ASAPs: asynchronous hybrid self-reconfiguration algorithm for porous modular robotic structures

Jad Bassil, Benoit Piranda, Abdallah Makhoul, Julien Bourgeois

ASAPs: Asynchronous Hybrid Self-Reconfiguration Algorithm for Porous Modular Robotic Structures

Jad Bassil[1*], Benoît Piranda[1], Abdallah Makhoul[1] and Julien Bourgeois[1]

[1*]Univ. Franche-Comté, FEMTO-ST institute, CNRS, Cr Louis Leprince-Ringuet, Montbéliard, 25200, Bourgogne Franche-Comté, France.

*Corresponding author(s). E-mail(s): jad.bassil@femto-st.fr;
Contributing authors: benoit.piranda@femto-st.fr; abdallah.makhoul@femto-st.fr;
julien.bourgeois@femto-st.fr;

Abstract

Programmable matter refers to material that can be programmed to alter its physical properties, including its shape. Such matter can be built as a lattice of attached robotic modules, each seen as an autonomous agent with communication and motion capabilities. Self-reconfiguration consists in changing the initial arrangement of modules to form a desired goal shape, and is known to be a complex problem due to its algorithmic complexity and motion constraints. In this paper, we propose to use a max-flow algorithm as a centralized global planner to determine the concurrent paths to be traversed by modules through a porous structure composed of *3D Catoms* meta-modules with the aim of increasing the parallelism of motions, and hence decreasing the self-reconfiguration time. We implement a traffic light system as a distributed asynchronous local planning algorithm to control the motions to avoid collisions. We evaluated our algorithm using *VisibleSim* simulator on different self-reconfiguration scenarios and compared the performance with an existing fully distributed synchronous self-reconfiguration algorithm for similar structures. The results show that the new method provides a significant gain in self-reconfiguration time and energy efficiency.

Keywords: Self-Reconfiguration, Modular Robots, Porous Structure, Max-Flow

1 Introduction

Programmable matter is a matter that can modify its physical properties, such as its shape, according to the surrounding environment. Although, this property can be obtained by different means, we are interested in building this matter with a large number of autonomous computing particles or modules, communicating locally and self-organizing into the required collective behavior. Particles can move around each other by altering their physical form, detaching, and forming new connections with each other to transform from an initial shape to a goal shape. This is called the self-reconfiguration problem [34]. The difficulty of this problem lies in the properties of a limited memory space and the locality of the information. Indeed, due to limited space, individual modules or particles have very low computational and energetic resources. Furthermore, due to the locality of information, a module does not have the information about the global configuration of the modular robot system, and then it cannot take decisions individually. Moreover, having mobile and connected robots is insufficient to

obtain self-reconfigurable programmable matter. A subset of these robots must move while avoiding collisions and maintaining network connectivity. Therefore, self-reconfiguration is a very complex problem, since the number of possible configurations increases exponentially as the size of the system (number of particles) increases [6].

The self-reconfiguration time, that is, the time required to transform an initial shape into a goal shape, is an important parameter that must be optimized. Sequential algorithms [11, 14] where the reconfiguration is performed one module at a time are impractical due to their slow speed, especially when the number of modules is large. Two main optimizations are possible to reduce the time complexity of this problem. They consist of reducing the moving distance of modules and permitting the simultaneous (parallel) displacement of a large number of modules in the system [15, 19, 35].

In addition, the modules have limited energy resources. They can be powered through an external source or by utilizing energy harvesting techniques from their environment, such as ambient light or electromagnetic sources. Optimizing energy consumption, employing energy-efficient algorithms is essential for the functionality of the modular robot. Therefore, our proposed algorithm aims to reduce energy consumption during self-reconfiguration by reducing the number of motions executed by the modules.

The potential applications of this technology span diverse fields, including interactive CAD design [6], flexible tangible interfaces [28] and shape-shifting multi-purpose objects that can change their functionality on demand.

Meta-modules are formed by locking together multiple modules, functioning cohesively as a single unit to alleviate motion constraints and facilitate self-reconfiguration planning. In [2] we proposed to use a porous structure composed of meta-modules as a scaffold that can then be coated to better represent the goal shape. The meta-modules groups quasi-spherical robotic modules called *3D Catoms* that move by rolling on their neighbors. The use of a porous structure where modules are regularly arranged reduces the density of the modular robotic ensemble, allowing the modules to flow freely through the internal empty volume following precalulated deterministic and parallel motion paths. We then, proposed *RePoSt* a synchronous round-based self-reconfiguration algorithm that, given the goal configuration in the form of a constructive solid geometry tree [38], it finds in each round a set of disjoint paths that connect meta-modules that do not belong to the goal shape to empty positions that must be filled.

In this work, assuming that the initial and goal configurations are known, we propose *ASAPs* a hybrid centralized/distributed self-reconfiguration algorithm consisting of a centralized global planner that computes concurrent paths in an initial phase. The modules then execute a distributed asynchronous motion coordination algorithm to flow towards their goal position. By "centralized", we refer to a system wherein a single entity, in this case, the global planner, takes the lead in computing paths given the whole meta-modules inter-connections graphs of the initial and goal configurations. Conversely, when we mention "distributed," we imply a system where tasks or decisions are handled by individual modules in a coordinated manner. We compared *ASAPs* to *RePoSt* and the results show that *ASAPs* provides a significant improvement in performance.

The remainder of the paper is organized as follows. Section 2 presents the related work that influenced this paper. Section 3 presents the *3D Catom*: the modular robotic system that we are using. Section 5 describes the global centralized planner and the distributed motion control algorithm of *ASAPs*. Section 6 analyzes the complexity of *ASAPs*. Section 7 presents the conducted simulations, and analyzes the results while comparing the performance of *ASAPs* with *RePoSt*. Finally, the paper is concluded and perspectives are presented in Section 8.

2 Related Work

Modular robots can be classified into multiple types of structural formation. Chain-type formation consists of modules arranged in a tree-like fashion [7, 29, 41–43]. They are able to perform locomotion gait on rough terrains. In a lattice-type formation, the modules reside in a regular 2D or 3D lattice structure [12, 13, 23, 25, 33, 39]. Modules in a lattice-type modular robot are assigned a unique coordinate value that a planner can exploit for efficient self-reconfiguration. Some modular robots exhibit a combination of chain-type and

lattice-type formations, combining the characteristics of both types, they form the hybrid-type [18, 20, 22, 30]. Lattice-type modular robots are more suitable for creating a programmable matter since they allow more flexibility in approximating a given shape.

The primary challenge to be addressed for a modular robot-based programmable matter to successfully execute its tasks is self-reconfiguration. Self-reconfiguration consists of performing module-level movements to change an initial configuration into a goal one. The authors in [1, 9, 34] provide surveys on self-reconfiguration planning methods. A self-reconfiguring solution that uses sequential movements such as in [11, 14] motion to achieve a target shape simplifies planning by eliminating potential problems such as dealing with deadlocks and collision uncertainties. However, sequential motions are highly restrictive in medium-to-large self-reconfiguring modular robots, as they tend to significantly lengthen the duration of the reconfiguration process. Therefore, parallelism of movements is required. Movements through the internal volume of the robot allow for a higher degree of parallelism and require a smaller number of movements to reach the goal configuration according to [29]. Internal movements can be achieved using tunneling or/and scaffolding.

Tunneling allows in-place self-reconfiguration of modular robots where modules flow in parallel within its internal volume. Many tunneling-based algorithms exist [8, 15, 16, 19, 40]. They require a specific modular robotic hardware design that consists of modules with simple cubic geometries and uses actuators to perform translation and rotation motions. They use meta-modules as the building unit of the structure to facilitate planning and ease motion constraints.

Scaffolding first proposed in [17] is another technique used to to optimize the self-reconfiguration process. It consists of building the structure using hollow sub-structures or meta-modules leaving enough empty volume inside the structure that allows modules to navigate through it in parallel while avoiding blocking and collisions due to overcrowding at the cost of decreasing the granularity of the configuration. Scaffolding is then used to reduce the complexity of the reconfiguration of cubic modules that moves by translation and rotation guided by cellular automata in [31] or gradient descent in [32] by approximating the target configuration with a porous representation.

Thalamy et al. proposed a self-reconfiguration scheme for modular robotic programmable matter using the same hardware that we are using in this paper: the *3D Catoms*. It envisions assembling the scaffold of a shape using multi-module tiles. The tiles are built with modules that flow upward from a reserve of modules placed beneath the shape. The shape can then be coated with a thin layer of modules to better represent the target shape [36]. Flowing modules use a local message passing local coordination algorithm inspired by the traffic light system that forces modules to keep enough empty space between them to avoid collisions. We use the same coordination and collision avoidance method when modules are flowing while executing the operations as explained in section 4. The difference from our work is that they describe only the assembling of a shape starting from a reserve of modules placed beneath it, not the self-reconfiguration of an initial shape into a goal one.

