Deterministic Scaffold Assembly By Self-Reconfiguring Micro-Robotic Swarms

Pierre Thalamy[a,*], Benoît Piranda[a], Frédéric Lassabe[a], Julien Bourgeois[a]

[a] *Univ. Bourgogne Franche-Comté, FEMTO-ST Institute, CNRS*
1 cours Leprince-Ringuet, 25200, Montbéliard, France.

Abstract

The self-reconfiguration of large swarms of modular robotic units from one object into another is an intricate problem whose critical parameter that must be optimized is the time required to perform a transformation. Various optimizations methods have been proposed to accelerate transformations, as well as techniques to engineer the shape itself, such as scaffolding which creates an internal object structure filled with holes for easing the motion of modules. In this paper, we propose a novel deterministic and distributed method for rapidly constructing the scaffold of an object from an organized reserve of modules placed underneath the reconfiguration scene. This innovative scaffold design is parameterizable and has a face-centered-cubic lattice structure made from our rotating-only micro-modules. Our method operates at two levels of planning, scheduling the construction of components of the scaffold to avoid deadlocks at one level, and handling the navigation of modules and their coordination to avoid collisions in the other. We provide an analysis of the method and perform simulations on shapes with an increasing level of intricacy to show that our method has a reconfiguration time complexity of $O(\sqrt[3]{N})$ time steps for a subclass of convex shapes, with N the number of modules in the shape. We then proceed to explain how our solution can be further extended to any shape.

Keywords: Modular Robotic Swarm, Self-Reconfiguration, Large-Scale Swarm Coordination, Distributed Algorithm, Scaffolding

1. Introduction

Optimization methods is a field of research in constant innovation: Making its debut with methods using a centralized approach based on iterative searches, it quickly moved towards new paradigms like genetic algorithms (GA) (Goldberg and Holland, 1988). Then, it benefited from the invention of the concept of swarm intelligence (SI) (Beni, 2005), initially linked to modular robotic systems, called cellular robotic systems at the time. This revolution gave birth to ant colony optimization (ACO) (Dorigo et al., 1996) and particle swarm optimization (PSO) (Kennedy and Eberhart, 1995), which are methods that rely on the emergence of the solution through the interaction of the elements of the system and that can be applied to many different kinds of applications.

In the Programmable Matter Project[1], we are building a modular robot composed of quasi-spherical micro-robots able to compute, communicate and move around each other (see Figure 1). We also develop algorithms to perform various tasks with these robots. Among these tasks, the most difficult to perform is almost certainly *self-reconfiguration*, which transforms an initial configuration of connected micro-robots into a final one. The number of unique configurations that can be created with n modules is huge: $(c \times w)^n$, where c is the number of possible connections per module, and w the number of ways to connect the modules together (Park et al., 2008). Furthermore, the most critical parameter of self-reconfiguration is the time required to perform a transformation, the reconfiguration time. To optimize the reconfiguration time, modules must move concurrently, which unfortunately 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 (Barraquand and Latombe, 1991). The exploration space for reconfiguration between two random configurations is therefore exponential in the number of modules, which prevents complete optimal planning for all but the simplest configurations.

To solve this problem more efficiently we propose therefore two optimizations. The first one is to change the way we define an object: rather than constructing an object filled with micro-robots, we define it using its boundary representation. Second, we propose to build an object using an internal *scaffold* that leaves internal holes inside the shape to facilitate motion and coordination. This scaffold can then be coated by modules afterward so as to preserve the external aspect of the object. Accordingly, while the object will look like a plain object from the outside, it will actually be composed exclusively of a scaffold with an added coating. The resulting object will thus contain fewer micro-robots than it would otherwise, and these micro-robots will be able to move inside the object; these two features significantly contribute to decreasing the reconfiguration time.

As an example, an estimate of the reconfiguration time for our sliding blocks (Piranda et al., 2013) and using the metrics introduced in (Zhu and El Baz, 2019) is 12 h for 800 blocks. This is just a rough estimate to stress that time is really an issue

*Corresponding author.
Email addresses: pierre.thalamy@femto-st.fr (Pierre Thalamy), benoit.piranda@femto-st.fr (Benoît Piranda), frederic.lassabe@utbm.fr (Frédéric Lassabe), julien.bourgeois@femto-st.fr (Julien Bourgeois)

[1] http://projects.femto-st.fr/programmable-matter/

in self-reconfiguration algorithms. Besides, it was showed in practice that it could take 11.66 h to reconfigure an ensemble of 1,000 Kilobots (Rubenstein et al., 2012). Given a rotation time of 20 ms, as gauged by our latest hardware experiments with 2 mm *3D Catoms*, and using scaffolding, we estimate that it would take roughly 6 s to build the scaffold of a cube of size $19 \times 19 \times 19$ modules ($110\,cm^3$) and made of nearly 1,200 modules—while the filled cube of the same size would consist of 6,859 modules. Even though 6 s is an order of magnitude and not a precise result, it shows that together, electrostatic actuation and the scaffolding algorithm we propose can dramatically reduce the reconfiguration time, enabling practical applications for programmable matter.

The objective of this article is to introduce a method for building an internal robotic scaffold of a large class of objects in sublinear time, through the coordinated effort of up to millions of distributed rotating modules in a 3D grid.

Section 2 starts by introducing the related work, followed by an overview of the fundamentals of our method in Section 3. Section 4 then presents the first version of our algorithm, specialized in building various dimensions of square pyramids, as a way to familiarize the reader with the concept and mechanisms at play. We then dive, in Section 5, into the partial generalization of the scaffolding algorithm to a well-defined sub-class of convex shapes, and demonstrate how it performs theoretically, and in simulation on various reconfiguration cases. In Section 6, we describe the conditions under which our work can be extended to any shape, and the solutions that we are investigating for that purpose, before concluding and proposing some future work in Sections 8 and 9. All the presented algorithms are fully decentralized and operate on robotic ensembles where micro-robots act like autonomous agents. Lastly, the simulations presented in this paper were performed in *VisibleSim* (Piranda, 2016), a discrete event simulator for large modular robotic systems.

2. Related Work

One way to generally approach the kind of problem tackled in this work is by use of metaheuristics from the field of biologically inspired computing (Ser et al., 2019). Those are stochastic techniques that are generally classified into two groups: nature-inspired metaheuristics on the one hand, such as evolutionary algorithms (EA) (Bäck et al., 1997); and swarm intelligence (SI) on the other (Beni, 2005; Strumberger et al., 2019). GA (Goldberg and Holland, 1988) are the most widely used kind of EA algorithms, and can be used to evolve quality solutions to optimization and search problems (Goldberg and Holland, 1988; Chung and Shin, 2019) using genetics-inspired operations. Besides, SI uses decentralized control to let a system composed of multiple agents self-organize exclusively through interactions with themselves and their local environment. This leads to the emergence of complex behavior from the whole system, much like the complex behavior of ant colonies, flocks of birds, or honey bee colonies, in biological systems, which can be applied to a variety of computing applications (Rajasekhar et al., 2017). While both EA and SI have been successfully used in tackling NP-hard problems individually, hybrid methods that combine both SI and EA (GA, usually) also exist such as in PSO (Tuba et al., 2015).

Early work on self-reconfiguration considered only a very limited number of modules in the system (dozens), and with intricate geometries that made self-reconfiguration excessively complex (bipartite systems, for instance (Kotay and Rus, 2000; Ünsal et al., 2000), or others (Yoshida et al., 1998)). The field also moved away from heavily centralized approaches to more adequate distributed methods, better fitted for large-scale reconfiguration and the dynamic nature of the underlying systems. While most self-reconfiguration methods are deterministic, a number of stochastic methods can be found in the literature. Early attempts using stochastic relaxation (Yoshida et al., 1998) or simulated annealing (Kurokawa et al., 1998) suffered from a difficulty to converge into the goal shape as they were getting trapped into local minima. Nonetheless, more recent attempts such as (Fitch and McAllister, 2013)—based on a Markov decision process to optimize the number of connection/disconnection of modules—have proven very promising as they can be easily made generic and applied on diverse hardware systems and models. Stochastic methods might also be by themselves more robust to faults in hardware systems during reconfiguration, which is critical, while deterministic methods would need additional correction mechanisms.

On the other hand, deterministic approaches generally have a guaranteed convergence into the goal shape, but might need additional correction mechanisms in case of hardware failures as opposed to the built-in robustness of stochastic methods. The most popular self-reconfiguration model is by far the simple *sliding-cube*, which resides in a cubic lattice and can perform translations and convex rotations on the surface of other modules, or only one of the former in some models. Approaches vary from disassembly/reassembly through an intermediate shape (Fitch et al., 2003), tunneling through the shape with sliding-only cubes (Kawano, 2015) (both with quadratic operating time cost), to more specialized methods such as (Bie et al., 2018) which can build branching structures in a linear number of module motions using Lindenmayer-systems and cellular automata (Bie et al., 2018; Zhu et al., 2017).

Most related works did not assume a model with a complex geometrical grid and rotation only motions, unlike ours. However, Yim et al. (2001) showed with hundreds of modules that a flat 2D disk of Rhombic Dodecahedral (RD) modules in a Face-Centered-Cubic (FCC) Lattice could be reconfigured into various shapes in $O(n)$ reconfiguration steps (individual module motions) through a planning method named *goal ordering*, with n the number of modules, albeit with no guarantee of convergence. They nonetheless found that deadlock avoidance could still be improved slightly by adding randomness in the decision process of the modules. This is most relevant as these RD modules have motions constraints very similar to the ones of our own model, introduced in the next section. While it is usually assumed that all the modules composing a modular robot must remain connected at all times during reconfiguration, dropping this constraint can lead to interesting systems such as the rotating *M-Blocks* (Sung et al., 2015).

