A Distributed Self-Reconfiguration Algorithm for Cylindrical Lattice-Based Modular Robots

André Naz[*], Benoît Piranda[*], Seth Copen Goldstein[**] and Julien Bourgeois[*]

[*]Université de Franche-Comté, FEMTO-ST Institute, UMR CNRS 6174
{andre.naz, benoit.piranda, julien.bourgeois}@femto-st.fr
[**]School of Computer Science, Carnegie Mellon University
seth@cs.cmu.edu

Abstract

Modular self-reconfigurable robots are composed of independent connected modules which can self-rearrange their connectivity using processing, communication and motion capabilities, in order to change the overall robot structure. In this paper, we consider rolling cylindrical modules arranged in a two-dimensional vertical hexagonal lattice. We propose a parallel, asynchronous and fully decentralized distributed algorithm to self-reconfigure robots from an initial configuration to a goal one. We evaluate our algorithm on the millimeter-scale cylindrical robots, developed in the Claytronics project, through simulation of large ensembles composed of up to ten thousand modules. We show the effectiveness of our algorithm and study its performance in terms of communications, movements and execution time. Our observations indicate that the number of communications, the number of movements and the execution time of our algorithm is highly predictable. Furthermore, we observe execution times that are linear in the size of the goal shape.

Keywords— Distributed algorithm, Self-reconfiguration algorithm, Modular robotic, Programmable Matter, Ensembles

1 Introduction

Modular Self-reconfigurable Robots (MSR) [1] are distributed robotic systems composed of independent connected modules which are able to collaborate and coordinate their activities in order to achieve common goals. Every module has its own computation and communication capabilities, sensors and actuators. MSR have a wide range of potential applications. This work is part of the Claytronics project [2, 3] in which we envision massive-scale MSR, composed of up to millions of modules, to build programmable matter, i.e., matter that can change its physical properties under program control.

The most used algorithm in MSRs is the self-reconfiguration algorithm which causes the modules to move from one configuration (the *initial shape*) to another one (the *goal shape*) (see Figure 1). Self-reconfiguration has several applications. In the context of programmable matter, it enables an MSR to assume different shapes. Self-reconfiguration can also be used to adapt MSR to changes in the environment or to specific tasks. For instance, in [4], the authors use the self-reconfiguration to rearrange modules connectivity in order to reach an optimal network topology.

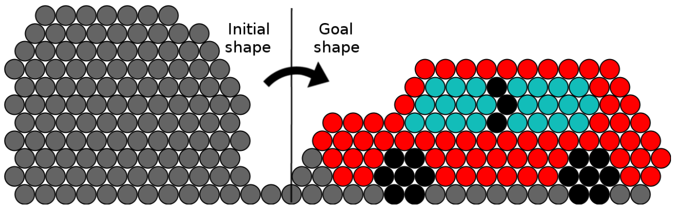


Figure 1: Example of initial and goal shapes. Self-reconfiguration is the process during which the initial clump of modules on the left self-reconfigures into the car shape on the right.

Self-reconfiguration algorithms pose several challenges. Firstly, planning is challenging as the number of possible unique configurations is huge: $(c \cdot w)^n$ where n is the number of modules, c the number of possible connections per module and w the ways of connecting the modules together [5]. Depending on the physical constraints, modules can often move concurrently which makes the configuration space grow at the rate of $O(m^n)$ with m the number of possible movements and n the number of modules free to move [6]. The exploration space for reconfiguration between two random configurations is therefore exponential in the number of modules which prevents finding a complete optimal planning for all but the simplest configurations. The optimal self-reconfiguration planning for chain-type MSRs is then an NP-complete problem [7], and, to the best of our knowledge, nothing has been proved so far for lattice-based MSR. Secondly, in addition to the path planning problem, the self-reconfiguration process is also challenging as it is a distributed process that requires distributed coordination of mobile autonomous modules connected in time-varying ways. In particular, modules have to coordinate their motions in order to not collide with each other.

Self-reconfiguration algorithms are tailored for a specific class of modular robots, with specific motion constraints [8], for example using cubes sliding on the floor, some motions need a cooperation process that complicates motion algorithms [9]. In this paper, we base our model on the millimeter-scale cylindrical robots [10, 11] (see Figure 2), called 2D Catoms, developed in our project. Catoms are the basic unit for Claytronics. 2D Catoms have been partially validated with the realization of a hardware prototype. In this paper, we assume 2D Catoms can communicate together using neighbor-to-neighbor communications and move by rolling around each other as long as they respect some motion constraints (see section 2).

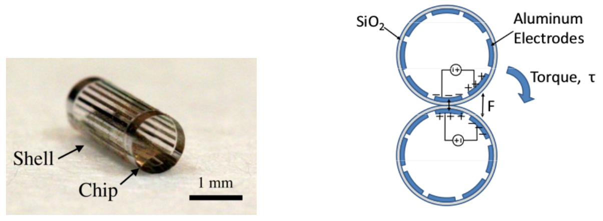


Figure 2: The 2D Catom. A fabricated prototype (on the left) and the actuation scheme (on the right) [10].

The contribution of this paper is to propose the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm which is asynchronous, deterministic, fully decentralized and able to manage almost any kind of initial and goal compact shapes (see section 4). Although our work is focused on the algorithm, we carry out our analysis with respect to hardware constraints based on the 2D Catoms prototype developed in [10, 11]. C2SR is a step toward realizing programmable matter.

We implemented our algorithm in C++ and evaluated it through simulations with our simulator, VisibleSim [12, 13]. We show the effectiveness of C2SR on large-scale ensembles composed of up to ten thousands of modules. We also show the effectiveness of our algorithm and study its performance in terms of communications, movements, and execution time. Our observations indicate that the number of communications, the number of movements and the execution time of our algorithm is predictable. Furthermore, its execution time appears to be linear in the size of the goal shape.

The rest of this paper is organized as follows. In section 2, we define the system model and assumptions. Afterwards, we discuss the related work in section 3. In section 4, we present the general idea of C2SR and in section 5, we describe its implementation. In section 6, experimental results are presented and analyzed. Section 7, concludes this paper and section 8 proposes some directions for future work.

2 System Model and Assumptions

In this paper, we consider the millimeter-scale cylindrical robots [10, 11] (see Figure 2), called 2D Catoms, developed in the Claytronics project. Some of the 2D Catoms functionalities have been validated using this prototype.

A 2D Catom consists of a 6-mm long and 1-mm diameter cylindrical shell. A high voltage CMOS die is attached inside the tube. The chip includes a storage capacitor and a simple logic unit. The tube has electrodes used for power transfer, communications and actuation. The power is spread from a powered floor through the ensemble using neighbor-to-neighbor power transfer.