Lengiewicz and Holobut [19] presented a method to self-reconfigure large ensembles of cubic modules that form a porous scaffolding structure made of cubic meta-modules of 7 modules. They tackled the self-reconfiguration problem by decomposing it into two subproblems: determining how the boundary of the current configuration must evolve to reach the goal configuration and finding an optimal flow of modules between the boundaries of the current shape and through its volume using an asynchronous distributed max-flow search based on local memory and communication. Their proposed algorithm is efficient, with the number of movements of the modules proportional to the resolution of the robot. This method might be used with any other hardware system that has the ability to internally move modules through a scaffolding setup.

In a previous work, we proposed *RePoSt* [3] a fully distributed round-based self-reconfiguration algorithm based on local communication and local memory. In each round, a set of streamlines is determined that connects meta-modules that do not belong to the goal shape to empty positions in the target shape. Modules flow along the streamlines to be assembled in an empty position in

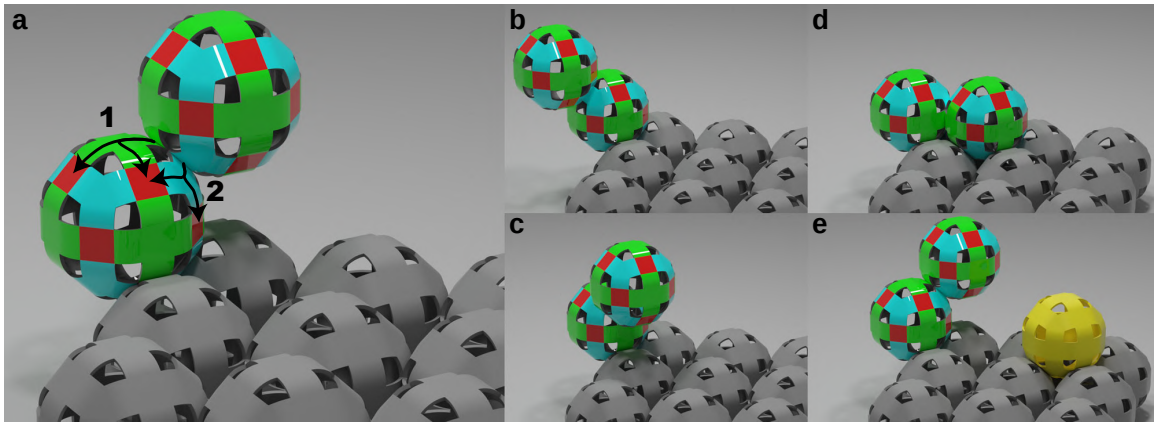


Fig. 1: Motion capabilities of the *3D Catom*. a) Arrows #1 and #2 shows the two kind of rotations respectively along green surface and blue surface. b) Shows the final position of the top *3D Catom* after a rotation along arrow #1. c) Shows the final position of the top *3D Catom* after a rotation along arrow #2. In figure d) the final position of the top *3D Catom* is reachable but the same position is not reachable in figure e) due to yellow module.

the target shape determined by a distributed goal shape description using a constructive solid geometry tree [38]. Streamlines are disjoint and are found using the distributed max-flow search algorithm proposed in [19]. The time complexity of *RePoSt* is $O(M.d)$ where M is the number of rounds bounded by the number of meta-modules and d is the length of the longest streamline.

The evaluation results of *RePoSt* showed that the predominant factor influencing self-reconfiguration time is the duration required for modules to reach their target positions in comparison to the time spent on computations and communications for path determination. Consequently, we aim to increase the parallelism of motions at the cost of additional centralized computation which is fast relative to modules movements. In this work, we propose a hybrid centralized/distributed algorithm that precalculates the global max-flow between the initial and goal configuration a priori in a centralized planner. Then, modules flow in parallel to fill the goal configuration in a single round decreasing the complexity to $O(d)$ as explained in Section 6.

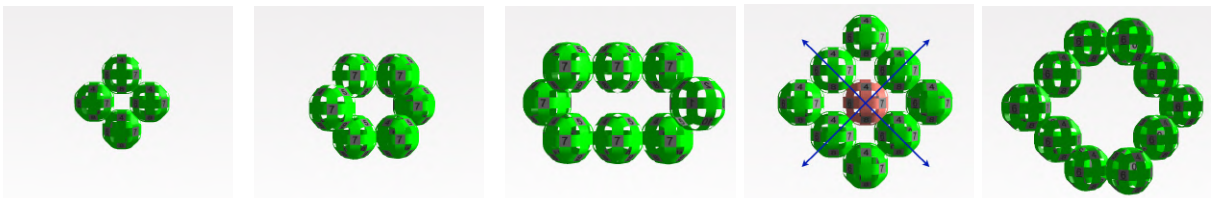
Our approach differs from the distributed max-flow method proposed in [19] by implementing a centralized max-flow search on a graph that represents the combined initial and goal configurations. This approach allows to find a set of overlapping paths that modules can follow simultaneously to reach the goal configuration, thereby increasing flow and reducing self-reconfiguration time at the cost of using a centralized global planner instead of a fully distributed one.

3 Modular Robotic System

In this paper, we consider the problem of self-reconfiguration of a modular robot composed of *3D Catoms*. *3D Catoms* where first proposed by [24]. They are 3,6 mm-diameter quasi-spherical modules residing in a face-centered cubic lattice as shown in Fig.1. A *3D Catom* can be attached to up to 12 neighbors using electrostatic actuators on their surface colored in red in Fig.1. They can communicate using message-passing through their latching interfaces.

A *3D Catom* moves by rotating on the surface of a fixed neighbor acting as a pivot (cf. Fig.1). The rotation of a *3D Catom* is subject to the following constraints:

- Collision constraint: no more than one *3D Catom* should be moving to an empty position to avoid collisions.
- Bridging constraint: a *3D Catom* cannot enter a free position that has two occupied positions in opposite directions as shown in Fig. 1e.
- Blocking constraint: a *3D Catom* entering a free position must not be blocked by other *3D Catoms*.



(a) (b) (c) (d) (e)

Fig. 2: Different meta-module sizes. (a) 4 modules, (b) 6 modules, (c) 8 modules, (d) 8 modules with a red module in a blocked position due to the bridging constraint caused by the green modules on the blue axis and (e) 10 modules.

The *3D Catoms* modular robot forms a distributed system where:

- Communications are done in a local fashion. A *3D Catom* can only communicate with its directly connected neighbors by sending messages through their latching interfaces using electrostatic signals.
- All *3D Catoms* share the same coordination system, in [27] Piranda et al. propose a distributed algorithm to set coordinates to a set of connected modules. They store their coordinates in their local memory and update them after each position change.
- The interconnections graph must be connected all the time.
- All *3D Catoms* execute the same distributed program and perform their computations locally.
- *3D Catoms* can be programmed to react to an internal event once detected, such as the reception of a message, a rotation end, a disconnection of a neighbor, etc.

4 Porous Structure Anatomy

In this section we describe the anatomy of the porous structure first proposed in [2] on which we apply our self-reconfiguration algorithm.

The porous structure is made up of hexagonal shaped meta-modules formed with 10 *3D Catoms* placed in a face-centered cubic lattice. The size 10 is the smallest size that allow modules to flow through the empty volume of the meta-module and to keep enough space between adjacent meta-modules for modules to flow between them without blocking and collisions. The selection of size 10 for the meta-modules was determined through an iterative process involving the evaluation of various sizes. This assessment included manual and visual checks in *VisibleSim* simulator [37] to ensure modules could flow through the internal volume of the meta-module and allow inter-meta-modules movements while satisfying the blocking and the bridging constraints mentioned in Section 3. Fig. 2 shows smaller meta-modules with sizes four, six and eight. It could be seen in Fig.2 (d) that considering a meta-module with eight modules, if we place a module (shown in red) in the center, it is in a blocked position due to the bridging constraint and cannot pass through. At least two modules must be added at positions along the two axis shown in the figure to avoid the bridging condition which sum up to ten modules. We also checked that a symmetrical meta-module with nine modules is not possible. Therefore, the size ten is the smallest possible size that allows modules flow through the interior volume.

A meta-module can be in two states, sparse or full. The full meta-module is filled into its empty internal volume with additional 10 *3D Catoms*. The meta-modules are arranged in a 3D regular cubic lattice as shown in Figure 3. Each meta-module is connected to its neighbors with at least one module. The left, right, back and front meta-modules have their modules positions flipped horizontally. The bottom and top meta-modules are attached to the front or back of the two upper or lower modules according to the $\overrightarrow{Z}$ axis to preserve the symmetry of the structure and facilitate the movements of the modules between meta-modules.