In order to aid self-reconfiguration, researchers introduced the notion of *scaffolding* in (Kotay and Rus, 2000), where the ensemble could be made porous thanks to the construction of multi-module structures so that modules could easily flow internally, simplifying planning, reducing the number of modules to displace, and therefore accelerating reconfiguration altogether. Scaffolding was further studied in (Støy, 2006; Støy and Nagpal, 2007) with a very simple scaffold geometry thanks to the simplicity of their cubic sliding and rotating robotic model. They proposed an elegant deterministic reconfiguration method based on cellular automata and simple gradients. The same scaffold geometry later inspired (Lengiewicz and Holobut, 2019), with a resembling model but solving reconfiguration through a max-flow search to optimize the flow of modules between the boundaries of the initial shape and those of the goal shape. Both achieved self-reconfiguration with a number of individual movements linear in the number of modules present in the system.

Furthermore, our approach is somewhat conceptually similar to (Dewey et al., 2008), where modules are arranged into regular multi-module units called *metamodules*, that can be in an empty (only structural modules of the unit), or a filled state (surplus of modules in the unit). Modules flow through the growing shape from filled metamodules to empty metamodules guided by a planner and achieve a completion time linear with the diameter of the ensemble. They did not address however the resource allocation aspect of the reconfiguration, that is to say how to decide on which part of the initial shape will fill each part of the goal shape, an inescapable and complex problem.

However, these scaffolding approaches considered an initial shape as a prebuilt scaffold but none addressed how to construct the scaffolding structure from a mass of modules, which is the topic of this paper. This work is, therefore, putting forward an original solution to a previously unstudied problem, as shape assembly work from the modular robotic literature is usually more concerned with the final latching of modules at specific locations than the planning of the motion that led them there (Tucci et al., 2018), and classical self-reconfiguration approaches with massive ensembles generally transform a shape into another rather than build one from the ground up (Dewey et al., 2008; Butler et al., 2002; Lengiewicz and Holobut, 2019). Hence, identifying bases of comparison for evaluating this work in regard to other assembly or self-reconfiguration solutions is arduous and the resulting findings might be inconclusive. As a matter of fact, this a much deeper problem in this line of work, as traditional (i.e., shape to shape) self-reconfiguration works are already afflicted by this evaluation conundrum, due to the variance in robotic models, capabilities, and modes of motion (Thalamy et al., 2019b; Ahmadzadeh et al., 2016).

3. Fundamentals

3.1. Modular Robotic Model

Our work considers the self-reconfiguration of modular robots made of quasi-spherical rotating modules named *3D Catoms*. These robots can attach to each other and rotate around one another using electrostatic actuation.

3.1.1. 3D Catom

3D Catoms have a quasi-spherical geometry which consists of 12 flat squares (named connectors, drawn in red in Figure 1a) linked by curves. Connectors are centered and tangent to the contact points of a dense set of spheres placed in a *Face-Centered-Cubic (FCC)* grid (cf. (Piranda and Bourgeois, 2018) for relative contact points coordinates). Two kinds of curves are placed between connectors to allow the rotation of a *3D Catom* around another: The first shape, the *hexagonal actuator* (drawn in green in Figure 1a) is made of a triangle and 3 sections of the body of a cylinder; the second shape, the *octagonal actuator* (drawn in blue in Figure 1a) is made of a square and 4 sections of the body of a cylinder.

The 12 electrostatic connectors on the surface of the *3D Catoms* are used for latching, actuation between modules, and peer-to-peer communication between connected neighbors. *3D Catoms* latch onto each other using their electrostatic connectors and assemble to form *FCC* lattice structures, resulting in staggered vertical layers of modules.

Individual movements consist in a rotation from one connector of a neighbor module to another connector on the surface of this same module, which acts as a *pivot*. Figure 1b shows the two possible ways of performing rotations, through an octagonal actuator at the bottom (rotating around the green pivot), and using a hexagonal actuator at the top (rotating around the yellow pivot). Each rotation displaces the rotating module from one cell to an adjacent one of the FCC lattice. More complex motions comprised of several steps are therefore sequences of individual rotations on the surface of neighbor modules.

Although this can seem like a strictly abstract model, the Programmable Matter Project is actively engaged in creating hardware *3D Catoms*. Current efforts are focused on the production of a prototype *3D Catom* that is 3.6 mm in diameter. Figure 1b shows the printed envelop of this micro-robot.

3.1.2. Module Assumptions and Critical Constraints

While the movement of a single module can seem trivial, the intricacy of self-reconfiguration becomes apparent when considering *3D Catoms* in a swarm context, with multiple modules attempting to perform their respective tasks in parallel. We model a modular robot consisting of a connected ensemble of *3D Catoms* as a distributed system, where:

- all modules are identical and they all execute the exact same distributed program;
- the graph constituted by all modules in the systems and their interconnection must remain connected at all times—this is the *connectivity constraint*;
- modules can only react to either the reception of a message, to the connection/disconnection of a neighbor, or to an internal event such as a timer event;
- computation is only performed locally to each *3D Catom*;
- communication is also performed exclusively in a local fashion, with modules only communicating with their

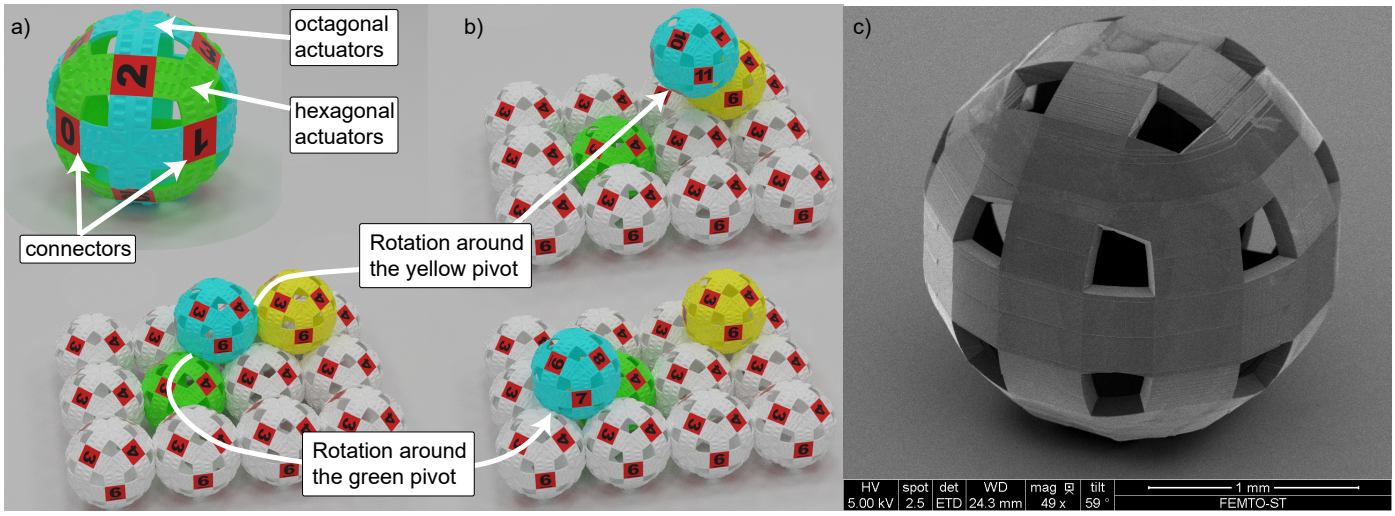

Figure 1: **The *3D Catom*:** sample motions and first prototype (Nanoscribe picture, thanks to Gwenn Ulliac)

immediate neighbors on the FCC grid, and through a message-passing scheme;

- all modules share a common coordinate system and have a global knowledge of the goal shape;
- modules perform asynchronously.

Furthermore, *3D Catoms* do not undergo any deformation when rotating, as deformation is likely to require moving parts, and *3D Catoms* are meant to be inexpensive and mass-producible by design. This raises, however, an important constraint on the movement of modules, as the geometry of *3D Catoms* thus does not allow a module to enter or leave a position that is surrounded by two opposing modules. This means for instance that in the case of two lines of modules growing into each other, it would not be possible to insert the last module required to bridge the gap between the two lines. This is a major constraint on any self-reconfiguration, as this means that the construction of any shape must follow a strict set of ordering principles and construction rules so as to avoid the occurrence of deadlocks during construction. From here on, we will refer to this constraint as the *bridging constraint*.

3.2. Scaffold Anatomy

The objective of our work is to build an internal scaffolding of a goal object as fast and efficiently as possible. This scaffold, which forms a sort of highly regular skeleton of an object, is composed of an arrangement of regular units sharing a common structure named *scaffold tiles*.

3.2.1. Structure of a Scaffold Tile

The scaffold tile is the parameterizable unit of the scaffold. All the tiles composing a *3D Catom* scaffold share a common geometry, but their exact structure can vary depending on the specific location of the tiles within the shape.