We assume that 2D Catoms are organized into a horizontal pointy-topped hexagonal lattice where modules have up to six neighbors. Modules can communicate together using neighbor-to-neighbor communications. We assume that modules automatically discover their neighbors using communications after becoming attached. We assume that moving modules cannot communicate with any other module. $\mathcal{N}_{C_i}^N$ denotes the network neighbors of the module C_i. Catoms on the periphery have clockwise (CW) and counter-clockwise (CCW) neighboring Catoms that also belong to the periphery. For instance, in Figure 3, C_9 is C_6's CW peripheral neighbor and C_{10}'s CCW one. C_{11} is both C_{12}'s CW and CCW peripheral neighbor.

$p_{C_i} = (x_{C_i}, y_{C_i})$ denotes the coordinates of the 2D Catom C_i in the horizontal hexagonal lattice. $p_{C_i}.x$ denotes C_i's column in the lattice, while $p_{C_i}.y$ denotes C_i's height. For instance, in Figure 3, $p_{C_2}.y = 0$ and $p_{C_9}.y = 2$. We assume that, at any time, modules know both their coordinates in the lattice and the coordinates of their neighbor through an external algorithm, e.g., [14] or a distributed and incremental version of [15].

Moreover, a 2D Catom can roll CW or CCW around a stationary module. During an atomic move, a module rotates 60° going from one cell of the lattice to its adjacent cell. We assume that a 2D Catom has only the capability to lift itself, it cannot carry or push other modules. A module can move if it satisfies the freedom of movement rule (see Rule 1).

Rule 1 (the freedom of movement rule) *Because of possible mismatching issues due to physical constraints, a 2D Catom can only move from/into a cell if this cell is currently unoccupied and no two symetrically opposing cells adjacent to that cell are occupied (see Figure 3). Furthermore, we consider the floor as if it were filled with 2D Catoms. If a 2D Catom, C_i, satisfies the freedom of movement rule, $free(C_i)$ is true, otherwise it is false.*

In the current design, a 2D Catom is able to perform a revolution in 1.67 seconds or 3.35 seconds [11], which corresponds to an average speed of 1.88 $mm \cdot s^{-1}$ or 0.94 $mm \cdot s^{-1}$. We assume that 2D Catoms are not provided with any hardware mechanism to handle collision. Thus, collisions have to be prevented by the self-reconfiguration algorithm, using communications.

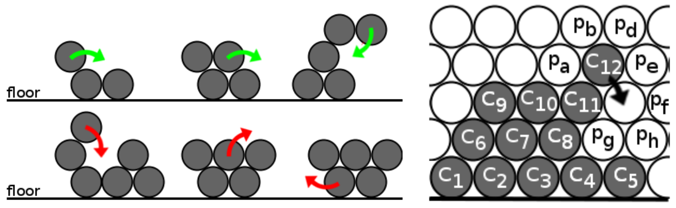


Figure 3: On the left, motion constraints: examples of feasible (on the top) and infeasible moves (on the bottom). On the right, a labeled system: gray cells are occupied by a module whereas white cells are empty. Some of the empty cells are labeled with their position (e.g., p_a, p_b, etc.).

We use $\mathcal{N}_p^K$ to denote the set of modules geographically adjacent to position p. A module C_i, moving from p_{C_i} to p'_{C_i}, is somewhere between these two positions, and thus, C_i belongs to the set of geographically adjacent modules of all the cells adjacent to p_{C_i} or p'_{C_i}. For instance, in the labeled system depicted in Figure 3, module C_{12} is moving and, thus it belongs to $\mathcal{N}_{p_a}^K$, $\mathcal{N}_{p_b}^K$, $\mathcal{N}_{p_d}^K$, $\mathcal{N}_{p_{C_{12}}}^K$, $\mathcal{N}_{p_e}^K$, $\mathcal{N}_{p_{C_{11}}}^K$, $\mathcal{N}_{p'_{C_{12}}}^K$, $\mathcal{N}_{p_f}^K$, $\mathcal{N}_{p_g}^K$ and $\mathcal{N}_{p_h}^K$. Note that in the presence of moving modules, $\mathcal{N}_{p_{C_i}}^K$ may be different from $\mathcal{N}_{C_i}^N$. Also notice that the construction of the $\mathcal{N}^K$ sets is not automatic. 2D Catoms are not equipped with any presence sensor. Maintaining on Catoms the $\mathcal{N}^K$ set of some specific nearby positions, using only communications, is one of the key operations in the implementation of our algorithm.

$\mathcal{I}$ and $\mathcal{G}$ respectively denote the initial and the goal shapes. We assume that every module stores a representation of the shape geometry of $\mathcal{G}$. Our algorithm also assumes some admissibility conditions for $\mathcal{I}$ and $\mathcal{G}$ (see section 4).

In this paper, colors are used for illustration purposes only. The current prototype is not equipped with any mechanism to glow with color. It is possible to do so, but the weight of that color mechanism will probably change the 2D Catom motion speed.

Furthermore, we assume a failure-free environment, i.e., we assume there is no module, communication, move or lattice failure during the algorithm execution.

3 Related Work

Self-reconfiguration and self-assembly have attracted a lot of attention in the last two decades. Algorithms have been proposed for modules of different shapes, with different physical motion constraints and arranged in various ways. In this paper, we consider self-reconfiguration of 2D Catom systems, rolling elements organized in a vertical and two-dimensional hexagonal lattice. Algorithms also differ by their restriction on the initial and goal shapes. Our algorithm can manage almost any kind of initial and goal compact shapes (see section 4). Algorithms also vary in their control properties. In particular, they can be centralized or distributed and synchronous or asynchronous. In this paper, we propose a distributed and asynchronous algorithm.

In [16], the authors propose a distributed algorithm to perform chain-to-chain self-reconfiguration in a hexagonal lattice. Modules move in synchronous rounds. This work was latter extended to allow self-reconfiguration from a chain configuration to an arbitrary shape with some admissibility conditions [17, 18]. These algorithms assume less restrictive motion constraints than the motion constraints we assume for the 2D Catoms. For instance, these algorithms allow the two first motions described as infeasible in Figure 3, starting from the left.

Self-reconfiguration presented in [19, 20] consists in using map-less representation for describing shapes. The benefit lies in a reduced memory footprint, but the number of supported goal shapes is limited. Proposed distributed algorithms manage to construct square shapes with spherical modules arranged in a two-dimensional hexagonal lattice. Due to the fact that initial and goal shapes are fixed, the number of movements can be predicted.

Algorithms to reconfigure an initial clump of modules arranged in a hexagonal lattice to a chain configuration were proposed in [21, 22]. These algorithms do not require message passing and do not use any pre-processing. In these algorithms, modules can both rotate and slide over other modules. Thus, these algorithms assume less restrictive motion constraints than ours.