Each meta-module can execute the following three operations in the six directions ($\pm\overrightarrow{X}$, $\pm\overrightarrow{Y}$,

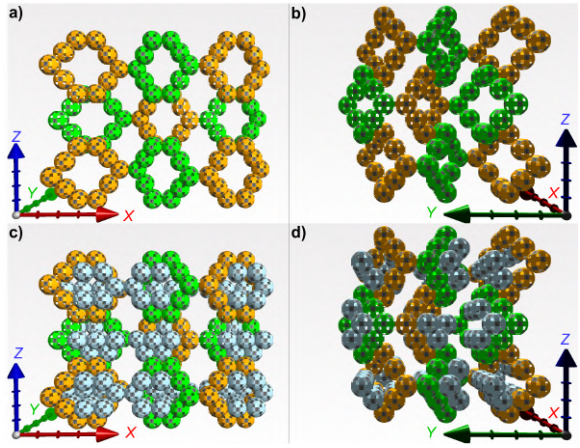


Fig. 3: The porous structure in a 3D cubic lattice. a) meta-modules in the XZ plane. b) meta-modules in YZ plane. c) Full meta-modules in the XZ plane. d) Full meta-modules in YZ plane.

$\pm\overrightarrow{Z}$) in the cubic meta-module scale lattice as shown in Fig.4:

1. *Dismantle* operation to break the meta-module and transport its composing module to a next meta-module in any given direction.
2. *Transfer* operation transports the modules through the empty internal volume of the structure to a neighbor meta-module.
3. *Build* operation to build a meta-module at an empty position.

An operation is defined as a sequence of hand-coded movements to transport the modules from a starting position to a target one. Each movement is coded as a triplet in the form of $< current\_position, next\_position, state >$. *state* can take three possible values:

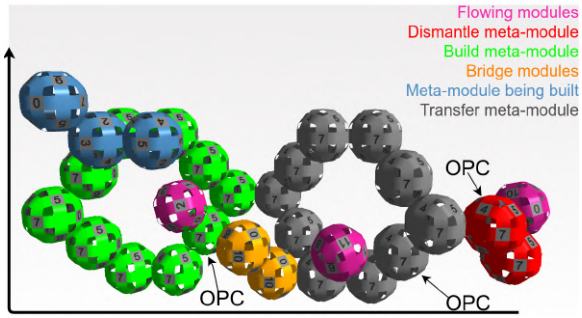


Fig. 4: An example of operations execution (best viewed in color).

1. MOVING indicates that the module must keep moving after *next_position* is reached.
2. WAITING indicates that after the current movement the module must wait and serve as a bridge for next modules.
3. IN_POSITION indicates that the module will reach its target position for the current operation.

Sequences of movements for each operation are predefined and stored in each robot memory. The number of movements and memory for each operation is shown in [3]. In total, 2,24 kB of memory is needed [3]. A special module is designated in a meta-module, as described in Section 5.2, to serve as an operation coordinator (OPC) to indicate the sequence of movements that a module must perform for the current operation. Fig. 4 shows an example of the execution of operations on three meta-modules where the red meta-module performs the dismantle operations in the left direction, the grey meta-module performs the transfer operation in the left direction and the green meta-module performs the build operation in the upward direction. The video[1] shows in its first segment the operations in execution.

These operations can be used to transform any initial connected configuration of meta-modules into a goal configuration. The purpose of a self-reconfiguration planner is to specify which operation to execute and in which direction to transform an initial configuration to a goal one.

5 Algorithm Description

In order to transform an initial configuration I into a goal configuration G, we consider 3 different groups of meta-modules: Meta-modules that are in the initial configuration but not in the final one ($I \setminus G$), meta-modules present in both configurations ($I \cap G$), and others that are in $G \setminus I$.

Meta-modules in $I \setminus G$ must be dismantled, and their composing modules must flow inside the structure to fill empty positions in $G \setminus I$ by building new meta-modules.

Fig. 5 shows the general flow of the algorithm. Given the initial and goal configurations, a global planner will perform a centralized computation that finds the flow paths of the modules from $I \setminus G$

[1] https://youtu.be/MUfuY0ao-0w

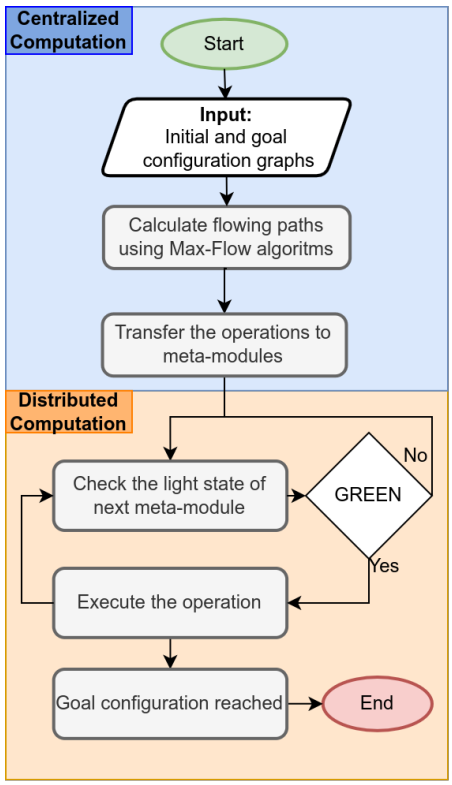


Fig. 5: The general flow of the algorithm. The top section describes the global centralized planning. The bottom section describes the distributed flow algorithm of the modules.

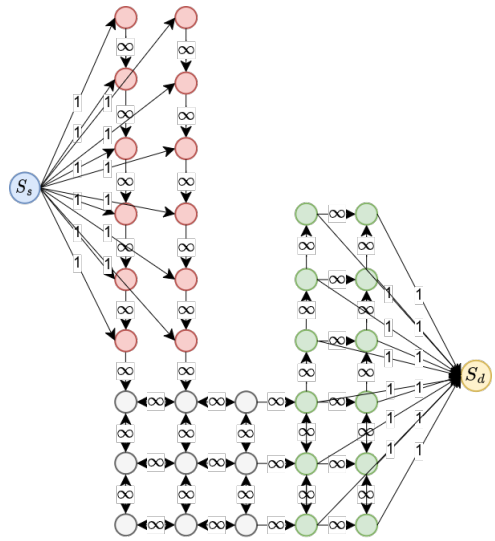
to $G \backslash I$ in an initialization phase using a Max-Flow algorithm. The resultant flowing paths will allow to specify which operation to execute on each meta-modules. Then, the operations are transferred to their respective meta-modules on flowing paths. Once the operations are assigned, the modules execute a distributed algorithm based on the traffic light system that controls the flow of the modules on concurrent paths without collisions.

In this section, we first describe a global planning algorithm based on the Max-Flow search to determine the operations to execute on each meta-module and in which direction. Then we describe an asynchronous distributed algorithm to control modules' flow on the paths without collisions.

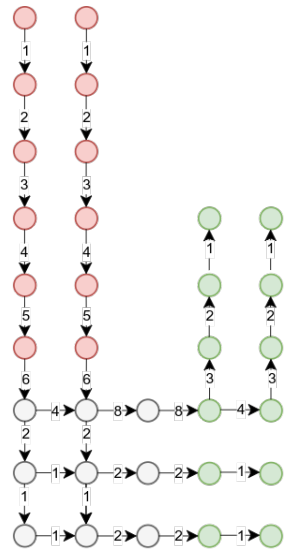
5.1 Global Planning

Given an initial configuration I and a goal configuration G, a global planner must specify the operations to execute on each meta-module. Once the operations are determined, they are communicated through neighbor-to-neighbor connections, following a spanning tree rooted at the central station.

The global planner is a centralized process that is executed in an initialization phase. The



(a) $\mathcal{G} = G \cup I$ construction example. Nodes in R_s are colored in red. Nodes in R_d are colored in green. Nodes in $R_s \cap R_d$ are colored in light gray. The label on each edge is the capacity of that edge.



(b) The resultant flow after applying the max-flow algorithm.

Fig. 6: Graph construction (a) and flow result (b).

global planner is implemented using a central computing station connected to at least one of the meta-modules. Since modules have limited knowledge about their configuration graph, they must collaborate to send a representation of their current configuration to the global planner. This can be done by each of the meta-modules sending their position to the global planner via a convergecast on a spanning-tree rooted at the global planner, which can be costly, or by a more efficient distributed shape recognition algorithm. In [5], we proposed an algorithm that allow modules to determine a representation of their current shape by finding distributedly a set of overlapping boxes which they can report to the global planner to build the inter-connections graph of the initial configuration.