A tile consists of a number of components placed in an appropriate coordinate system, where $\overrightarrow{x}$ and $\overrightarrow{y}$ are classical orthogonal axes but where the vertical axis $\overrightarrow{z}$ is skewed and defined as $\overrightarrow{z} = (\frac{\sqrt{2}}{2}; \frac{\sqrt{2}}{2}; \frac{1}{2})$. These components are:

- A root module at the center of the tile, to which we will refer hereinafter as the *tile root* or simply *R* module (in white in Figure 2a).
- Two horizontal branches placed orthogonally across the $\overrightarrow{x}$ and $\overrightarrow{y}$ axes named the *X* and *Y* branches (in red and green in Figure 2b, respectively).
- Four upward branches ascending at a 45° angle and placed orthogonally to each other: the *Z* branch along the $\overrightarrow{z}$ axis, and the *RZ*, *RevZ*, and *LZ* branches at 90°, 180°, and 270°clockwise from *Z* (therefore following axes $(1, -1, 1)$, $(-1, -1, 1)$ and $(-1, 1, 1)$), respectively—all in light blue in Figure 2b.
- Four *support* modules: S_Z, S_{RevZ}, S_{LZ}, and S_{RZ}; one under each of the ascending branches at respective positions $(1, 1, 0)$, $(-1, -1, 0)$, $(-1, 1, 0)$, and $(1, -1, 0)$ relative to the tile root *R*. Supports are absolutely necessary for modules coming from below the tile so that they can traverse it vertically, as imposed by the *bridging constraint* (in yellow in Figure 2b).

3.2.2. Parameters and Conditional Structure

Let *b* be the parameter of the scaffold that defines the length of the branches of the tiles in number of modules. There is a lower bound on the value of *b* as under four modules in length tiles become too dense to allow module movement through all of their internal paths. Furthermore, an upper bound on the value of *b* is given by the mechanical strength of the connectors of the hardware *3D Catoms*, which is still undefined at the moment. Varying the length of tile branches allows control on the resolution of the target shape and thus the speed of self-reconfiguration, as higher *b* values would result in shapes with

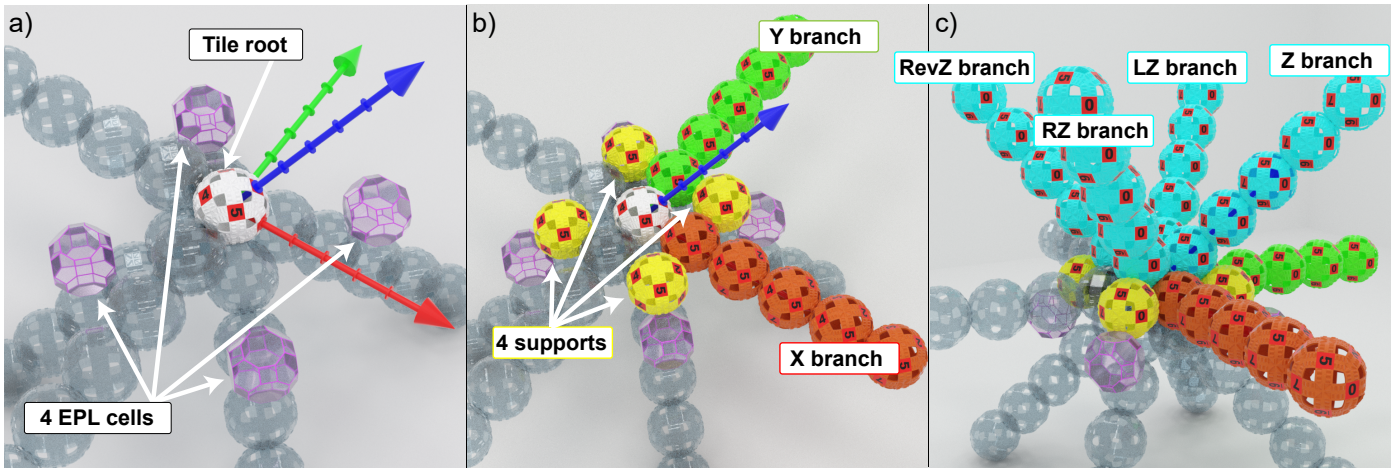

Figure 2: **Anatomy of a scaffold tile**: a) *Tile root* and vertical entry point locations, ingoing branches from parent tiles in transparency; b) *Supports* and outgoing horizontal branches; c) Outgoing upward branches.

lower density and fewer modules to place but also might result in a lower fidelity for the details of the shape. Throughout this paper, we will assume $b = 6$ as the length of the branches, as it is a very reasonable value mechanically.

If necessary, we will refer to a specific module of the tile with a name formed from its branch followed by its order within that branch (e.g., $RevZ_i$, where $i \in [1, b-1]$), or from S with the branch above it in subscript for *support* modules (e.g. S_{LZ}). Note that for any branch, the module of order 0 is always the tile root.

Furthermore, while b defines the *maximum* length l of the branch of a tile, a branch can have anywhere between 1 and b modules when part of the scaffold. A length of 1 means that the branch should not be grown for that tile, and only the tile root remains—tiles can, therefore, have a variable number of grown branches. A length of b means that the branch must be grown and it is likely that another tile will be grown from the tip of that branch once complete. A length anywhere between the two means that due to the geometry of the shape and placement of the tile within that shape, the full branch must not be grown and a child tile will not be grown from this branch.

To sum it all up, a tile always has a *root* module, and can grow between 0 and 6 branches, each between 2 (including the *tile root*) and b modules long. Furthermore, *support* modules need only be present if the upward branch below it and ingoing to its tile has been grown. Thus, a full tile (with all branches grown), can have anywhere between 1 (the R module), and $1 + ((b-1) \times 6) + 4$ modules.

Finally, we may also consider a number of additional branches opposing each of the aforementioned branches, which are named using *Opp* as a prefix, but these are special cases used for growing the shape in reverse that will be covered in due time.

3.2.3. Tile Construction Ordering

Due to the bridging constraint and the other motion constraints imposed on the modules, the tile cannot be built in any order[2]. There are a few rules that must be respected to limit the number of possible intersecting paths and avoid deadlock when building a tile:

1. The first component of the tile to be added will always be the *tile root*. This is even more crucial as the module claiming this component has a major role to play in the rest of the construction of the tile, as we will see,

2. Then, while the exact order depends on the location of the tile, the *support* modules and X_1/Y_1 modules must be built if present.

3. Horizontal branches must be built before all vertical branches are grown in order to prioritize the horizontal growth of the shape.

3.2.4. Connecting Tiles

Tiles assemble by connecting the tip of a fully grown branch (i.e., where $l = b$) to the tile root of another, as demonstrated in Figure 3.

Much like for the tile itself, we must enforce a construction order for the scaffold. This construction order follows the diagonal of the shape to be built. This means that the first tile to be built (*seed tile*) will be on a corner of the base of the object, and the last one will be the one on the opposite corner of its top layer. We arbitrarily choose this corner as the one with minimal x and y coordinates. Therefore, the growth of the shape will by default (there are exceptions) proceed along the $\overrightarrow{x}$ and $\overrightarrow{y}$ axes for a given plane, and from bottom to top. Then we can say that a tile that has been built before another that is connected to it (i.e., a *neighbor* tile) is a *parent tile* of the latter—conversely, the other is a *child* tile of the *parent*. A tile usually has more than 1 parent and up to 6 (one for each branch) if all ingoing branches are grown. Parent tiles are responsible for the growth

[2] See youtu.be/DjLwsrzA0MI?t=0 for an example of tile construction, in the case of a full tile.

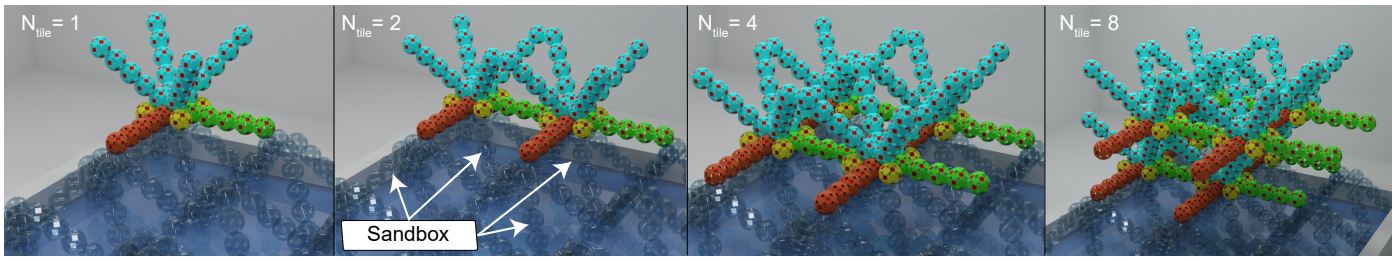

Figure 3: **Anatomy of the entire scaffold:** Breakdown of a sample scaffold consisting of an arrangement of 8 tiles with all branches grown, directly over the sandbox (branches from sandbox tiles in transparency).

of their children tile by feeding them modules through the connecting upward branch.

By generalization, we can generate a polytree, named *construction polytree*, representing the growth of a scaffold into a given object, where nodes are tiles and edges express construction precedence, with the seed tile as the root of the underlying tree.