In [23], the authors propose a distributed shape formation algorithm based on hole motions, for ensembles arranged in a hexagonal lattice. This algorithms can construct various shapes by randomly moving empty spaces within the ensemble. Although a wide variety of shapes can be built, this algorithm requires less restrictive motion constraints than ours, e.g., it allows the two first infeasible motions in Figure 3.

In [24], the authors propose a parallel, decentralized and asynchronous algorithm for the Kilobot swarm system [25] to self-assemble almost any kind of compact two-dimensional shapes. This algorithm has been applied on hardware systems with more than a thousand individual robots per swarm entities. However, these swarm robots have different physical motion constraints. During the self-assembly process, Kilobots may collide with one another. While this is possible with Kilobots, this is not acceptable in our system.

Existing protocols contain interesting ideas but consider different physical motion constraints, different restrictions on the initial and the goal shapes and different control properties. The contribution of this paper is to propose a distributed, fully decentralized, asynchronous and parallel self-reconfiguration algorithm for 2D Catoms that can manage almost any kind of initial and final compact shapes.

4 C2SR Algorithm at a Glance

In this section, we present the general idea of the Cylindrical-Catoms Self-Reconfiguration (C2SR) algorithm[1] that reconfigures a robot composed of modules from an initial shape $\mathcal{I}$ to a goal one $\mathcal{G}$.

Both shapes have to satisfy some admissibility conditions. We provide some intuitions about them in this paragraph and in Figure 4. A more formal description of the conditions and their demonstration are left for future work. Both shapes are compact, i.e., they do not contain holes, they are homeomorphic to a sphere. Moreover, both shapes are next to each other and intersect in one or more bottom cells. Let the peripheral path be the path formed from the empty cells on the periphery of both shapes, starting from and ending at the second horizontal layer (see Figure 4). This path has to be large enough to allow some modules, which progress along that path in the same direction with an empty space of at least one cell between successive modules, to move without violating our motion constraints and without risking colliding/getting attached with one another (see Figure 4 and Rule 1). Note that this condition implies that, at the upper layers, the horizontal space between the initial and the goal shapes has to be sufficiently large to enable these modules to move between the two shapes. Furthermore, the number of 2D Catoms in $\mathcal{I}$ has to be greater or at least equal to the number of target positions in $\mathcal{G}$ (i.e., $|\mathcal{I}| \geq |\mathcal{G}|$).

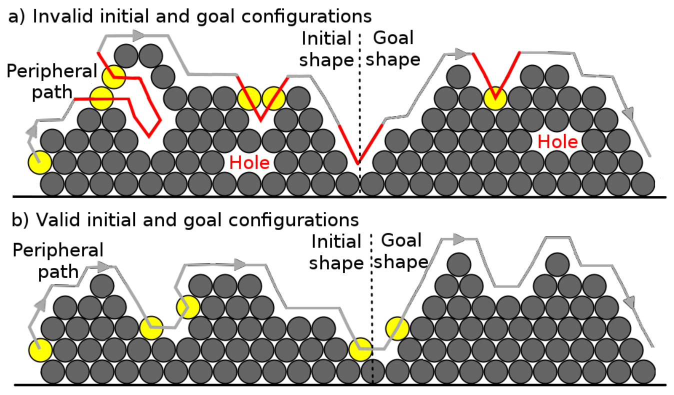


Figure 4: Invalid (on the top) and valid (on the bottom) initial and goal configurations. Modules in yellow, which are not part of the initial or the goal shapes, progress along the peripheral path in the same direction with an empty space of at least one cell between successive modules. The configurations on the top are not valid for several reasons. First, they do not intersect in at least one cell. Second, they both contain a hole. Third, the peripheral path is not large enough in locations in red. Indeed, modules in yellow could not move without violating our motion constraints and without getting attached with each other.

During the execution of C2SR with shapes individually composed of only continuous horizontal layers, the goal shape is progressively constructed from the bottom layer to the top one by stripping the initial shape, module by module in the reverse order (see Figure 5). Because of physical constraints, at a given instant, only modules on the periphery can move. In order to avoid module collisions and deadlocks, peripheral modules form a stream: modules roll in the same direction d (CW in Figures 1 and 5), and maintain an empty cell between each other using message exchanges. Modules in the stream do not overtake each other.

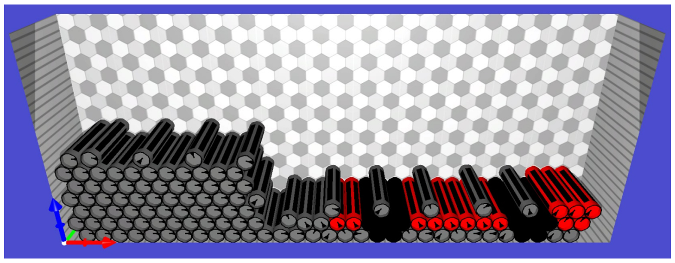


Figure 5: Screenshot during the self-reconfiguration process with the initial and goal shapes of Figure 1. Modules in the stream progress by rotating CW.

A module locally decides to start taking part in the stream if it satisfies the stream entrance rule (see Rule 2). Intuitively, a free module enters the stream if moving in the direction d consists in: moving around a module on the ground, or descending $\mathcal{I}$, or moving around $\mathcal{G}$, or moving in $\mathcal{G}$ without leaving it and without going up.

Rule 2 (the stream entrance rule) *Let us consider two modules C_i and C_j such that both C_i and C_j are on the periphery and C_j is the next peripheral neighbor of C_i in the direction of rotation, d. p'_{C_i} denotes the position that C_i would occupy after its rotation around C_j. C_i decides to take part in the stream if the following logical condition is satisfied:*

$$\begin{aligned} stream(C_i) :- \ & free(C_i) \\ & \wedge \ (\ (p_{C_i} \notin \mathcal{G} \wedge p_{C_j}.y = 0) \\ & \quad \vee \ (p_{C_i} \notin \mathcal{G} \wedge p'_{C_i}.y \leq p_{C_i}.y) \\ & \quad \vee \ (p_{C_i} \notin \mathcal{G} \wedge p_{C_j} \in \mathcal{G}) \\ & \quad \vee \ (p_{C_i} \in \mathcal{G} \wedge p'_{C_i} \in \mathcal{G} \wedge p'_{C_i}.y \leq p_{C_i}.y) \ ) \end{aligned}$$

A module in the stream decides to move if it satisfies the stream progression rule (see Rule 3). More precisely, a module in the stream can move if the set of modules geographically adjacent to its destination cell contains no more than three modules and none of them, except the module itself, belongs to the stream (see Figure 6). This rule requires local interactions with neighbors adjacent to its source and destination positions. These modules are at most two cells away. The admissibility conditions on $\mathcal{I}$ combined with the two rules above, guarantee that these modules are at most five network hops away.