The meta-modules configuration can be represented as a lattice graph in which the nodes represent meta-modules and the edges represent the connections between adjacent meta-modules. The global planner starts by constructing a graph $\mathcal{G}$ representing $I \cup G$ (whole space reached by the reconfiguration process). A demand region R_d is defined as a connected subgraph that contains nodes in $G \setminus I$. A supply region R_s is a connected subgraph that contains nodes in $I \setminus G$. The nodes in R_d and R_s are connected with edges with infinite capacity to the closest neighbor in terms of hop distance to any node in $I \cap G$. Multiple supply and demand regions can exist in a single graph $\mathcal{G}$ depending on the symmetrical difference between the initial shape I and the goal shape G ($I \triangle G$). A super-supply S_s node is added to the graph and is connected to all nodes in all supply regions with an edge of capacity 1. All nodes in the demand regions are connected to a super-demand S_d node with an edge of capacity 1. The nodes in $G \cap I$ are connected with edges of infinite capacity. An example of this construction is shown in Fig. 6a.

Once the graph $\mathcal{G}$ is built, inspired by [19], we apply the Edmonds-Karp algorithm [10]. This algorithm is specifically designed for flow networks, a type of directed graph where edges have specified capacities and can accommodate flows that do not exceed those capacities. The application of the algorithm aims to determine the maximum flow between the super-supply node S_s and the super-demand node S_d. Its execution involves finding the shortest augmenting paths between S_s and S_d using breadth-first searches on the residual graph. It terminates when no more augmentations can be found.

A max-flow algorithm generates paths connecting all nodes in R_d to R_s, facilitating parallel motion. It also solves the assembly problem by finding the best order to dismantle sources in R_s regions and assemble them in demand regions. This preserves connectivity during dismantling and avoids collisions in demand regions by ensuring nodes don't assemble at the same empty position simultaneously.

The flow resulting after applying the Edmonds-karp algorithm on the graph of Fig. 6a at each edge is shown in Fig. 6b. The nodes S_s and S_d are removed because they are virtual nodes and do not represent any meta-module. The flow value f_{uv} on an edge u, v indicates the number of modules to be routed by meta-module u to meta-module v.

Each meta-module must know the flow value towards its neighbor meta-modules. These values must be sent from the central station to all meta-modules. This can be done at an initialization phase through tree-based broadcasts starting from a root module wired to the central station.

5.1.1 Flow Properties

Applying the Edmonds-Karp algorithm to the graph construction mentioned above produces a flow with the following properties:

Property 1 The flow covers all demand regions i.e. in the resultant flow, a path exists that connects a supply node to a demand node.

Edmonds-karp algorithm satisfies the flow conservation constraint at the terminal nodes, which states that the sum of the flow flowing out of the source S_s is equal to the sum of the flow flowing into the sink S_d. Since the number of supply nodes is equal to the number of demand nodes and the number of edges going out of S_s is equal to the number of edges going in S_d, each augmenting path, excluding S_s and S_d, starts with a supply node and ends at a demand node.

Property 2 The total length in terms of hop distance of the paths connecting the supply nodes to the demand nodes is minimized.

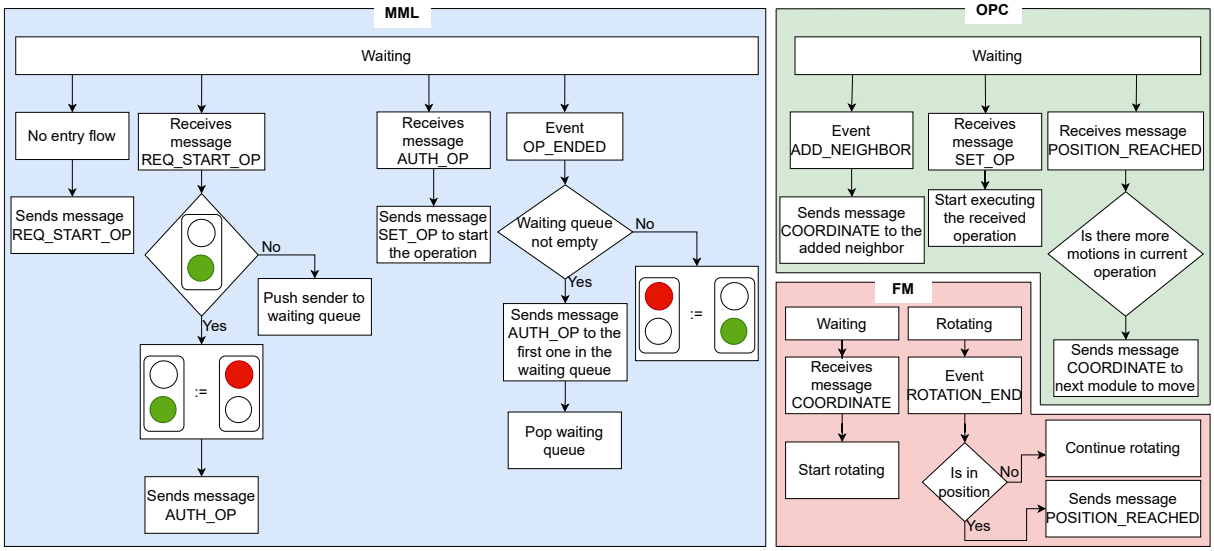


Fig. 7: Simplified view of the behavior of the MML, OPC and FM executing the distributed flow control algorithm.

The Edmonds-Karp algorithm finds the shortest possible augmenting path using a breadth-first search. The length of the paths found at each iteration increases monotonically. Therefore, the total length of the path set is minimized. This is an important property, as minimizing the distance traveled reduces the number of commands required for the modules. Therefore, the total energy consumed during self-reconfiguration is also reduced.

Property 3 No two paths connecting supply nodes to demand nodes share a common edge with opposite directions which may cause a head-on collision.

Having two paths in the resultant flow with two edges with opposite directions connecting the same two nodes contradicts property 2. This is because switching the destinations on those paths will reduce the total length.

5.2 Distributed Flow Control Algorithm

Once the flow values on connections between neighbor meta-modules are received, the modules can start to flow from the supply regions to the demand regions. To do so, a distributed algorithm is executed on each of the meta-modules using a message-passing method inspired by traffic light to handle the flow on concurrent paths. To do so, we identify three module's roles:

1. *Meta-Module Leader* (MML) is a module chosen in each meta-module whose purpose is to handle computation and communications between meta-modules. For instance, messages between meta-modules are sent from a MML to another. The MML can be any of the ten modules forming a meta-module.
2. *Operation Coordinator*(OPC) is a module that coordinates the operations executed by the meta-models by choosing the sequence of movements to execute by a flowing module. The OPC is the first module a moving module from a previous operation get connected to. Or, in the case of dismantle operations, it is the last module connected to the meta-module in the operation direction.
3. *Flowing Module* (FM) is a module in motion executing an operation's motion sequence.

Algorithms 1, 2, and 3 describe the behavior of each of the module's roles: MML, OPC and FM, respectively. Additionally, Fig. 7 presents a simplified depiction of the behavior associated with each role.

The flow starts by executing dismantle operations on meta-modules that are in R_s and do not

Algorithm 1 Distributed control algorithm on an MML (Part 1).

Data: F : A queue of pairs $< direction, flow >$ representing flow values in each directions.
Data: *lightState*
1: **Initialization**
2: **if** is MML of a meta-module u **then**
3: MUST_DISMANTLE(u)
4: **end if**
5: **end Initialization**

6: **Function** MUST_DISMANTLE(u)
7: **if** $u \in R_s$ and no flow is entering u **then**
8: $OPC.Operation \leftarrow dismantle(F_0.direction)$
9:
10: **send** REQ_START_OP() to MML in direction $F_0.direction$
11: **end if**
12: **end Function**

13: **Function** SET_OPERATION( )
14: **if** hasNeighborInDirection($F_0.direction$) **then**
15: **send** SET_OP(transfer, $F_0.direction$) to OPC
16: **else**
17: **send** SET_OP(build, $F_0.direction$) to OPC
18: **end if**
19: $F_0.flow \leftarrow F_0.flow - 1$
20: **if** $F_0.flow = 0$ **then**
21: $F.pop()$
22: **end if**
23: **end Function**

Algorithm 1 Distributed control algorithm on an MML (Part 2)

24: **Msg Handler** REQ_START_OP( )
25: **if** $lightState = GREEN$ **then**
26: $lightState \leftarrow RED$
27: SET_OPERATION()( )
28: **send** AUTH_OP() to $senderMML$
29: **else**
30: $waiting.push(direction_{sender})$
31: **end if**
32: **end Msg Handler**