3.2.5. Entry Points into the Tile

Module navigation from one tile to another is supported by special positions around the base of each tile, named *Entry Point Locations* (*EPL* hereinafter). There are 4 EPL for a tile, one on each of the ingoing upward branches (see Figure 2, with entry points in transparent pink and ingoing branches in transparent blue). Entry points are located over the second-last module of the ingoing upward branches, and right below the support module for that branch, which guarantees the reachability of the higher portion of the tile.

Any module entering a tile will do so from one of the four EPL, that is to say, that modules always flow through the scaffold from the lower tiles to the tiles above, and always do so through the connecting ascending branches—and therefore never through the horizontal branches, except for tile construction purpose. What motivates this mode of operation is that it limits to a maximum the number of possible intersecting paths along the scaffold, which lowers the risks of motion disturbance between modules and eases coordination. Entry points also have a crucial functional role to play in module navigation across the tiles and scaffold as a whole, which will be addressed later on.

As a consequence, a tile will have a maximum of four incoming flows of modules, which is the number of different usable paths leading to it. One of the main challenges is hence to coordinate these flows of modules such that they cannot intersect and impinge on each other's courses.

3.2.6. Sandbox

There is one last essential component of the system that needs to be introduced, and it is the *sandbox*. Indeed, this work focuses on the coordinated construction of the scaffold of a shape from an ordered reserve of modules rather than from one prebuilt shape to another, which will be further addressed. This reserve of modules, named *sandbox*, is located underneath the reconfiguration scene, and its main purpose is to introduce (or discard) as many modules as needed for the reconfiguration, at various ground locations of the scene (see Figure 3). The sandbox is structured internally exactly as the scaffold (with the same parameter b) and contains a surplus of modules along its branches, which can then be called in for the reconfiguration above. Therefore, all modules are initially introduced to the growing shape through one of the ground tiles at the top of the sandbox.

Furthermore, the sandbox is connected to an external apparatus that powers the whole system and provides the distributed program that the modules will execute during reconfiguration.

3.3. Self-Reconfiguration

3.3.1. Module States

Throughout the self-reconfiguration, modules will change state depending on their current task (e.g., navigating the scaffold in search of a position to be filled, coordinating flows of modules, or passively responding to messages). For each of these states, we can consider that modules execute a different distributed algorithm, which will be synthesized over the course of this section. Figure 4 summarizes the behavior of each module state and transitions between them; the roles of the messages mentioned are explained in the next section. All the possible module states are briefly shown below:

- ***Idle:*** This is the default module state in which modules are when they are not yet introduced into the reconfiguring system by the sandbox. They are simply waiting to be called in to partake in the reconfiguration. While this state will be left out from the rest of the article, it is shown here to emphasize that modules do not just appear from nowhere, but are already within the system as *Idle* modules in the sandbox.

- ***Free Agent:*** Once modules are introduced from the sandbox, they enter the *Free Agent* state. This corresponds to a module that has not been assigned a final position as a component of the scaffold yet and will navigate the structure until it encounters a tile that has a position to be filled.

Then, once a *Free Agent* has been assigned a scaffold component to fill and has reached it, it can enter one of two states depending on the location of the component within the tile.

- ***Beam:*** By default, it enters the passive *Beam* state. *Beam* modules only help forwarding messages between neighboring modules and regulate module flows to ensure that

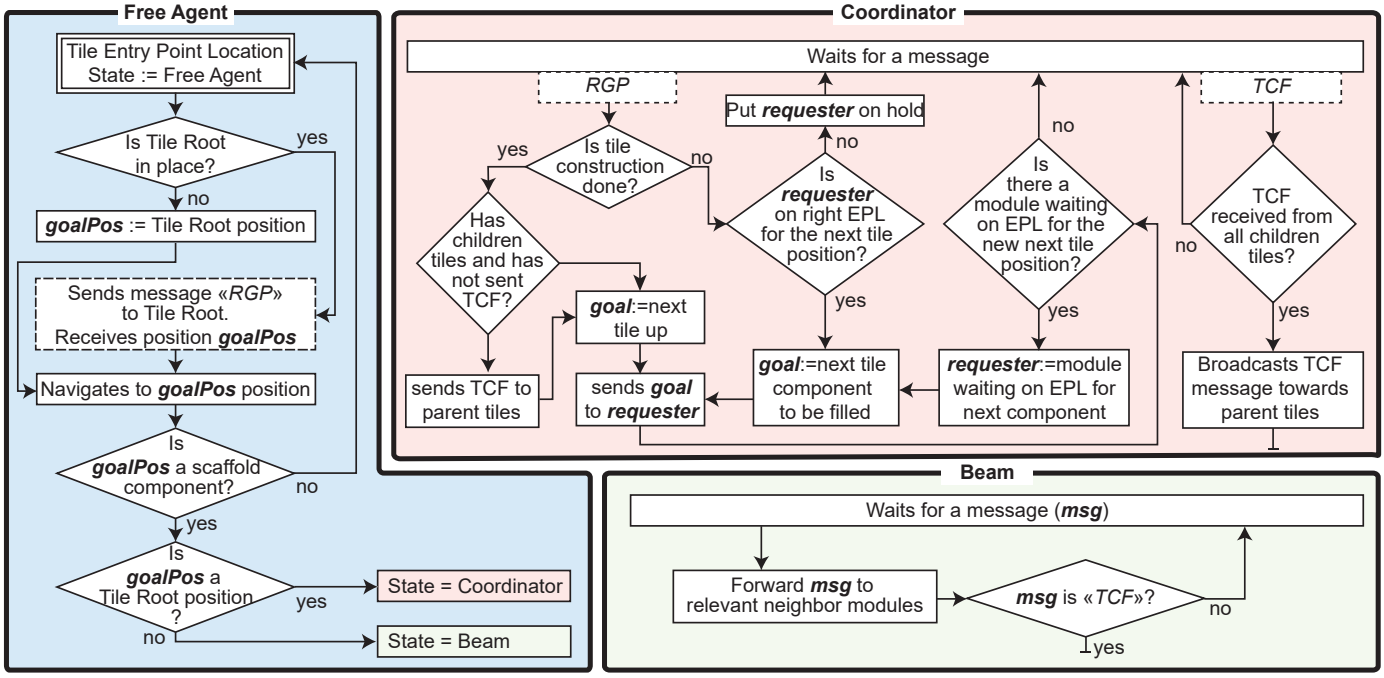

Figure 4: **Simplified view of the behavior of each module state and transitions between them.**

modules are not flowing too tightly, which could introduce collisions. *Beam* modules are either branch components or *support* modules.

- ***Coordinator:*** However, if the assigned component is the *tile root*, then the *Free Agent* module enters the *Coordinator* state. *Coordinators* are key modules of the self-reconfiguration, as their role is to assign a destination to *Free Agent* modules coming into their tile, either so that they go fill a component of that tile, or to reach one of the children tiles. *Coordinators* also schedule the construction of the tile to ensure that components are built in the right order and thus avoid collisions or deadlocks.

3.3.2. High-level Planning: Tile Construction Scheduling

Our proposed self-reconfiguration planning process[3] operates at two levels. On the one hand, the higher level is responsible for coordinating the construction of the scaffold at the level of the tile, directing module flows to the tiles that need to be constructed, when they need to.

Indeed, as previously mentioned, the growth of the goal shape proceeds according to a precise scheduling that ensures that structural deadlocks caused by an ill-formed tile construction ordering are avoided. A **diagonal growth direction** is enforced. Furthermore, based on this construction plan, the growth of the scaffold behaves according to a single crucial rule:

1. A tile can only begin its own construction once all of its ingoing branches (i.e., connecting it to its parent tiles) are complete.

[3]Please refer to the following video for a walkthrough of a reconfiguration into a cube: youtu.be/DjLwsrzA0MI?t=36.

The growth of the scaffold will, therefore, start from either a single ground position (seed tile, in the corner of the object) or multiple seed tiles in more complex shapes where the target object has a base that has 2-dimensional concavities. These initial ground tiles are tiles that rest onto the sandbox and have no ingoing horizontal branches, they are therefore ready to receive modules right away and start building. In the case of multiple initial tiles, the growth of the disjoint portions of the object will later synchronize at their junction based on the construction plan, or not synchronize at all if these portions are entirely disjoint.

The construction of a tile begins when a *Free Agent* module arrives at one of the EPL of the future tile (see Figure 5a, b & c), and claims the empty *tile root* position (Figure 5d). Once this module gets into position, it is ready to halt or direct any module that enters one of the EPL of the tile (see Figure 5d & e) in order to build the various branches and tile *supports* it needs (see Figure 5f). By default, when a module arrives at an EPL of a tile (whether it is already built or not), it halts there and requests a destination from the *Coordinator* of the tile it just entered (cf. Algorithm 2, ll. 9–12).