[1]Some examples of self-reconfiguration with C2SR are available online in video at https://youtu.be/XGnY-oS4Nw0

Rule 3 (the stream progression rule) *A module C_i can move from its position p_{C_i} to the position p'_{C_i} if the following condition is satisfied:*

$$
\begin{aligned}
progression(C_i) :- \; & stream(C_i) \\
& \wedge |\mathcal{N}^K_{p'_{C_i}}| \leq 3 \\
& \wedge \nexists C_j \in \mathcal{N}^K_{p'_{C_i}} \mid C_j \neq C_i \wedge stream(C_j)
\end{aligned}
$$

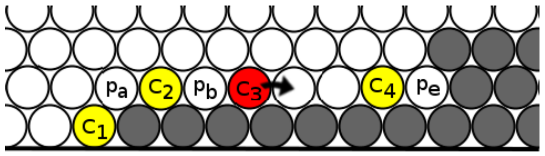


Figure 6: Stream progression rule: a simple example. Modules should rotate CW. White cells are empty and some of them are labeled with their position in the lattice (e.g., p_a, p_b, etc.). Modules C_1, C_2, C_3 and C_4 are in the stream. C_3 is moving. C_1 cannot move because C_2 is in the stream and $C_2 \in \mathcal{N}^K_{p_a}$. C_2 cannot move because C_3 is in the stream and $C_3 \in \mathcal{N}^K_{p_b}$. C_3 can move to p'_{C_3} because $\mathcal{N}^K_{p'_{C_3}}$ contains only three modules and none of them is in the stream, except C_3. C_4 cannot move because $|\mathcal{N}^K_{p_e}| = 5$.

Rule 3 prevents collisions. The admissibility conditions on $\mathcal{I}$ and $\mathcal{G}$, combined with Rules 2 and 3 prevent deadlock. Note that, because of the stripping order and the construction order, our algorithm also guarantees that at all time the system remains connected.

Each module checks for convergence using Rule 4 at the initialization and after every move. A module has converged if it is initially in a goal position, or if it has reached $\mathcal{G}$ and moving in direction d will cause it to leave $\mathcal{G}$ or to go up.

Rule 4 (the local convergence rule) *Let us consider two modules C_i and C_j such that both C_i and C_j are on the periphery and C_j is the next peripheral neighbor of C_i in the direction of rotation. p'_{C_i} denotes the position that C_i would occupy after its rotation around C_j. C_i has converged if it satisfies the following condition:*

$$
\begin{aligned}
converged(C_i) :- \; & (p_{C_i} \in \mathcal{I} \wedge p_{C_i} \in \mathcal{G}) \\
& \vee (p_{C_i} \in \mathcal{G} \wedge p'_{C_i} \notin \mathcal{G}) \\
& \vee (p_{C_i} \in \mathcal{G} \wedge p'_{C_i} \in \mathcal{G} \wedge p'_{C_i}.y > p_{C_i}.y)
\end{aligned}
$$

Applying these rules in a distributed asynchronous system with parallel communications and motions is challenging. It is especially complex to maintain $\mathcal{N}^K$ sets using only communications. A complete implementation that overcomes this challenge is presented in the next section.

5 C2SR Implementation

In this section, we provide a detailed implementation of C2SR[2]. Algorithm 1 shows the input and local variables of C2SR along with its initialization pseudo-code. Every module knows its position in the lattice, the goal shape, $\mathcal{G}$, and the rotation direction, d. Algorithm 2 describes some helper functions used in the description of our implementation of C2SR. Algorithm 3 provides the message handler pseudo-code of C2SR. Algorithm 4 gives the pseudo-code executed by a module after it finishes an atomic move. We assume that interrupts are disabled during message and event handler execution.

```
Input:
$p_{C_i}$ // position of $C_i$
$d \in \{CW, CCW\}$ // direction of rotation
$\mathcal{G}$ // goal shape
Local Variables:
$state$ // state of $C_i$
$Movings$ // cells from/into which a neighbor module is
    moving
$Pendings$ // pending clearance requests
$clearance$ // clearance for the current move (if any)

1  Initialization of $C_i$:
2  $Movings \leftarrow \emptyset$; $Pendings \leftarrow \emptyset$; $clearance \leftarrow \perp$;
3  if $p_{C_i} \in \mathcal{G}$ then
4      $state \leftarrow$ GOAL;
5  else if $isInStream()$ then
6      $state \leftarrow$ WAITING;
7      $requestClearance()$;
8  else
9      $state \leftarrow$ BLOCKED;
10 end
```

Algorithm 1: C2SR algorithm input, local variables and initialization detailed for any module C_i.

In our implementation, modules can have different states: BLOCKED, GOAL, WAITING or MOVING. WAITING and MOVING modules belong to the stream. At the initialization and during the execution, modules locally decide their state using Rules 1, 2 and 4. Modules in the stream move in rotation direction d around their peripheral neighbor in the d direction. Before moving, modules have to ensure that the stream progression rule (Rule 3) is satisfied. WAITING modules send CLEARANCE_REQUEST messages to get the authorization to move. Clearance requests are composed of the module source position and of its destination. These requests travel around the module destination cell. At each hop, modules check if the requested move satisfies the stream progression rule (see Algorithm 3, lines 1-24). If the stream progression rule is not satisfied the clearance request either has to be stored locally (see Algorithm 3, lines 6-9) or to be stored at the previous module using a DELAYED_CLEARANCE message (see Algorithm 3, lines 2-5, 11-14 and 32-35). If the stream progression rule is satisfied, the clearance is granted (see Algorithm 3, lines