33: **Msg Handler** AUTH_OP( )
34: Notify the OPC to start executing the operation
35: **end Msg Handler**

36: **Event** OP_ENDED ON META-MODULE u
37: **if** $waiting \neq \emptyset$ **then**
38: **send** AUTH_OP to $waiting_0$
39: $waiting.pop()$
40: **else**
41: $lightState \leftarrow GREEN$
42: MUST_DISMANTLE(u)
43: **end if**
44: **end Event**

have an entry flow (cf. the two top red nodes in Fig. 6b). However, before starting the execution of any operation, a meta-module must verify that the next meta-module is not executing any operation to prevent collisions on intersections. The system employs a traffic light mechanism, managing meta-modules' availability for operations through color-coded *lightState* variable (green for available, red for engaged). To initiate an operation, the MML sends a REQ_OP_START message to the MML of the next meta-module in its path (Algorithm 1 line 1-12). If the receiver's *lightState* is green, which means that it is not executing any operation, it sets the next operation to execute on it, then it responds with an AUTH_OP message, and its *lightState* becomes red. Once the AUTH_OP message is received, the OPC can start the operation. Otherwise, the receiver stores the direction of the sender in a queue and responds once it becomes free. This will cause flowing modules to wait for the next meta-modules they must enter to finish executing the operation in progress (Algorithm 1 line 25-44). This queuing along with not having two paths with opposite directions at an intersection as explained in Property 3 of Section 5.1.1 prevents meta-modules from being stuck in a state where they are indefinitely waiting for each other at intersections, effectively eliminating the risk of deadlock.

Each FM performing an operation keeps an iterator on the sequence of movements that it must execute. When an FM reaches an OPC, if the meta-module is authorized to perform the next operation, the OPC informs the module of which operation to execute and in which direction

Algorithm 2 Distributed control algorithm for an OPC

Data: *Operation*: The operation in execution.
Data: $mvt\_it = 0$: Iterator on Operation's movements.

1: **Event** ADD_NEIGHBOR(m)
2: **if** operation execution is authorized by MML **then**
3: **send** COORDINATE(*Operation*, *mvt_it*) to m
4: $mvt\_it \leftarrow mvt\_it$ + nb of moves to be performed by m
5: **else**
6: MML **sends** REQ_START_OP to next MML
7: **end if**
8: **end Event**

9: **Msg Handler** SET_OP(Op) State
$Operation \leftarrow Op$
10: **end Msg Handler**

11: **Msg Handler** POSITION_REACHED( )
12: **if** $mvt\_it < Operation.size()$ **then**
13: m = module at $Operation[mvt\_it].current\_position$
14: **send** COORDINATE(*Operation*, *mvt_it*) to m
15: $mvt\_it \leftarrow mvt\_it$ + nb of moves to be performed by m
16: **end if**
17: **end Msg Handler**

Algorithm 3 Distributed control algorithm for a FM

Data: *Operation*: The operation in execution.
Data: $mvt\_it = 0$: Iterator on Operation's movements.

1: **Msg Handler** COORDINATE(Op, it)
2: $Operation \leftarrow Op$
3: $mvt\_it \leftarrow it$
4: ROTATETO($Operation[mvt\_it].nextPosition$)
5: **end Msg Handler**

6: **Event** ROTATION_END
7: **if** $mvt\_it = Operation.size \wedge Operation.isAssemble$ **then**
8: meta-module reached goal position
9: **else if** $Operation.state = MOVING$ **then**
10: $mvt\_it \leftarrow mvt\_it + 1$
11: ROTATETO($Operation[mvt\_it].nextPosition$)
12: **else if** $Operation.isDismantle \wedge Operation.state = IN\_POSITION$ **then**
13: **send** POSITION_REACHED to OPC
14: **end if**
15: **end Event**

16: **Event** REMOVE_NEIGHBOR
17: **if** $Operation.state = WAITING$ **then** ▷ Bridge
18: **if** all modules have passed **then**
19: $mvt\_it \leftarrow mvt\_it + 1$
20: ROTATETO($Operation[mvt\_it].nextPosition$)
21: **end if**
22: **end if**
23: **end Event**

by sending a COORDINATE message containing the type of operation and the value of the iterator. Otherwise, the MML requests the authorization to start the operation from the next MML (Algorithm 2 line 1-6). On reception, a FM knows from which movement it must begin and starts to rotate until it reaches the IN_POSITION state, which means that the module has finished its sequence of movements for the current operation, or the WAITING state, which means that it reached a bridging position, so it must wait for the next modules to pass it (Algorithm 3).

When a MML detects that the operation's execution ended and there exists a meta-module waiting for its authorization to start the pending operation, it sends an AUTH_OP message to the waiting meta-module to start executing the operation. Otherwise, if it is in R_s and does not have an entry flow, it sets *lightState* as green and dismantles itself. Therefore, meta-modules in R_s execute the dismantle operation one after the other starting from the end of a path so that they do not disconnect the configuration.

The FMs must keep enough empty space between them to avoid blocking and collisions. To do so, a traffic-light like motion coordination algorithm [35] is used. It requires exchanging messages between the FM, its next rotation pivot, and

its next latching point. The function *rotateTo* in Algorithm 3 includes this algorithm. The reader can refer to [35] for a complete description of the motion coordination algorithm.

5.3 Self-reconfiguration from an initial to a goal configuration with different sizes

The meta-module design presented in [2] allows it to be filled in its internal volume with 10 additional modules, that is, the size of another meta-module, giving the structure the ability to expand or compress by a factor of 2. Therefore, the size of the goal configuration can differ from the size of the initial configuration. If N is the number of modules in the initial configuration and S_G the number of meta-modules filled or empty in the goal configuration, then: $\lceil \frac{N}{20} \rceil \leq S_G \leq \frac{N}{10}$.

If the number of meta-modules in the goal configuration is larger than the initial configuration ($\mathsf{G} > \mathsf{I}$), there must be at least $N_F = \mathsf{G} - \mathsf{I}$ full meta-modules in I. The filled meta-modules can be considered as supplies and added to the supply region. They can execute an operation that allows their filling modules to be transferred to the demand. So, the algorithm can proceed as previously explained.

If the number of meta-modules in the goal configuration is smaller than the initial configuration ($\mathsf{G} < \mathsf{I}$), there will be some excess of meta-modules that does not belong to G after the algorithm is complete. The number of modules that make up meta-modules in excess can be filled in the empty meta-modules in G. Therefore, the algorithm can be executed in two phases. The first executes the algorithm as previously explained. Once the completion of the algorithm is detected, which requires a termination detection mechanism, the second phase executes the same algorithm, but the global planner considers the supply region as all the meta-modules in excess and the demand region as all the empty meta-modules that can be filled.

6 Complexity Analysis

In this section, we analyze the complexity of *ASAPs* algorithm. The total time needed for self-reconfiguration includes the time T_0 taken by the global planner to find paths, the time for module transport T_1, and the time for message transmission T_2.

The computational complexity of the global planner is given by the complexity of constructing $\mathcal{G} = G \cup I$ plus the complexity of the max-flow algorithm. The construction of $\mathcal{G}$ linearly depends on the number of nodes V, so $O(V)$. The complexity of Edmonds-karp algorithm on any graph is given as $O(V.E^2)$ [10] where V is the number of nodes and E is the number of edges in $\mathcal{G}$. In our case, $\mathcal{G}$ is a graph representing nodes in a cubic lattice so, the maximum number of edges E is equal to $6.V$. Therefore, the computational complexity of the global planner is $O(V.(6.V)^2 + V) = O(V^3) = T_0$.

The time for transforming an initial shape to the target shape is mostly due to the time of modules flow. The longest distance that a module can travel is the diameter $d_{\mathcal{G}}$ of $\mathcal{G} = G \cup I$ where G is the goal configuration and I is the initial configuration. Modules can flow in parallel following concurrent paths of maximum length $d_{\mathcal{G}}$ and meta-modules can flow simultaneously following each other along a path. Therefore, the time complexity for modules flow on any path without waiting time on intersections can be expressed as $O(d_{\mathcal{G}})$. Regarding waiting times at intersections between multiple paths, for any meta-module situated at an intersection within a path, there can be a maximum of five incoming flowing paths in the 3D regular cubic lattice. In a scenario where five meta-modules are waiting for their operation execution at an intersection, and they belong to a path with a maximum length of $d_{\mathcal{G}}$, the complexity for waiting time is $O(5.d_{\mathcal{G}})$. Therefore, the overall complexity flow time is $O(d_{\mathcal{G}}) = T_1$.