But before going further, let us introduce below the distributed messages on which high-level planning relies:

MESSAGE_NAME (*ACRONYM*) [DATA]

INGOING_BRANCH_READY (IBR) [*recipient*, *branch*]: This message is used to discover when all branches ingoing to a tile are complete. It is sent by the tip module of a fully grown branch that extends into a future new tile, identified by the *branch* data. It is sent to all the tips of the branches ingoing to that tile that are already in place. If a tip module receives an

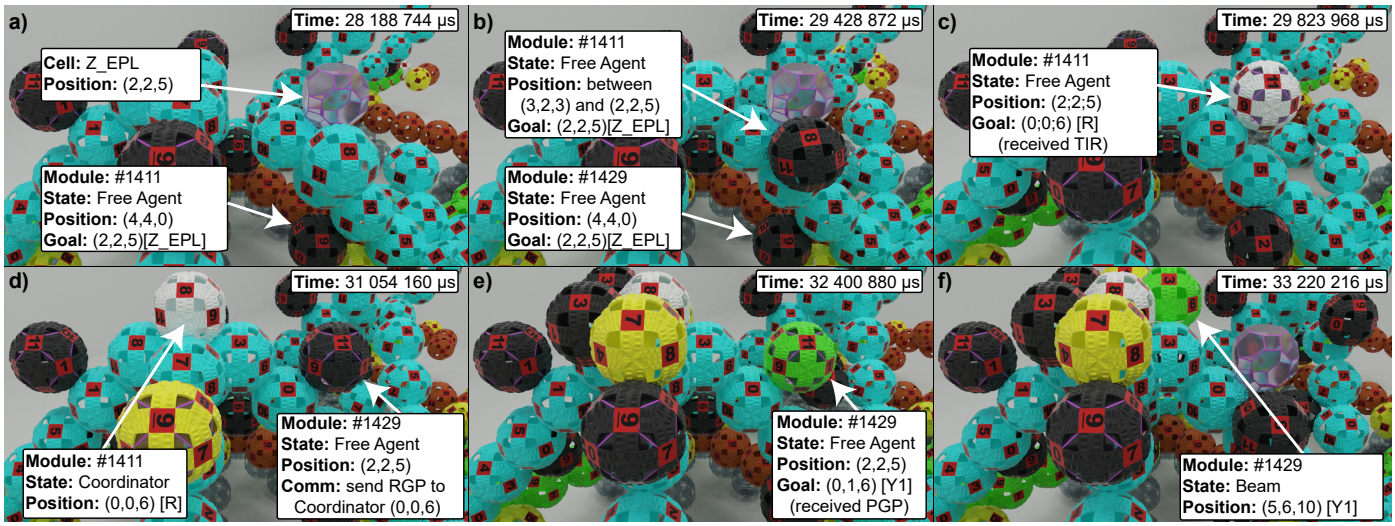

Figure 5: **Simulation snapshots of the *Free Agent* goal assignment process.** Two *Free Agents* (#1411 and #1429 drawn in black) climb up to the *Z_EPL* cell and get assigned their position in the future tile: *Tile root* for #1411 through a TIR message as the tile was missing its *Coordinator* (in white), and *Y1* for #1429 through a PGP/RGP transaction (in green). Each then reaches its final position in the tile, before updating its state accordingly.

IBR message from a new *branch*, it responds with an IBR message to notify the sender that its branch is in place too, which ensures that all tips have a correct representation of the current state of the tile at all times. IBR is not sent to the branch tips to which the sender is directly connected, as they can locally detect the presence of neighbor modules through their connectors.

TILE_INSERTION_READY (TIR) [∅]: When a branch tip module has received an IBR from all the branches ingoing to its tile, it can instruct a module waiting on the EPL over its branch to claim the free *tile root* position, by sending it a TIR message. Only one of the ingoing branches has this responsibility, which depends on the location of the tile. If no module is waiting on the EPL, then the branch tip stores the message and sends it to the next module that enters its EPL. This is used as a synchronization mechanism between parts of the scaffold growing concurrently, so as to ensure the correct implementation of the construction plan.

REQUEST_GOAL_POSITION (RGP) [*sender*]: RGP is sent to the local *Coordinator* by a *Free Agent* module when it arrives at the EPL of a tile, and is used to request a destination to continue the flow (cf. Algorithm 2, ll. 9–12)

PROVIDE_GOAL_POSITION (PGP) [*recipient*, *goal*]: This is the response sent by a *Coordinator* to a *Free Agent* module waiting on an EPL, when it receives an RGP message from it. After receiving an RGP message, the coordinator checks whether it needs resources from that ingoing branch at the time, and either puts the requesting module on hold until new resources are needed (in which case it simply differs the response) or responds right away with a goal position for that module (cf. Algorithm 1, ll. 1–10 and Algorithm 2, ll. 26–28). The *goal* positions can either be the position of a component of that tile that needs to be filled, or the position of one of the EPL of the children tiles. The latter occurs if all the components that are built from a branch are complete, in which case the requesting module is forwarded up to the child tile at the end of the branch located above its position. *recipient* is equal to the *sender* of the RGP request and is used for rooting the answer back. Each time that a coordinator responds with a PGP message providing a destination within its tile, it checks all the modules waiting on entry points that it has put on hold, and evaluates whether the new next position to be filled can be assigned to one of them (see, Algorithm 1, ll. 11–19).

COORDINATOR_READY (CR) [∅]: In some cases, arriving modules might send RGP to the tile before the *tile root* has taken its position. In such a case, the RGP messages cannot be delivered. Hence, in order to increase the robustness of the algorithm, the *Coordinator* sends a CR message to all the entry points of the tile once it gets into position, to which any module receiving it will respond by resending its RGP message.

TILE_CONSTRUCTION_FINISHED: (TCF) [∅]: When a *Coordinator* module from a leaf tile (in terms of the construction polytree) has finished constructing its tile, it sends a TCF message to the *Coordinators* of all of its parent tiles. When parents have finished constructing their own tile and have received a TCF message from all of their children, then they also send one to their parent. This is repeated until the seed tile of the scaffold has received all of its expected TCF, which marks the end of the self-reconfiguration and then terminates the algorithm.

At the start of the self-reconfiguration, there is nothing but an empty sandbox, with *Idle* modules waiting on the entry points of all of the ground tiles right above the sandbox. Then the seed module comes into place. It is the module that claims the *tile root* position of the corner tile acting as the seed for the self-reconfiguration.

Once in place, it gets into the *Coordinator* state. It then initializes based on its knowledge of the target shape and position within it, an ordered list of components to be filled to complete

its tile, and their matching entry points: it is the construction plan of the tile. Indeed, in every construction plan, each component is coupled with an EPL that will be exclusively used for supplying the module that will claim that location. More precisely, every branch or *support* has a preferred feeding EPL by default: the EPL right below them for upward branches and *supports*, Z_{EPL} for the *root R*, RZ_{EPL} for *X* branch, and LZ_{EPL} for *Y* branch. However, depending on the location of the tile to be built, and thus its set of ingoing branches, some of these EPL might not exist for the tile. Therefore, alternate EPLs might need to be used in each of these cases.

From there on, the *Coordinator* waits for *Free Agent* modules to enter an EPL of its tile and send an RGP message (see ll. 9–12 Algorithm 2). If the sender is on the EPL of the next component to be filled, it directs it right away to its goal component or otherwise awaits a request from the correct EPL (see ll. 1–10 Algorithm 1). However, once a *Coordinator* receives a request from an EPL from which no more components will be built, it responds right away and directs the incoming module to the EPL of the branch directly above it, thus forwarding it to one of its children tiles to continue the construction process. This is repeated until all the tiles constituting the scaffold are complete.

Furthermore, the *IBR* / *TIR* messaging system ensures that the precedence in the construction order of tiles is respected, by enforcing synchronization points between portions of the goal shape growing concurrently.

This process corresponds to Algorithm 1 for the point of view of the *Coordinator*, while the point of view of the *Free Agent* appears later in Algorithm 2. Note that in all presented algorithms, low importance messages and handlers have been left out. Figure 4 also summarizes the high level reconfiguration process.

Algorithm 1: Distributed control algorithm pseudo-code for the *Coordinator* module role.

```
1  Msg Handler REQUEST_GOAL_POSITION(RGPmsg):
2      epl = getEPLForPosition(RGPmsg.srcPos);
3      if plan.isOver() then
4          goalPos = getEPLForBranchAbove(epl);
5      else if plan.nextComponentIsFedBy(epl) then
6          goalPos = plan.popNextComponent();
7      else
8          moduleWaitingOnEPL[epl] = true; return;
9      sendMsg(sender, PGP(RGPmsg.srcPos, goalPos));
10     checkModulesWaitingOnEntryPoints();

11 Function checkModulesWaitingOnEntryPoints:
12     do
13         moduleAwoken = false;
14         foreach epl ∈ getAllEntryPoints() do
15             if plan.nextComponentIsFedBy(epl) and
                moduleWaitingOnEPL(epl) then
16                 goalPos = plan.popNextComponent();
17                 sendMsg(sender, PGP(epl.pos, goalPos));
18                 moduleAwoken = true;
19     while moduleAwoken = true;
```

3.3.3. Low-level Planning: Module Navigation

On the other hand, the lower level of planning defines how a module navigates the structure from its current location to its assigned goal position within the tile it is currently traversing. This is now entirely local to the module, based on its current neighborhood, origin, and destination. The high-level planning process thus handles the navigation between tiles, by providing each module with its origin (the position of an EPL) and its destination (the position of a component or of an EPL above), while the low-level planning handles the navigation within the tiles themselves. It does so by the use of local motion rules, that match a series of individual displacements (rotations between lattice positions), to the local context of a *Free Agent* module.

It is worth noting that in principle any low-level planing method could work, whether stochastic or deterministic as ours, as long as it provides a solution for safely displacing a *Free Agent* module from its current position to its assigned destination.