[2]The complete source code is available online at https://github.com/claytronics/visiblesim

```
Function hasConverged():
    // The local convergence rule (Rule 4)
    return converged($C_i$);
end

Function areAdjacentCells($p_1, p_2$):
    return true if cells at positions $p_1$ and $p_2$ are adjacent in
      the hexagonal lattice, false otherwise;
end

Function opppositeDirection(d):
    // $d \in \{CW, CCW\}$
    return the opposite direction of $d$;
end

Function isFree():
    // The freedom of movement rule (Rule 1)
    return free($C_i$) considering both $\mathcal{N}^N_{C_i}$ and $Movings$;
end

Function isInStream():
    // The stream entrance rule (Rule 2)
    return stream($C_i$) considering both $\mathcal{N}^N_{C_i}$ and $Movings$;
end

Function getNeighbor(dir):
    return the peripheral neighbor in direction $dir$ (see Section
      2);
end

Function getNeighbor(dir, pos):
    return $C_k \in \mathcal{N}^N_{C_i}$ such that $C_i$ is connected to $C_k$ on the
      connected interface that immediately follows the interface
      pointing to position $pos$ in direction $dir$;
end

Function requestClearance():
    $C_k \leftarrow getNeighbor(d)$;
    $p'_{C_i} \leftarrow$ position after rotation in direction $d$ around $C_k$;
    $r \leftarrow (src \leftarrow p_{C_i}, dest \leftarrow p'_{C_i}, cnt \leftarrow 0)$;
    send CLEARANCE_REQUEST($r$) to $C_k$;
end

Function forwardClearance($c(src, dest), C_j$):
    if areAdjacentCells($c.src, p_{C_i}$) then
        $C_k \leftarrow getNeighbor(oppositeDirection(d), c.src)$;
        if $C_k \neq C_j \wedge areAdjacentCells(c.src, p_{C_k})$ then
            send CLEARANCE(c) to $C_k$;
        else
            $Movings \leftarrow Movings \cup \{c.src\}$;
            send CLEARANCE(c) to $C_l \mid p_{C_l} = c.src$;
        end
    else if areAdjacentCells($c.dest, p_{C_i}$) then
        $C_k \leftarrow getNeighbor(oppositeDirection(d), c.dest)$;
        send CLEARANCE($c$) to $C_k$;
    end
end

Function forwardEndOfMove($c(src, dest), C_j$):
    if areAdjacentCells($c.src, p_{C_i}$) then
        $C_k \leftarrow getNeighbor(oppositeDirection(d), c.src)$;
        if $C_k \neq C_j \wedge areAdjacentCells(c.src, p_{C_k})$ then
            send END_OF_MOVE($c$) to $C_k$;
        end
    else if areAdjacentCells($c.dest, p_{C_i}$) then
        $C_k \leftarrow getNeighbor(oppositeDirection(d), c.dest)$;
        send END_OF_MOVE($c$) to $C_k$;
    end
end
```

Algorithm 2: C2SR helper functions detailed for any module C_i.

```
When CLEARANCE_REQUEST($r(src, dest, cnt)$) is
received by $C_i$ from $C_j$ do:
if $state$ = WAITING then
    send DELAYED_CLEARANCE($r$) to $C_j$;
    return;
end
if $r.dest \in Movings$ then
    $Pendings \leftarrow Pendings \cup \{r\}$;
    return;
end
if $state$ = BLOCKED $\vee$ $state$ = GOAL then
    if $r.cnt = 3$ then
        send DELAYED_REQUEST($r$) to $C_j$;
        return;
    end
    $r.cnt \leftarrow r.cnt + 1$;
end
$C_n \leftarrow getNeighbor(d, r.dest)$;
if $C_n \neq C_j \wedge areAdjacentCells(p_{C_n}, r.dest)$ then
    send CLEARANCE_REQUEST($r$) to $C_n$;
else
    $c \leftarrow (r.src, r.dest)$;
    $Movings \leftarrow Movings \cup \{r.dest\}$;
    $forwardClearance(c, \perp)$;
end

When CLEARANCE($c(src, dest)$) is received by $C_i$ from
$C_j$ do:
if $c.src = p_{C_i}$ then
    $clearance \leftarrow c$;
    send START_TO_MOVE to $C_j$;
else
    $forwardClearance(c, C_j)$;
end

When DELAYED_CLEARANCE($r(src, dest, cnt)$) is
received by $C_i$ from $C_j$ do:
if $r.src \neq p_{C_i}$ then
    $Pendings \leftarrow Pendings \cup \{r\}$;
end

When START_TO_MOVE is received by $C_i$ from $C_j$ do:
send START_TO_MOVE_ACK to $C_j$;

When START_TO_MOVE_ACK is received by $C_i$ from $C_j$
do:
$state \leftarrow$ MOVING;
$C_k \leftarrow getNeighbor(d)$;
move around $C_k$ in direction $d$;

When END_OF_MOVE($c(src, dest)$) is received by $C_i$
from $C_j$ do:
$Movings \leftarrow Movings - \{c.src, c.dest\}$;
$forwardEndOfMove(c, C_j)$;
if $isInStream()$ then
    $state \leftarrow$ WAITING;
    $requestClearance()$;
else if $\exists r \in Pendings \mid r \in areAdjacentCells(r.dest, c.src)$
  then
    $C_n \leftarrow getNeighbor(d, r.dest)$;
    if $areAdjacentCells(r.dest, p_{C_n})$ then
        send CLEARANCE_REQUEST($r$) to $C_n$;
    else
        $cl \leftarrow (r.src, r.dest)$;
        $Movings \leftarrow Movings \cup \{cl.dest\}$;
        $forwardClearance(cl, \perp)$;
    end
end
```

Algorithm 3: C2SR algorithm message handler detailed for any module C_i.

```
1  When $C_i$ has finished to move do:
2  $p_{C_i} \leftarrow clearance.dest$;
3  send END_OF_MOVE(clearance) to $getNeighbor(d)$;
4  $clearance \leftarrow perp$;
5  if $hasConverged()$ then
6      $state \leftarrow$ GOAL;
7  else if $isInstream()$ then
8      $state \leftarrow$ WAITING;
9      $requestClearance()$;
10 end
```

Algorithm 4: C2SR algorithm event handler detailed for any module C_i.

20-24). The clearance is then progressively forwarded back to the module that initiated the request (see Algorithm 3, lines 25-31).

To prevent collision, modules maintain a list of neighbor cells from/into which a module is moving. After having moved to a new position, modules send an END_OF_MOVE (EOM for short) message that is progressively forwarded around the cell of their previous position (see Algorithm 4, line 3 and Algorithm 3, lines 42-57). Upon reception, of an EOM message, delayed clearances are potentially re-activated (see Algorithm 3, lines 48-57).

START_TO_MOVE and START_TO_MOVE_ACK messages guarantee that no message is lost when a module decides to actually move (see Algorithm 3, lines 36-41).

Modules never need to communicate with modules farther than two cells away in the lattice, which means that, due to our requirements, modules never need to send messages that have to travel more than five hops. Thus, our algorithm uses only local interactions between modules.

6 Experimental Evaluation

We implemented C2SR in C++ and evaluated it using VisibleSim [12], a simulator for modular robots. This section presents our experimental results. Through our experiments, we show the effectiveness of C2SR and its efficiency in terms of communications, movements and execution time.