It is interesting to compare this complexity with the complexity of the *RePoSt* algorithm, which was $O(d.N_{rounds})$, where N_{rounds} was the number of rounds necessary to achieve reconfiguration. In the worst case $N_M - 1$ rounds are required, where N_M is the number of meta-modules. *ASAPs* performs the self-reconfiguration in a single round with an increase in motion parallelization, which takes less time to achieve the goal configuration.

The message complexity of *ASAPs* is due to sending the flow values to their corresponding meta-modules and to the messages used during the flow control algorithm described in Section5.2.

Sending the flow values can be done via a breadth-first spanning tree rooted at the central station which take $O(d_{C_I}.N_M)$ messages where C_I is the initial configuration and d_{C_I} denotes the diameter of C_I. Each flowing module movement along the flowing path requires sending a fixed amount of messages. Therefore, the flow of modules requires $O(d_{\mathcal{G}}.N_m)$ where N_m is the number of modules. The message complexity of both steps can be expressed as $O(d_{C_I}.N_M + d_{\mathcal{G}}.N_m) = T_2$.

7 Simulation and Results

Simulations were carried out using the *VisibleSim* simulator [37]. The same code is executed by each robot, here we simulate with *3D Catoms* that are able to communicate with up to 12 neighbors and move by rotating on their connected neighbors. The code embed all the agents that can be activated depending on the neighborhood of the robot and the received messages.

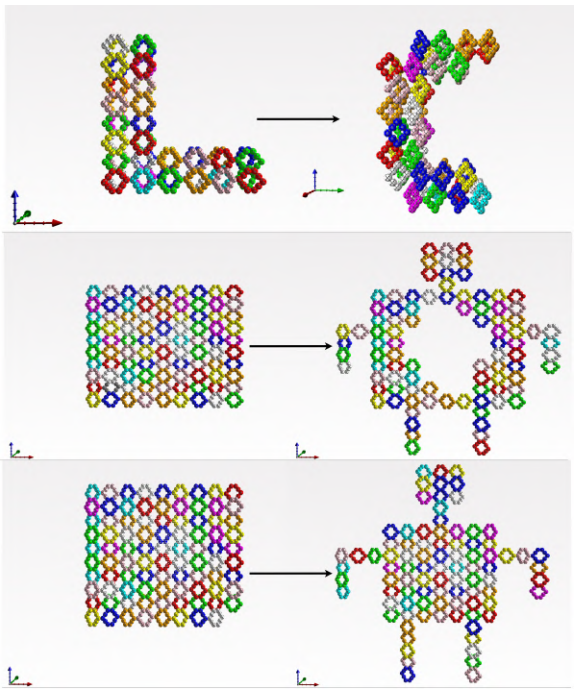


Fig. 8: Three different self-reconfiguration scenarios, from the top to the bottom: L2C, Hollow Human and Solid Human. Initial configurations are on the left and goal configuration on the right.

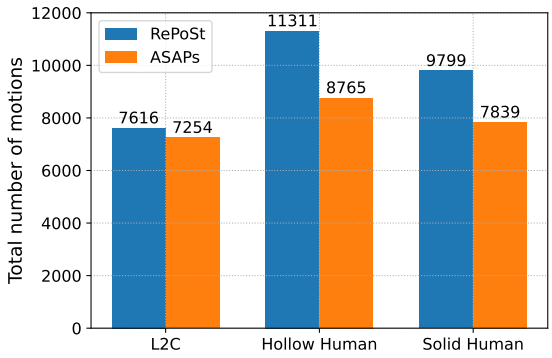
The basic dismantle, transfer and assembly actions applied in meta-modules during the implemented algorithm were presented in [2].

7.1 Presentation of the experiments

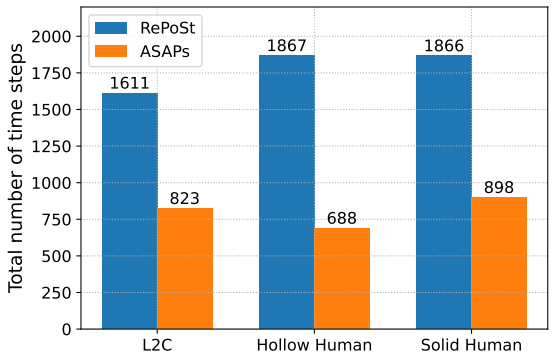
In order to evaluate the algorithm, we consider different same size initial and final configurations with several properties:

- **L2C**: From a two layers L shape made of 48 meta-modules to a two layers C shape. This model shown in Fig. 8.a, is similar to a narrow line of 2×2 meta-modules of the section. The narrowness of the line reduces the number of possible simultaneous motions.
- **Hollow Human**: From a single layered 2D square of 90 meta-modules to a single layered 2D hollow humanoid shape. This model shown in Fig. 8.c proposes fewer internal meta-modules in its goal configuration than the previous one. The meta-modules in the central area must be dismantled without causing a disconnection.
- **Solid Human**: From a single layered 2D square of 89 meta-modules to a single layered 2D 2D humanoid shape. This model shown in Fig. 8.b is rich of numerous different paths in the central area (the body of the humanoid shape), but there are only some paths to reach the head, arms, and legs area which will cause bottlenecks. The goal configuration is formed by transporting the meta-modules on the initial configuration borders to the head, arms and legs.

These three self-reconfigurations are presented in a video[2]. This video shows the dismantle, displacement and building of meta-modules in parallel to transform the initial shape to obtain the final configuration of meta-modules. In the final part of the video, we demonstrate the expandable nature of the structure by transitioning from a three-layer to a four-layer cuboid configuration. This is achieved by emptying the meta-modules of the bottom layer to build the new top layer.



(a)



(b)

Fig. 9: Comparisons of the total number of motions (a) and and total number of time steps (b) of *ASAPs* and *RePoSt* algorithms for the 3 experimental shapes.

7.2 Experiments analysis

Fig. 9 compares the speed and the total number of motions of the self-reconfiguration process for the *ASAPs* algorithm and *RePoSt* presented in [2].

First, we notice that for the three self-reconfigurations, the *ASAPs* algorithm is faster than *RePoSt*, and this is mainly because the parallelization of movements is much more important due to the coordinated flow of modules on the pre-allocated concurrent paths. *ASAPs* algorithm is 1.95 times faster for L2C, 2.7 times faster for the Hollow Human configuration and 2.08 times faster for the Solid Human configuration.

[2] https://youtu.be/MUfuY0ao-0w

Second, regarding the total number of motions executed by the modules during the self-reconfiguration, *ASAPs* requires less number of motions than *RePoSt* to converge to the goal shape. This is due to the global max-flow planning method that minimizes the total length of the found paths connecting meta-modules in the supply region to the meta-modules in the demand region (cf. property 2). Therefore, *ASAPs* is more energy efficient than *RePoSt*.

The variations in performance between *ASAPs* and *RePoSt* across configurations are notably influenced by *RePoSt*'s round-based approach and the resultant limitations on the number of non-intersecting paths generated in each round which depends on the geometry of the configurations.

Fig. 10 shows the number of module motions and the number of modules that are waiting per time step when executing *RePoSt* and *ASAPs*. A time step corresponds to the average time required for a *3D Catom* rotation.

The regular oscillations of the curve shown in Fig. 10 (a), (b) and (c) for the *RePoSt* algorithm are evidence of the successive rounds of this algorithm, they regularly cause periods with a low number of movements due to the time required for the determination of streamlines at each round. This effect disappears almost completely on the curve given by self-reconfiguration with *ASAPs* algorithm. This is because once the meta-modules receive the path information, they all start to flow asynchronously guided by the distributed control algorithm explained in Section 5.2. Therefore, the *ASAPs* number of motions curve starts by increasing until it reaches its maximum value, then stabilizes before it starts to decrease when the modules start to reach their goal positions. This shows the increase in motion parallelisms achieved using *ASAPs*.