In more concrete terms, each local rule matches a tuple $\langle neighborhood_{bin}, EPL, destination, step \rangle$ to a displacement vector $\overrightarrow{disp}$, where each element corresponds to:

- $neighborhood_{bin}$: A 12-bit word that shows the current state of each of the connectors of the module, ordered according to their default orientation. A 1 means that the connector is connected, while 0 means that there is no neighbor connected to it.
- *EPL*: The last EPL traversed by the current module, used as the origin of the motion path.
- *destination*: The coordinates of the goal component or EPL that the module is trying to reach as the destination of the motion path.
- *step*: The current step of the multi-motion displacement between the origin and the destination—i.e., the first rotation would be step 1, the second step 2, etc... Usually, a motion path within a tile with $b = 6$ has between 2 and 9 individual steps.
- $\overrightarrow{disp}$: The displacement that the mobile module will have to perform in order to reach the next position in the current motion path.

Therefore, whenever a module must perform a motion, it checks its local rules database against its current context and obtains the next rotation it should perform. If the rule matching processes fails, probably due to the module being early at its location (hence with a not-yet-ready local neighborhood), the module waits for its local neighborhood to update (marked by an ADD_NEIGHBOR or REMOVE_NEIGHBOR event) and re-attempts matching (see ll. 23–25 Algorithm 2).

The exact algorithm used by *Free Agent* modules to navigate between two distant positions is summarized in Algorithm 2, and lines 13–22 specifically address local-rule matching.

Algorithm 2: Distributed control algorithm pseudo-code for the *Free Agent* module role.

```
1  Event ROTATION_END: ARRIVED_FROM_SANDBOX:
2      if myPos == goalPos then
3          if isTileComponent(myPos) then
4              agentRole = agentRoleForComponent(myPos);
5          else reachedNewTileEntryPoint() ;
6      else
7          step++;
8          planNextRotation();

9  Function reachedNewTileEntryPoint():
10     coordinatorPos = getNearestTileRootFrom(myPos);
11     nextHop = findSupportOrBranchTipNeighbor();
12     sendMsg(nextHop, RGP(myPos));

13 Function planNextRotation():
14     ngbh = getNeighborhood();
15     disp = matchRules(ngbh, lastEPL, goalPos, step);
16     if disp then
17         nextPos = myPos + disp;
18         pivot = findPivotForMotionTo(nextPos);
19         sendMsg(pivot, PLS(myPos, nextPos));
20         waitingForLocalRuleMatch = false;
21     else
22         waitingForLocalRuleMatch = true;

23 Event ADD_NEIGHBOR: REMOVE_NEIGHBOR:
24     if waitingForLocalRuleMatch then
25         planNextRotation();

26 Msg Handler PROVIDE_GOAL_POSITION(PGPmsg):
27     step = 0; goalPos = PGPmsg.goalPos;
28     planNextRotation();

29 Msg Handler GREEN_LIGHT_ON(GLOmsg):
30     rotate(nextPos, pivot);
```

The main drawback of this approach, however, is that the number of local rules that are necessary to cover all possible paths from an EPL to a component reachable from that entry point is very high. For that reason, designing rules by hand is a tedious process, and the sheer number of rules might overload the limited memory of the modules. Thus, improvements on the current format of the rules should be researched in order to reduce their memory footprint and attempt to factorize eligible rules.

3.3.4. Motion Coordination Algorithm

Finally, there is one last process that takes place during self-reconfiguration and that needs to be introduced, and it relates to motion coordination between mobile modules. In our work, motion coordination and collision avoidance are ensured through two methods: a passive rule-based mechanism and an active process. The former has already been introduced, as it relates to the ordering in the construction of the tile, which reduces the likelihood that module paths will intersect during construction. There is however an additional measure that must be taken to ensure that modules cannot impinge on their respective motions, and that is to leave a gap between moving modules at all times, an idea previously explored in the context of 2D self-reconfiguration by Naz et al. (2016). This is necessary because when modules move right next to each other, one of them might get blocked between two modules, and due to the bridging constraint cause a deadlock of the construction process. This coordination is message-based and relies on a green-light handshake between modules seeking to move, their motion pivot, and their future latching point. Three different kinds of messages are required, which are detailed below:

MESSAGE_NAME (*ACRONYM*) [DATA]

PROBE_LIGHT_STATE (PLS) [*sender*, *motionTarget*]: Sent by a module seeking to move to location *motionTarget*, one rotation away from the sender. The sender *Free Agent* sends this message to the pivot module it plans to use for its motion to *motiontarget* (see Algorithm 2, l. 19). Then, the destination of the message is discovered during routing, and the message forwarded to it (see Algorithm 3, l. 21). This destination module (hereinafter *light pivot*) is the module further along the motion path of the sender, among the modules to which it will connect upon reaching *motionTarget*. When a *Beam* module receives a PLS message, it computes the *light pivot* for the requested motion based on its local knowledge of the neighborhood. If it is not the *light pivot*, it forwards the request to it.

GREEN_LIGHT_ON (GLO) [*recipient*]: However, if it is the one that should respond, it checks whether it already has a *Free Agent* module on one of its interfaces (*red light* state). If there is none, then it means it is in the *green light* state, and it responds right away with a GLO message to the *sender* of the PLS request (see Algorithm 3, ll. 15–16). Otherwise, it memorizes that the *sender* module is waiting to move towards it and turns into the *orange light* state and differs its response until it is free of its current *Free Agent* neighbor (see Algorithm 3, ll. 1–5 and 18–19).

FINAL_TARGET_REACHED (FTR) [∅]: Finally, the FTR message is sent by a module that has performed a final motion to take its place as a scaffold component and that is adjacent to the *light pivot* of the module. In this scenario, FTR is sent to the *light pivot* to inform it that it can now turn back to the *green light* state even though the two modules are still connected to each other.

There are therefore three different states in which a *Beam* module can be: *green light*, if it is ready to receive a new *Free Agent* on one of its connectors; *red light*, if it already has a *Free Agent* module connected to it; or *orange light*, if it was in the *red light* state but there is also another module that is waiting for the pivot to turn back to the *green light* state to perform its motion.

The transition between this states is not only assured via messaging, as modules also monitor their interfaces to react to any

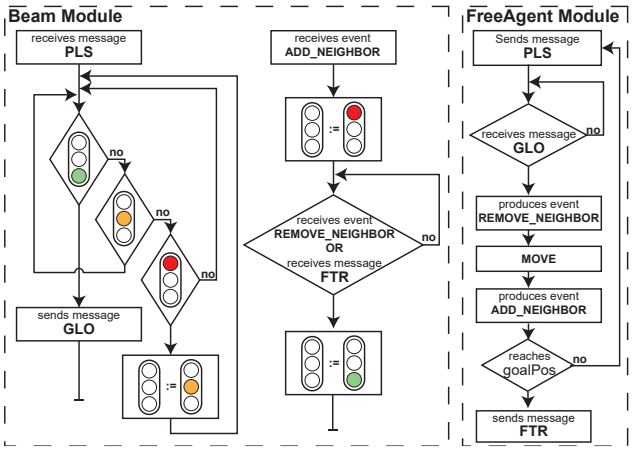

Figure 6: **Light state transition diagram**. The two *Beam* routines are executed concurrently on pivot modules.

connection or *disconnection* event and update their state accordingly. Thus, if a *Beam* modules notices a new connection from a *Free Agent* (characterized by a neighbor with a position that is not part of the scaffold), it turns *red* (see Algorithm 3, l. 5). Conversely, if it notices a disconnection from a *Free Agent* module, it turns its state back to *green light* (see Algorithm 3, ll. 6–7). These mechanisms are summarized in Figure 6 and the pseudo-code for it from the points of view of the *Free Agent* and *Beam* modules can be seen on Algorithms 2 and 3, respectively. This protocol has been shown to be robust to stochastic variations in the rotation duration of individual modules in (Thalamy et al., 2019c).

Algorithm 3: Distributed control algorithm pseudo-code for the *Beam* module role.

```
1  Function setGreenLightAndResumeFlow():
2      if state == ORANGE then
3          sendMessage(sender, GLO(waitingModule));
4      state = GREEN;

5  Event Handler ADD_NEIGHBOR: state = RED ;

6  Event Handler REMOVE_NEIGHBOR:
7      setGreenLightAndResumeFlow();

8  Msg Handler REQUEST_GOAL_POSITION(RGPmsg):
9      forwardMsgTowards(coordinator, RGPmsg);

10 Msg Handler PROVIDE_GOAL_POSITION(PGPmsg):
11     forwardMsgTowards(PGPmsg.recipient, PGPmsg);

12 Msg Handler PROBE_LIGHT_STATE(PLSmsg):
13     dst = computeLightPivotForTarget(motionTarget);
14     if dst == self then
15         if state == GREEN then
16             sendMsg(sender, GLO(PLSmsg.srcPos));
17         else
18             state = ORANGE ;
19             waitingModule = PLSmsg.srcPos;
20     else
21         forwardMsgTowards(dst, PLSmsg);

22 Msg Handler FINAL_TARGET_REACHED(FTRmsg):
23     setGreenLightAndResumeFlow();
```

4. Building Simple Pyramids

Now that all the fundamental elements of our work have been introduced, this section will present as a case study the construction of a square pyramid from the sandbox.

4.1. Motivations

The square pyramid of size h, or h-pyramid, is a pyramid with a square base of dimensions h tiles and a height of h tiles. This is the first shape for which we have implemented our self-reconfiguration method, as it is the most simple shape that can be built with it, due to the geometry of individual tiles.