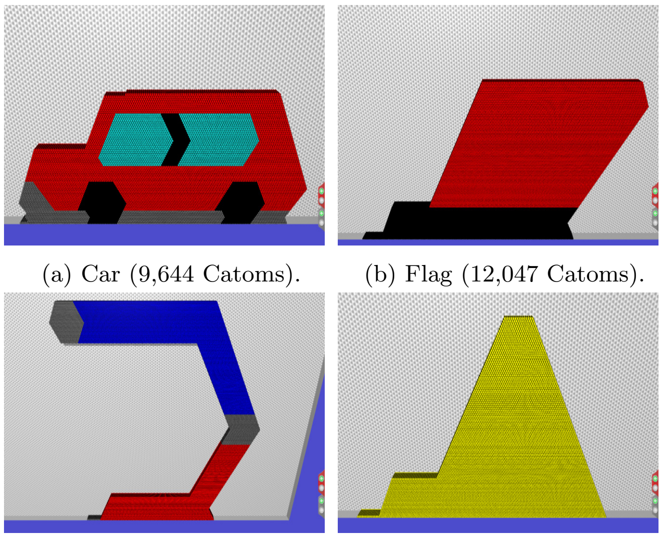
VisibleSim enables one to perform simulations with different and variable motion and communication delays. In our evaluation, we assume that neighboring modules communicate together using 8-N-1 serial communications. Hence, we assume the effective bit-rate is equal to 80% of the link bit-rate. We assume the effective average communication bit-rate between two neighboring modules follows a Gaussian distribution. Moreover, we assume the average motion speed during atomic moves of a 2D Catom also follows a Gaussian distribution. We do not simulate delays due to processing and interruptions because we assume them to be negligible in comparison to communication and motion delays.

Unless explicitly mentioned, we assume the following simulation parameters. We consider the effective average communication bit-rate during message exchanges between two neighboring modules has a distribution centered on 38.9 $kbps$ with a standard-deviation of 389 bps (1% of the mean). Moreover, we assume the average motion speed during atomic moves of a module has a distribution centered on 1.88 $mm \cdot s^{-1}$ with a standard-deviation of 0.0188 $mm \cdot s^{-1}$ (1% of the mean).

We evaluate C2SR on the self-reconfiguration of random clumps of 2D Catoms into four kinds of shapes, namely a car, a flag, a magnet and a pyramid shape (see Figures 1 and 7). For each target shape, we generated different versions of the goal configurations using different scales ranging from a dozens to ten thousands of modules.

6.1 Effectiveness Evaluation

As shown in Figure 7, C2SR is able to self-reconfigure ensembles composed of more than 10,000 2D Catoms.



(a) Car (9,644 Catoms). (b) Flag (12,047 Catoms).

(c) Magnet (10,220 Catoms). (d) Pyramid (8,033 Catoms).

Figure 7: Screenshots of VisibleSim at the end of the simulation of C2SR with different kinds of goal shapes composed of about 10,000 2D Catoms.

6.2 Communication Evaluation

Figure 8 shows the total number of messages sent during the execution of C2SR according to the size of the goal shape. For the shapes we considered, the number of messages seems to depend on the size of the goal configuration and not on the actual shape of the arrangement. Moreover, the standard-deviation is very small, so small, that it is not visible on the figure. Thus, for a goal shape of a given size, C2SR always sends approximately the same number of messages. Furthermore, as shown in Figure 8 by the curve of best fit $y(x) = 20.29x^{1.53}$, this number of messages is highly predictable and increases polynomially with the size of the goal shape.

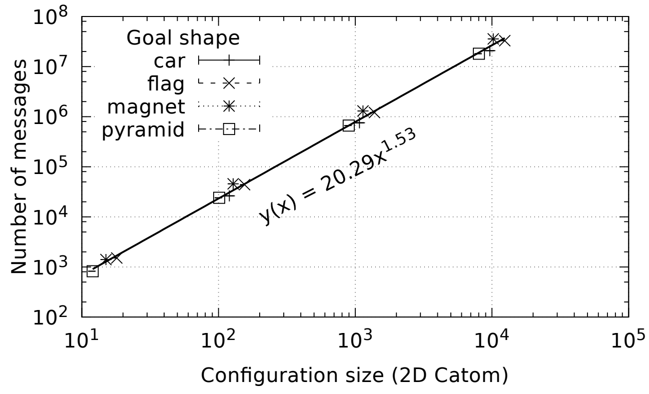


Figure 8: Average total number of messages (± standard-deviation) versus the size of the system for different goal shapes. For each point, 10 executions were performed.

Figure 9 indicates that a few modules tend to send a lot more messages than the other modules. Intuitively, modules that stay at the boundary between $\mathcal{I}$ and $\mathcal{G}$ are communication hotspots because many modules have to communicate with them before rolling over them in order to reach $\mathcal{G}$ (see Figure 13).

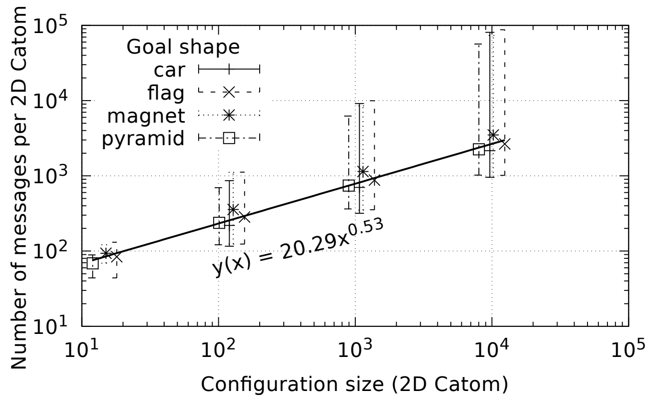


Figure 9: Average number of messages sent per 2D Catom (± min/max) versus the size of the system for different goal shapes. For each point, 10 executions were performed.

Figure 10 shows the maximum message queue size reached by the modules during the execution of C2SR, taking into account both the incoming and the outgoing messages. The maximum message queue size is constant and equal to two regardless of the shape of the goal configuration and regardless of its size. We recall that messages generated by C2SR have a small and constant size. As a consequence, the traffic generated by C2SR is well controlled and modules do not require a lot of memory space to store incoming and outgoing messages.

Figure 11 shows the average number of hops traveled by the packets during the execution of C2SR. The average and the maximum number of hops traveled by the packets is small and relatively constant regardless of the shape of the goal configuration and regardless of its size. This confirms that C2SR only involves local interactions, as announced in the previous section.

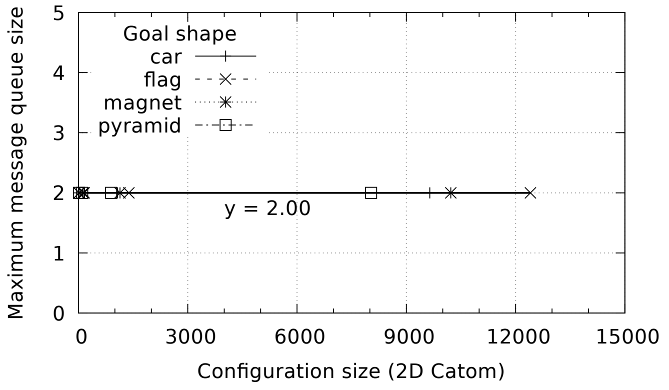


Figure 10: Maximum reached message queue size (incoming and outgoing messages) versus the size of the system. For each point, 10 executions were performed.