In Fig. 10 (d), (b) and (f), the number of waiting modules at a time step varies with the amount of flow. When more modules leave their initial position and start flowing, the number of waiting modules will increase because modules will wait for each other to keep enough space when flowing in a train-like fashion towards their destination. In *ASAPs* an increased number of modules can be found in a waiting state at the beginning time steps for two reasons: first, the number of flowing modules is more important, so more modules

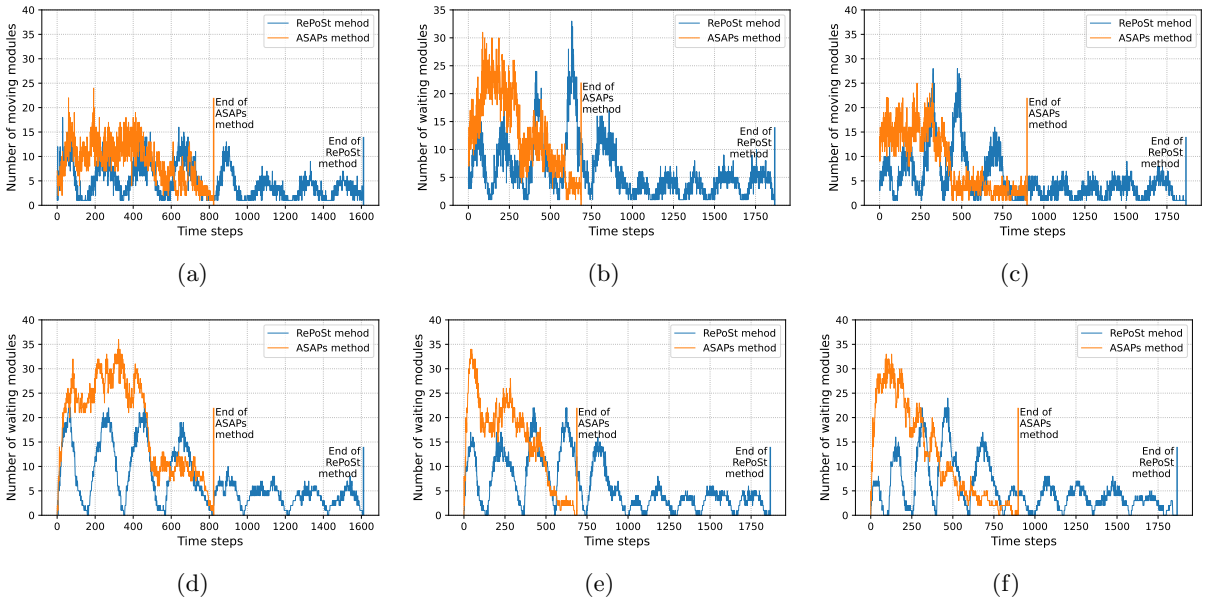


Fig. 10: Comparisons of *ASAPs* and *RePoSt* number of modules in motions per time step (a, b and c) and number of waiting modules (d, e, and f) for the three experimental shapes: (a) and (d) for the L shape to C shape (L2C), (b) and (e) for the Square to Hollow Human shape and (c) and (f) for the Square to Solid Human.

are waiting to keep enough space with other flowing modules on the same path, and second, when modules must wait at intersections of paths in case an operation is being executed at the intersection. For the *RePoSt* algorithm, modules follow disjoint paths, so they are not required to wait on intersection.

The waiting time of the modules that execute *ASAPs* also depends on the bottlenecks at intersections of paths that reach a narrow area. For example, Fig. 11, shows the paths generated by the max-flow in three configurations where the modules in R_s must cross none, two, or three bottleneck nodes to fill empty positions in R_d. Fig. 12 shows the total self-reconfiguration time of the three examples. It can be seen that when multiple paths intersect on one node causing a bottleneck, the self-reconfiguration time increases. The reason is that only one operation is executed at a time on the bottleneck nodes. Modules that need to go through bottlenecks are waiting for their turn, as explained in Sections 5.2.

8 Conclusion and Future Work

In this paper we presented *ASAPs* a hybrid self-reconfiguration algorithm for programmable matter based on modular robots. We used a porous structure made of hexagonal shape meta-modules made of *3D Catoms*. The algorithm consists of a centralized global path planner based on a max-flow search and a distributed message-passing asynchronous flow control algorithm. We evaluated *ASAPs* in simulation and compared the parallelism, total distance traveled and self-reconfiguration time with *RePoSt* a distributed synchronous self-reconfiguration algorithm for similar structures. The results show an important improvement of efficiency in both the total distance traveled which affect the energy used by the modules and in the self-reconfiguration time.

Future works will focus on the usage of intermediate configurations in the aim of reducing the effect of bottlenecks that might exist in a configuration on the self-reconfiguration time. For example, in the case where a bottleneck is detected

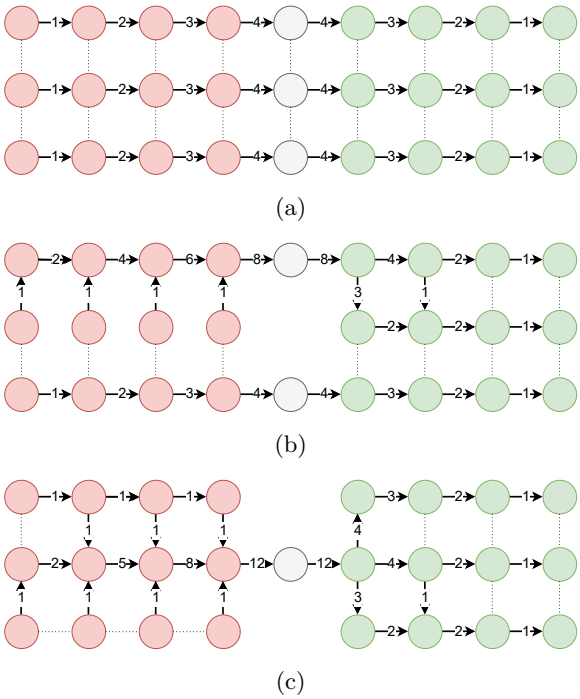


Fig. 11: The result of paths generated by the max-flow on 3 configurations with different bottleneck sizes. R_s nodes are in red and R_d nodes are in green. (a) No bottlenecks. (b) Two bottlenecks. (c) One Bottleneck.

in a given configuration, we can build a temporary meta-module at an empty position next to the bottleneck to double the flow towards demand regions and decrease the waiting time at intersections. Additionally, in order for our algorithms to be applicable on real-hardware we intend to incorporate physical constraints, such as ensuring structural stability [26] and fault-tolerance [4, 21] into the self-reconfiguration planning process.

Declarations

Funding. This work has been supported by the EIPHI Graduate School (contract "ANR-17-EURE-0002").

Conflict of interest. The authors have no relevant financial or non-financial interest to disclose.

Authors' contributions. J.Bassil conceptualized and implemented the proposed algorithm and wrote the main manucript text. B.Piranda provided supervision, contributed to writing and editing the manuscript, prepared Figure 1, and edited the YouTube video. A.Makhoul provided supervision, validated, reviewed and edited the manuscript. J.Bourgeois provided supervision, reviewed the manuscript and acquired funding.

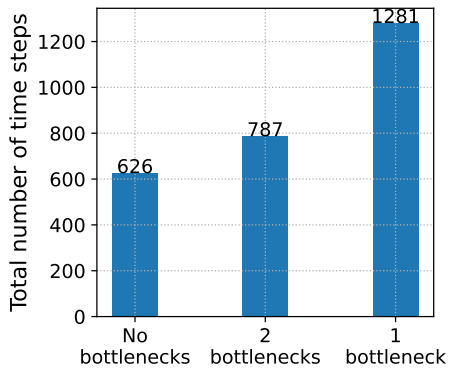


Fig. 12: Total number of time steps on the self-reconfiguration example of Fig. 11.

References

[1] Ahmadzadeh H, Masehian E (2015) Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization. Artificial Intelligence 223:27–64. https://doi.org/10.1016/j.artint.2015.02.004, URL https://linkinghub.elsevier.com/retrieve/pii/S0004370215000260

[2] Bassil J, Piranda B, Makhoul A, Bourgeois J (2022) A new porous structure for modular robots. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, AAMAS '22, p 1539–1541

[3] Bassil J, Piranda B, Makhoul A, Bourgeois J (2022) Repost: Distributed self-reconfiguration algorithm for modular robots based on porous structure. In: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 12,651–12,658, https://doi.org/10.1109/IROS47612.2022.9981212

[4] Bassil J, Tannoury P, Piranda B, Makhoul A, Bourgeois J (2022) Fault-tolerance mechanism for self-reconfiguration of modular robots. In: 2022 International Wireless Communications and Mobile Computing (IWCMC), IEEE, pp 360–365

[5] Bassil J, Yaacoub JPA, Piranda B, Makhoul A, Bourgeois J (2023) Distributed shape recognition algorithm for lattice-based modular robots. In: 2023 International Symposium on Multi-Robot and Multi-Agent Systems (MRS), pp 85–91, https://doi.org/10.1109/MRS60187.2023.10416786

[6] Bourgeois J, Piranda B, Naz A, Boillot N, Mabed H, Dhoutaut D, Tucci T, Lakhlef H (2016) Programmable matter as a cyber-physical conjugation. In: 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), IEEE, pp 2942–2947

[7] Castano A, Shen WM, Will P (2000) Conro: Towards deployable robots with inter-robots metamorphic capabilities. Autonomous Robots 8(3):309–324