Indeed, there are reasons why this is so:

1. As the dimensions of h-pyramids are multiples of b, all branches are either grown fully or not grown at all, there is no need to deal with incomplete branches.

2. Then, due to the geometry of h-pyramids, all the tiles of the shape will have 4 ingoing upward branches. This means that we can simply assign one EPL to each of the *supports* and branches to be grown—e.g., the *RevZ* branch can always be built from the *Z* ingoing branch below, no need therefore to handle any additional motion path from another EPL to *Z*. Not only does it limits the number of local motion rules, but also the possible tile construction scheduling to just a few cases.

4.2. Assumptions

- All modules have complete knowledge of the target shape and can geometrically compute whether a coordinate belongs to the target shape, and if it does, which scaffold component it corresponds to.

- Modules rely on a relative coordinate system for which the origin is the *R* module of the current tile, both for *Free Agent* and scaffold *Beam* modules.

- The goal h-pyramid is positioned in such a way that the corners of the base of the pyramid are at a *tile root* position.

- Construction starts from the corner of the base of the pyramid with minimal x and y coordinates.

4.3. Self-Reconfiguration

The self-reconfiguration proceeds exactly as explained in Section 3, by starting from a corner of the base of the pyramid and growing tiles in order until the tile at the tip of the pyramid has finished constructing. Nevertheless, we introduce in this section two possible variants of the reconfiguration algorithm.

4.3.1. Surplus Modules Management

There are two possibles variants of the algorithm that can be used. In the first one, named *Continuous Flow Algorithm* (Thalamy et al., 2019c), modules continuously flow through the structure from the sandbox to any available path in the structure, even though they might not be needed. The flow is regulated by the *Coordinators* depending on their construction needs, and by the light-based local coordination mechanisms of *Free Agents*. In this scenario, the goal shapes contain a surplus of modules on each of the branches of the scaffold at the end of the reconfiguration, modules that could then be further used for evolving the shape or covering its surface. The amount of modules in excess is a function of the length of the branches b, and of the number of upward branches $UBranches$ in the shape, which can be expressed as:

$$E = N_{UBranches} \times \frac{b}{2} - 2$$

On the other hand, in the second variant of the algorithm, named *No Surplus Algorithm*, low-level *Coordinators* from the base of the scaffold (connected to the sandbox) compute the exact requirements of the whole portion of the scaffold that will receive their flow of modules, and only send what is needed. This can be computed at the start of the reconfiguration by these *Coordinators* as they have full knowledge of the goal shape. They compute it using a centralized, local, and recursive tree counting algorithm. The base *Coordinators* virtually explore the set of children of their tile and their respective children recursively, for each tile computing the number of components that will be constructed from the ingoing branch through which their fed modules will flow. By summing the resource needs of all of the tiles that their flow will reach, the total number of modules that need to be called in from this particular section of the sandbox can be derived. In the case of faulty modules, a message-based resource request system could be implemented to request replacement modules and increase the robustness of the algorithm. This is the version that will appear in the following experiments.

Both have identical algorithmic complexities however, as they are equivalent; the only difference is in the number of modules involved in the self-reconfiguration process.

4.4. Analysis

The analysis in this section relies on the same reasoning as the analysis for our previous heavily synchronized version of this algorithm (Thalamy et al., 2019a).

4.4.1. Number of modules

This section provides a brief analysis of a scaffolded h-pyramid, and the performance of our algorithm on this class of shapes.

Throughout this section and the rest of the paper, we will use the term *tile layer* to designate a horizontal section of the object that is composed of all tiles whose root is on the same horizontal plane. Let $N_{tiles}(i)$ denote the number of tiles at tile layer i, with tile layer 0 as the base of the object. We have:

$$N_{tiles}(i) = (h-i)^2 \tag{1}$$

From $N_{tiles}(i)$, we can express the total number of tiles in a h-pyramid as:

$$N_{tiles} = \sum_{i=0}^{h-1} N_{tiles}(i) = h^3 - 2h^2 + h \tag{2}$$

Then let $N_{modules}(i)$ denote the number of modules in tile layer i of the h-pyramid. By counting the number of roots, supports, horizontal, and upward branches on a given layer, we find:

$$\begin{aligned} N_{modules}(i) =& (h-i)\left[(h-i-1)b + 1 + (h-i-1)(b-1)\right] \\ &+ 4(b-1)(h-i-1)^2 + 4(h-i)^2 \end{aligned} \tag{3}$$

By summing the number of modules on each layer of an h-pyramid, we obtain the total of number of modules in the shape:

$$\begin{aligned} N_{modules} &= \sum_{i=1}^{h} N_{modules}(i) \\ &= (2b - \frac{1}{3})h^3 + (\frac{9}{2} - 2b)h^2 + \frac{5}{6}h \end{aligned} \tag{4}$$

As a point of comparison, we provide the total number of modules composing a filled h-pyramid below:

$$\begin{aligned} N_{modules}^{filled} &= \sum_{i=1}^{b(h-1)+1} i^2 \\ &= \frac{2b^3(h-1)^3 + 9b^2(h-1)^2 + 13b(h-1) + 6}{6} \end{aligned}$$

It shows that it takes $\frac{b^2}{6}$ fewer modules to build a scaffolded shape than the corresponding filled one. This saving has a tremendous impact on the duration of self-reconfiguration.

4.4.2. Complexity Analysis

We now aim to determine the complexity of the reconfiguration time of our method. In this section and the results discussed thereafter, time is expressed in *time steps*, where a single *time step* represents the average duration of a *3D Catom* rotation.

We assume that the time required to complete the construction of a single tile is constant in the case of the h-pyramid, as it only depends on the number of modules that compose it. In the explanations that follow, we will take the time of arrival of the tile root R of tiles as a reference point, especially the one of the first tile of each tile layer (*seed tile*). This is because these tiles act as synchronization points for the construction of the object. In the case of the h-pyramid, the top tile layer will consist only of the seed tile for that layer, which synchronizes the construction of the whole object.

Also, our analysis relies on the aforementioned construction polytree of the pyramid with the seed tile of the base as root (coordinates $(0,0,0)$), and with the seed tile of the top layer as

the only leaf (coordinates $(0, 0, (h-1)b)$). Let a critical path l_c of the construction polytree be, among the longest path between these two nodes, a path for which there will be no waiting time caused by synchronizations during reconfiguration. The branches composing the critical path are thus always the last ones to arrive at any synchronization point. In the case of the pyramid, there are two critical paths: along the $\overrightarrow{x}$ axis border of the base between $(0, 0, 0)$ and $((h-1)b, 0, 0)$ positions, followed by the opposite $\overrightarrow{y}$ axis border of the base between $((h-1), 0, 0)$ and $((h-1)b, (h-1)b, 0)$ positions, and up the backward edge to the top tile of the pyramid between $((h-1), (h-1)b, 0)$ and $(0, 0, (h-1)b)$ positions; or along the $\overrightarrow{y}$ axis border first, and then the opposite $\overrightarrow{x}$ axis and backward edges.

Theorem 1. *The height of the construction polytree of the h-pyramid is* $3(h-1)$.

Proof. If we follow a critical path of the pyramid, we see that the depth in the construction polytree between the seed tile and the end of one of the lateral edges (x or y from the last paragraph) of the base is $h-1$. Then the depth between the latter and the corner of the base opposing the seed tile is again $h-1$. Finally, the depth from this corner of the base to the top of the pyramid through the back edge is also $(h-1)$.

Therefore, the total height of the construction polytree of the h-pyramid is $3(h-1)$, which is in $O(h)$. □

Let $seed_i$ and $seed_{i+1}$ the seed tiles of layer i and $i+1$ from the ground, respectively. In the case of the h-pyramid, the critical path from $seed_i$ to $seed_{i+1}$ follows the Y branch of $seed_i$, then the X branch from the tile at $(0, b, 0)$ from $seed_i$, and finally through the *RevZ* branch from the tile at $(b, b, 0)$ from $seed_i$. As the time to build a tile is constant for a given value of b and therefore only depends on b, we can deduce, from an analysis of the set of local rules and from the scheduling between components that lead to the construction of a tile, the time in timestep it takes to traverse this critical path from $seed_i$ to $seed_{i+1}$ in the construction polytree. This corresponds to the time (in time steps) it takes for the tile root of the seed tile on layer i of the shape to come into position, which can be expressed as:

$$T_{tile} = 16(b-1) \tag{5}$$

And as the height of the construction polytree is $O(h)$, the total reconfiguration time can be expressed as:

$$T = \sum_{i=1}^{(h-1)} 16b - 16 = 16(b-1)(h-1) \tag{6}$$

As T_{tile} does not depend on i, we conclude that T is linear in the height of the pyramid h—i.e., the reconfiguration time is $O(h)$ time steps.