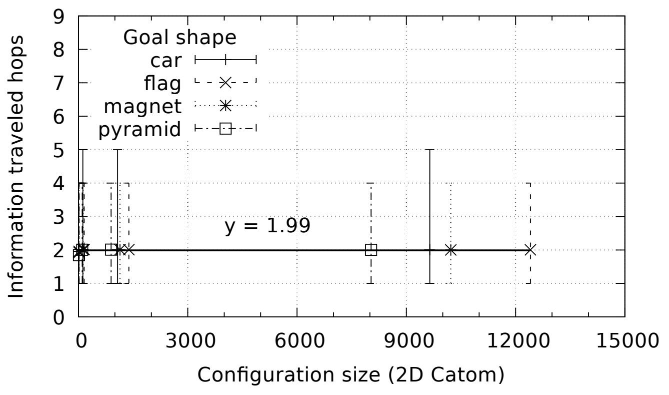


Figure 11: Average number of hops data have traveled (± min/max) versus the size of the system. For each point, 10 executions were performed.

6.3 Motion Efficiency

Figure 12 shows the total number of atomic moves performed during the execution of C2SR according to the size of the system for different goal shapes. Note that this figure is really similar to Figure 8. Here again, the number of atomic moves seems to only depend on the size of the goal configuration and not to the actual shape of the arrangement. As shown in Figure 12 by the curve of best fit $y(x) = 2.09x^{1.53}$, the number of atomic moves is highly predictable and increases polynomially with the size of the goal shape. Notice that the number of messages is approximately equal to ten times the number of moves (see Figures 8 and 12). Thus, an atomic move requires in average 10 messages.

As shown in Figure 13, many modules can move concurrently during the execution of C2SR. Thus, although the self-reconfiguration process may require many atomic moves, it remains reasonably time efficient as shown in the next subsection.

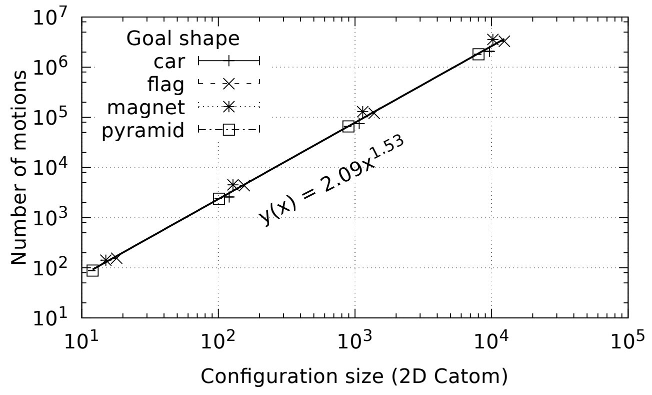


Figure 12: Average total number of atomic moves ($\pm$ standard-deviation) versus the size of the system for different goal shapes. For each point, 10 executions were performed.

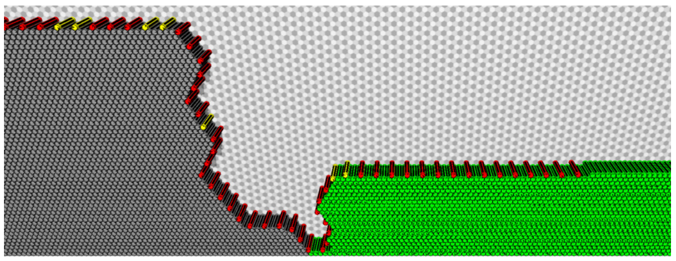


Figure 13: Screenshot of VisibleSim during a self-reconfiguration process. Modules in the stream progress by rotating CW. Blocked modules are in gray, waiting ones in yellow, moving ones in red and modules that have converged are in green.

6.4 Execution Time Efficiency

Figure 14 shows the average simulated time of C2SR execution according to the size of the system. For the different goal shapes we considered, this time seems to only depend on the size of the configuration and not to the actual shape of the arrangement. Moreover, the standard-deviation is very small and not visible on the figure. Thus, for goal shape of a given size, C2SR always approximately lasts the same duration. As shown in Figure 14 by the curve of best fit $y(x) = 0.017x + 0.149$, the simulated time is highly predictable and increases linearly with the size of the goal shape. The slope of the line gives the reconfiguration speed: C2SR fills on average $\frac{1}{0.017} \approx 59$ goal cells per minute.

Figure 15 shows the average simulated time of C2SR execution according to the average communication bit-rate for the two different motion speeds supported by the 2D Catoms. We consider the usual bit-rates of serial communications. We conducted this experiment for the car goal shape composed of 1,073 modules. Until 38.9 *kbps*, the self-reconfiguration process becomes much more faster as the average communication bit-rate increases. Beyond 38.9 *kbps*, the self-reconfiguration speed increases less quickly and tends to stabilize.

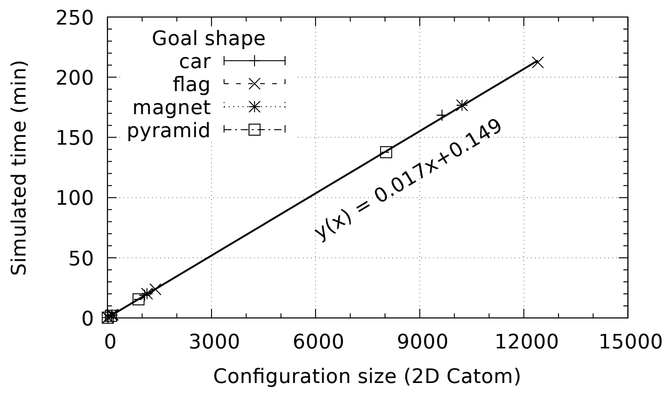


Figure 14: Average simulated time ($\pm$ standard-deviation) versus the size of the system for different goal shape. For each point, 10 executions were performed.

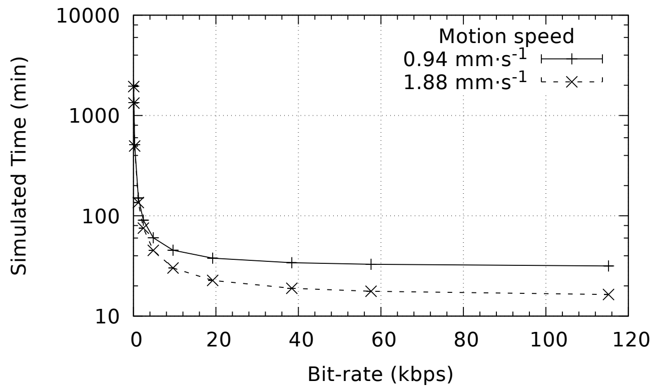


Figure 15: Average simulated time ($\pm$ standard-deviation) versus the communication bit-rate (random initial configuration to the car of $1,073$ 2D Catoms). For each point, 10 executions were performed.