[8] Dewey DJ, Ashley-Rollman MP, De Rosa M, Goldstein SC, Mowry TC, Srinivasa SS, Pillai P, Campbell J (2008) Generalizing metamodules to simplify planning in modular robotic systems. 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS pp 1338–1345. https://doi.org/10.1109/IROS.2008.4651094

[9] Dokuyucu Hİ, Özmen NG (2022) Achievements and future directions in self-reconfigurable modular robotic systems. Journal of Field Robotics https://doi.org/10.1002/rob.22139, URL https://onlinelibrary.wiley.com/doi/10.1002/rob.22139

[10] Edmonds J, Karp RM (1972) Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (JACM) 19(2):248–264

[11] Fitch R, Butler Z, Rus D (2003) Reconfiguration planning for heterogeneous self-reconfiguring robots. In: Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453), IEEE, pp 2460–2467

[12] Gerbl M, Gerstmayr J (2022) Self-reconfiguration of shape-shifting modular robots with triangular structure. Robotics and Autonomous Systems 147:103,930. https://doi.org/10.1016/j.robot.2021.103930

[13] Gilpin K, Knaian A, Rus D (2010) Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In: 2010 IEEE International Conference on Robotics and Automation, IEEE, pp 2485–2492

[14] Hourany E, Stephan C, Makhoul A, Piranda B, Habib B, Bourgeois J (2021) Self-reconfiguration of modular robots using virtual forces. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, pp 6948–6953

[15] Kawano H (2019) Distributed tunneling reconfiguration of sliding cubic modular robots in severe space requirements. In: Correll N, Schwager M, Otte M (eds) Distributed Autonomous Robotic Systems. Springer International Publishing, Cham, pp 1–15

[16] Kawano H (2020) Distributed tunneling reconfiguration of cubic modular robots without meta-module's disassembling in severe space requirement. Robotics and Autonomous Systems 124:103,369. https://doi.org/https://doi.org/10.1016/j.robot.2019.103369, URL https://www.sciencedirect.com/science/article/pii/S0921889019301447

[17] Kotay KD, Rus DL (2000) Algorithms for self-reconfiguring molecule motion planning. In: Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113), pp 2184–2193 vol.3, https://doi.org/10.1109/IROS.2000.895294

[18] Kurokawa H, Tomita K, Kamimura A, Kokaji S, Hasuo T, Murata S (2008) Distributed

self-reconfiguration of m-tran iii modular robotic system. The International Journal of Robotics Research 27(3-4):373–386

[19] Lengiewicz J, Hołobut P (2018) Efficient collective shape shifting and locomotion of massively-modular robotic structures. Autonomous Robots https://doi.org/10.1007/s10514-018-9709-6, URL https://doi.org/10.1007/s10514-018-9709-6

[20] Liu C, Lin Q, Kim H, Yim M (2022) Smores-ep, a modular robot with parallel self-assembly. Autonomous Robots pp 1–18

[21] Makhoul A, Bassil J (2023) Fault tolerance technique using bidirectional heteroassociative memory for self-reconfigurable programmable matter. In: 2023 International Wireless Communications and Mobile Computing (IWCMC), IEEE, pp 1619–1625

[22] Murata S, Yoshida E, Kamimura A, Kurokawa H, Tomita K, Kokaji S (2002) M-tran: Self-reconfigurable modular robotic system. IEEE/ASME transactions on mechatronics 7(4):431–441

[23] Østergaard EH, Kassow K, Beck R, Lund HH (2006) Design of the atron lattice-based self-reconfigurable robot. Autonomous Robots 21(2):165–183

[24] Piranda B, Bourgeois J (2018) Designing a quasi-spherical module for a huge modular robot to create programmable matter. Autonomous Robots 42(8):1619–1633. https://doi.org/10.1007/s10514-018-9710-0, URL http://link.springer.com/10.1007/s10514-018-9710-0

[25] Piranda B, Bourgeois J (2022) Datom: A deformable modular robot for building self-reconfigurable programmable matter. In: Matsuno F, Azuma Si, Yamamoto M (eds) Distributed Autonomous Robotic Systems. Springer International Publishing, Cham, pp 70–81

[26] Piranda B, Chodkiewicz P, Hołobut P, A. Bordas SP, Bourgeois J, Lengiewicz J (2021) Distributed prediction of unsafe reconfiguration scenarios of modular robotic programmable matter. IEEE Transactions on Robotics 37(6):2226–2233. https://doi.org/10.1109/TRO.2021.3074085

[27] Piranda B, Lassabe F, Bourgeois J (2022) DisCo: A multiagent 3d coordinate system for lattice based modular self-reconfigurable robots. In: IEEE International Conference on Robotics and Automation (ICRA 2023), IEEE, London, England, May. 28 June 02, 2023

[28] Pruszko L, Coutrix C, Laurillau Y, Piranda B, Bourgeois J (2021) Molecular hci: Structuring the cross-disciplinary space of modular shape-changing user interfaces. Proceedings of the ACM on Human-Computer Interaction 5:1–33. https://doi.org/10.1145/3461733, URL https://hal.archives-ouvertes.fr/hal-03215058https://hal.archives-ouvertes.fr/hal-03215058/document

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

[30] Sproewitz A, Laprade P, Bonardi S, Mayer M, Moeckel R, Mudry PA, Ijspeert AJ (2010) Roombots—Towards decentralized reconfiguration with self-reconfiguring modular robotic metamodules. In: Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pp 1126–1132, https://doi.org/10.1109/IROS.2010.5649504

[31] Stoy K (2006) Using cellular automata and gradients to control self-reconfiguration. Robotics and Autonomous Systems 54:135–141. https://doi.org/10.1016/j.robot.2005.09.017, URL https://linkinghub.elsevier.com/retrieve/pii/S0921889005001521

[32] Stoy K, Nagpal R (2007) Self-reconfiguration using directed growth. In: Alami R, Chatila R, Asama H (eds) Distributed Autonomous Robotic Systems 6. Springer Japan, Tokyo, pp 3–12

[33] Suh JW, Homans SB, Yim M (2002) Telecubes: Mechanical design of a module for self-reconfigurable robotics. In: Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292), IEEE, pp 4095–4101

[34] Thalamy P, Piranda B, Bourgeois J (2019) A survey of autonomous self-reconfiguration methods for robot-based programmable matter. Robotics and Autonomous Systems 120:103,242. https://doi.org/10.1016/j.robot.2019.07.012, URL https://doi.org/10.1016/j.robot.2019.07.012

[35] Thalamy P, Piranda B, Lassabe F, Bourgeois J (2020) Deterministic scaffold assembly by self-reconfiguring micro-robotic swarms. Swarm and Evolutionary Computation 58:100,722. https://doi.org/10.1016/j.swevo.2020.100722

[36] Thalamy P, Piranda B, Bourgeois J (2021) Engineering efficient and massively parallel 3d self-reconfiguration using sandboxing, scaffolding and coating. Robotics and Autonomous Systems 146:103,875. https://doi.org/https://doi.org/10.1016/j.robot.2021.103875, URL https://www.sciencedirect.com/science/article/pii/S0921889021001603

[37] Thalamy P, Piranda B, Naz A, Bourgeois J (2021) Visiblesim: A behavioral simulation framework for lattice modular robots. Robotics and Autonomous Systems p 103913. https://doi.org/https://doi.org/10.1016/j.robot.2021.103913, URL https://www.sciencedirect.com/science/article/pii/S0921889021001986

[38] Tucci T, Piranda B, Bourgeois J (2017) Efficient scene encoding for programmable matter self-reconfiguration algorithms. Proceedings of the ACM Symposium on Applied Computing Part F1280:256–261. https://doi.org/10.1145/3019612.3019706

[39] Ünsal C, Kiliççöte H, Khosla PK (2001) A modular self-reconfigurable bipartite robotic system: Implementation and motion planning. Autonomous Robots 10(1):23–40

[40] Vassilvitskii S, Yim M, Suh J (2002) A complete, local and parallel reconfiguration algorithm for cube style modular robots. In: Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on, pp 117–122 vol.1, https://doi.org/10.1109/ROBOT.2002.1013348

[41] White PJ, Yim M (2010) Reliable external actuation for full reachability in robotic modular self-reconfiguration. The International Journal of Robotics Research 29(5):598–612

[42] Yim M, Duff DG, Roufas KD (2000) Polybot: a modular reconfigurable robot. In: Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), IEEE, pp 514–520

[43] Zhang T, Zhang D, Gupta MM, Zhang W (2015) Design of a general resilient robotic system based on axiomatic design theory. In: 2015 IEEE International Conference on Advanced Intelligent Mechatronics (AIM), IEEE, pp 71–78