, *Programming Modular Robots with Locally Distributed Predicates,* In Proceedings of the IEEE Int. Con. on Robotics and Automation, 2008.

[20] D. Rus, M. Vona, *Crystalline robots: Self-reconfiguration with compressible unit modules*, Autonomous Robots 10(1), 107-124, 2001.

[21] R. Soua, L. Saidane, P. Minet, *Sensors deployment enhancement by a mobile robot in wireless sensor networks*, IEEE ICN 2010, Les Menuires, France, April 2010.

[22] K.Stoy, R.Nagpal, *Self-reconfiguration using Directed Growth*, 7th International Symposium on Distributed Autonomous Robotic Systems (DARs), France, June23-25, 2004.

[23] W. Shen, P. Will and A. Galstyan, Hormone-inspired self-organization and distributed control of robotic swarms. Autonomous Robots 17(1), 93-105, 2004.

[24] B. Warneke, M. Last, B. Leibowitz, and K.S.J Pister,K.S.J., 2001, *Smart Dust: Communicating with a Cubic-Millimeter Computer*, Computer Magazine, pp. 44-51, 2001.

[25] P. White, V. Zykov, J. C. Bongard, H. Lipson, *Three dimensional stochastic reconfiguration of modular robots* In: Proceedings of Robotics Science and Systems, pp. 161-168. MIT Press, Cambridge , 2005

[26] F. Zambonelli, M.P. Gleizes, M. Mamei, R. Tolksdorf, *Spray Computers: Explorations in Self-Organization*, Journal of Pervasive and Mobile Computing, Elsevier, Vol. 1, p. 1-20, 2005.

[27] *http://today.duke.edu/2008/06/microrobots.html*

[28] *http://www.pittsburgh.intel-research.net/dprweb*

[29] *http://smartblocks.univ-fcomte.fr/*