7 Conclusion

We have proposed Cylindrical-Catoms Self-Reconfiguration (C2SR), a parallel, asynchronous and fully decentralized distributed algorithm to self-reconfigure lattice-based MSR from an initial shape to a goal one.

The evaluation of C2SR has been conducted with real executions under a simulated physical environment (VisibleSim). These simulations show our algorithm to have nice properties.

The time for reconfiguration is linear in the number of modules and this time is predictable and seems to only depends on the number of modules. The number of messages sent is also predictable such that added with the number of movements, it can give an estimate of the power consumption of the algorithm. Communications are local such that no routing protocol is needed and the message queue of each module is always bounded by two on our simulation. The needed bandwidth is reasonable, as it uses less than 40 *kbps* on one example without slowing down the reconfiguration process.

8 Future Work

In future works, we will demonstrate the correctness of C2SR, i.e., we will prove that the goal configuration can be built if the shape admissibility conditions are satisfied. Moreover, we will study the performance of C2SR on other types of shapes and compare it to existing algorithms. We will also study the distribution of both the number of messages sent per module and the number of atomic moves performed per module. Our observations seem to indicate that our algorithm is highly predictable and that its execution time is linear to the size of the goal shape. A further step would be to prove it. Furthermore, we would like to reduce the memory usage of our algorithm induced by the storage of the goal shape representation. Indeed, hardware modules have limited memory capacity and cannot afford to store the complete representation of large goal shapes [3, 20, 19]. We envision two approaches to address this storage limitation, namely to use a compressed representation of the goal shape and/or to disseminate and share the representation of the goal shape between all modules [3].

Acknowledgments

This work has been funded by the Labex ACTION program (contract ANR-11-LABX-01-01) and ANR/RGC (contracts ANR-12-IS02-0004-01 and 3-ZG1F) and ANR (contract ANR-2011-BS03-005).

References

[1] Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007.

[2] Seth Copen Goldstein and Todd C. Mowry. Claytronics: An instance of programmable matter. In *Wild and Crazy Ideas Session of ASPLOS*, Boston, MA, October 2004.

[3] Julien Bourgeois, Benoit Piranda, André Naz, Hicham Lakhlef, Nicolas Boillot, Hakim Mabed, Dominique Douthaut, and Thadeu Tucci. Programmable matter as a cyber-physical conjugation. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Budapest, Hungary, October 2016. IEEE.

[4] Hicham Lakhlef, Hakim Mabed, and Julien Bourgeois. Distributed and dynamic map-less self-reconfiguration for microrobot networks. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 55–60. IEEE, 2013.

[5] M. Park, S. Chitta, A. Teichman, and M. Yim. Automatic configuration methods in modular robots. *International Journal for Robotics Research*, 27(3-4):403–421, March/April 2008.

[6] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, 1991.

[7] Feili Hou and Wei-Min Shen. Graph-based optimal reconfiguration planning for self-reconfigurable robots. *Robotics and Autonomous Systems*, 62(7):1047 – 1059, 2014.

[8] Kasper Stoy and Haruhisa Kurokawa. Current topics in classic self-reconfigurable robot research. In *Proceedings of the IROS Workshop on Reconfigurable Modular Robotics: Challenges of Mechatronic and Bio-Chemo-Hybrid Systems*, 2011.

[9] Benoit Piranda and Julien Bourgeois. A distributed algorithm for reconfiguration of lattice-based modular self-reconfigurable robots. In *PDP 2016, 24th Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing*, pages 1–9, Heraklion Crete, Greece, feb 2016. IEEE.

[10] Mustafa Emre Karagozler, Seth Copen Goldstein, and J. Robert Reid. Stress-driven mems assembly + electrostatic forces = 1mm diameter robot. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '09)*, October 2009.

[11] Mustafa Emre Karagozler. *Design, Fabrication and Characterization of an Autonomous, Sub-millimeter Scale Modular Robot*. PhD thesis, Carnegie Mellon University, 2012.

[12] Dominique Dhoutaut, Benoît Piranda, and Julien Bourgeois. Efficient simulation of distributed sensing and control environments. In *iThings 2013, IEEE Int. Conf. on Internet of Things*, pages 452–459, Beijing, China, August 2013.

[13] Benoit Piranda. Visiblesim: Your simulator for programmable matter. In Sándor Fekete, Andréa Richa, Kay Römer, and Christian Scheideler, editors, *Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271), May 2016.*

[14] Stanislav Funiak, Padmanabhan Pillai, Michael P. Ashley-Rollman, Jason D. Campbell, and Seth Copen Goldstein. Distributed localization of modular robot ensembles. *International Journal of Robotics Research*, 28(8):946–961, 2009.

[15] Moffo Dermas, Philippe Canalda, and François Spies. First evaluation of a system of positioning of microrobot with ultra-dense distribution. In *IPIN 2016, International Conference on Indoor Positioning and Indoor Navigation*. IEEE, October 2016.

[16] Jennifer E Walter, Jennifer L Welch, and Nancy M Amato. Distributed reconfiguration of metamorphic robot chains. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 171–180. ACM, 2000.

[17] Jennifer E Walter, Elizabeth M Tsai, and Nancy M Amato. Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. *IEEE transactions on Robotics*, 21(4):621–631, 2005.

[18] J Bateau, A Clark, K McEachern, E Schutze, and J Walter. Increasing the efficiency of distributed goal-filling algorithms for self-reconfigurable hexagonal metamorphic robots. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, pages 509–515, 2012.

[19] Hicham Lakhlef, Julien Bourgeois, Hakim Mabed, and Seth Copen Goldstein. Energy-aware parallel self-reconfiguration for chains microrobot networks. *Journal of Parallel and Distributed Computing*, 75:67–80, 2015.

[20] Hicham Lakhlef and Julien Bourgeois. Fast and robust self-organization for micro-electro-mechanical robotic systems. *Computer Networks*, 93:141–152, 2015.

[21] Stanton Wong and Jennifer Walter. Deterministic distributed algorithm for self-reconfiguration of modular robots from arbitrary to straight chain configurations. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 537–543. IEEE, 2013.

[22] S Wong, S Zhu, and J Walter. Unpacking a cluster of modular robots. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 103. WorldComp, 2015.

[23] Michael De Rosa, Seth Goldstein, Peter Lee, Jason Campbell, and Padmanabhan Pillai. Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1462–1468. IEEE, 2006.

[24] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[25] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3293–3298. IEEE, 2012.