Finally, the reconfiguration time must be expressed relative to the number of modules in the shape:

Theorem 2. *The time complexity of our self-reconfiguration method is* $O(N^{\frac{1}{3}})$ *for the construction of a scaffolded h-pyramid.*

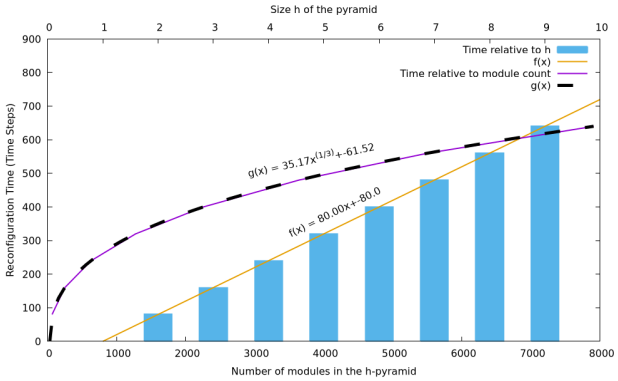

Figure 7: Reconfiguration time relative to tile count and module count for increasing sizes of h-pyramid

Proof. Using Equation 4, and considering that the parameter b is a positive constant, we can assume that there exist two positive real numbers $\{p, q\} \in \mathbb{R}^2$ verifying: $p \times h^3 < N < q \times h^3$.

Then, we deduce bounds for h:

$$\left(\frac{N}{q}\right)^{\frac{1}{3}} < h < \left(\frac{N}{p}\right)^{\frac{1}{3}}$$

Combining with previous Equation 6, and with $b = 6$, we deduce bounds for the motion time T:

$$80\left(\frac{N}{q}\right)^{\frac{1}{3}} - 80 < T < 80\left(\frac{N}{p}\right)^{\frac{1}{3}} - 80$$

We conclude that the reconfiguration time is $O(N^{\frac{1}{3}})$ time steps, with N the number of modules in the h-pyramid. □

4.5. Experiments

As mentioned in Section 1, all the simulations presented in this paper were performed in *VisibleSim* (Piranda, 2016), a discrete event simulator for large modular robotic systems.

We performed simulations of our algorithm on increasing sizes of h-pyramid, with $1 < h < 10$ to verify our findings from the analysis section. Figure 7 shows the simulation results, to which we have added a plot of the fit of both curves, demonstrating that reconfiguration time is indeed in $O(h)$ and $O(N^{\frac{1}{3}})$ for the h-pyramid.

5. Semi-Convex Generalization

Now that self-reconfiguration using our method has been demonstrated on a simple shape, this section will present how these results can be generalized to a greater class of shapes, to which we will refer to as *semi-convex* shapes.

5.1. Motivations and Challenges

This class of shape comprises all shapes in which no layer of the shape is larger than that of the base in number of tiles. That is to say, given a shape of height h and with $l_x(i)$ and $l_y(i)$

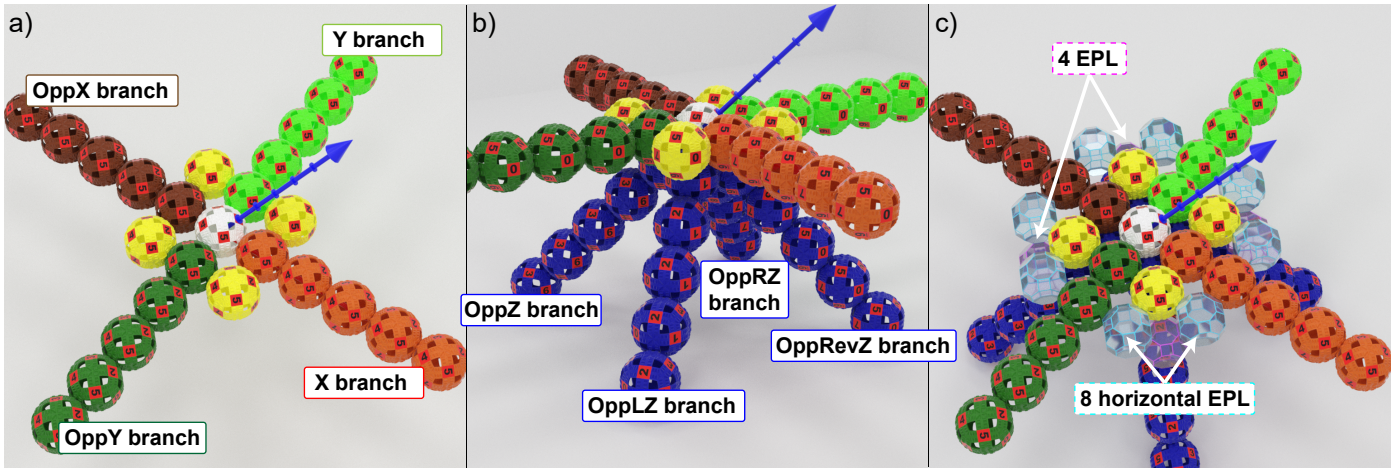

Figure 8: **Extended anatomy of a scaffold tile:** a) Opposing outgoing horizontal branches *OppX* and *OppY*; b) Opposing outgoing vertical branches, downward; c) Addition of 8 horizontal entry point locations (in transparent blue) for horizontal feeding, along with the 4 standard vertical EPL (in transparent pink).

respectively the width and depth of the tile layer i in number of tiles: $\forall i \in [1, h-1],\ l_x(i) \leq l_x(0) \wedge l_y(i) \leq l_y(0)$.

This is a subclass of convex shapes. The reason for not covering all convex shapes at this point is that with our system it is not harder to build a concave shape than a convex shape that does not fit our criteria since they have the same properties when taking the sandbox into account. Indeed, the difficulty lies in the fact that the shape cannot be considered in isolation from its construction substrate, whatever the surface or contraption it would rest on during reconfiguration. Therefore, in our case, an object can only be considered convex if the union of the object and the portion of the sandbox that is directly below it, is convex.

However, this is a more challenging problem than building pyramids, see below:

1. Tile branches can now have a length anywhere between 1 and b modules.
2. Because of varying branch lengths and the non-pyramidal geometry of the scaffold, not all tiles have 4 ingoing upward branches. However, due to our shape constraint, any outgoing upward branch is guaranteed to have a matching ingoing upward branch below it, and therefore can be directly fed by the EPL below as was before—e.g., all *RevZ* branches will have an ingoing Z branch below it, and thus a Z_{EPL} to feed it modules. However, horizontal branches might not have their default ingoing upward branch in place and thus a new construction scheduling and set of local rules must be produced in each possible case.
3. Additionally, there might be X and Y branches around the border of shapes with no tile preceding them along the $\overrightarrow{x}$ or $\overrightarrow{y}$ axes. This means that it will be the responsibility of the next tile along the axis to construct it. This also requires new additions into the system, such that a way to refer to these branches and construct them in the reverse direction, new local rules, and additional construction scheduling constraints.
4. There can now be multiple seed tiles for each tile layer of the object, growing portions of the shape in parallel and whose growth will need to synchronize at their junctions. A tile is a seed tile if it has no parent at the end of an ingoing horizontal branch. Therefore seed tiles can start building as soon as all their ingoing upward branches are complete, as is immediately the case with the seed tiles directly above the scaffold. Growing multiple disjoint sub-parts of the object in parallel appears trivial, as the placement of seed tiles can be easily inferred from the above criteria, and the synchronization aspect is already built into the existing high-level construction rules.

5.2. *Updated Model and Assumptions*

5.2.1. *Specifying Shapes*

In order to perform self-reconfiguration with full knowledge of the goal shape, all that modules need to know is the position of the origin tile (first tile at $x = y$, for them to agree on a coordinate system); a lookup function that can quickly answer on whether a given coordinate is inside or outside of the shape; and a geometric rule matching engine that can derive scaffold relevant information from coordinates (e.g., coordinate (a, b, c) is component X_4 of the tile whose tile root is at position $((a-4), b, c)$).

In the previous case with h-pyramids, the goal shape could simply be a number of geometrical rules describing an h-pyramid, with a straightforward lookup function, and an input parameter h. However, manually designing a set of geometrical rules for each goal shape in our class of shapes would be cumbersome and impractical. A generic way to describe shapes and represent them in the memory of the modules is hence required. For this purpose, we propose to use a *Constructive Solid Geometry* (CSG) tree model, which has already been successfully applied in the context of large-scale modular robotic systems in (Tucci et al., 2017). CSG is used to describe solids using a combination of simple shapes and Boolean set operators arranged as a tree. This description is remarkably compact for objects with a low level of detail, and it is scalable by design

thanks to its vectorial nature. Furthermore, it is very efficient at looking up whether a position is inside the object, which is critical in our case.

Therefore, by simply providing the modules with the two lookup functions from the scaffold geometry engine and the CSG one, modules can seamlessly compute and build the scaffolded version of the input goal shape.

5.2.2. Addition of Reversed Horizontal Branches

As stated in item 3 of the last section, some tiles of the scaffold will now need to construct horizontal branches to their left or behind them, which were previously always built by parent tiles. For this purpose, we introduce two new **outgoing** branches to the set of branches of a tile, named *OppX*, in reverse along the $\overrightarrow{x}$ axis (thus, following axis $(-1, 0, 0)$), and *OppY*, in reverse along the $\overrightarrow{y}$ axis (thus following axis $(0, -1, 0)$). Of course, these are by nature the same branches as the **ingoing** *X* and *Y* branches, but there is a logical distinction in that their directions are opposite and their tile of belonging is different (see Figure 8a).

Tiles will need to grow an *OppX* branch if the tile root of the tile before it along the $\overrightarrow{x}$ is not in the shape, but the branch between the two is at least one module long (not counting the *R* modules). Accordingly, tiles will need to grow an *OppY* branch if the tile root of the tile before it along the $\overrightarrow{y}$ is not in the shape, but the branch between the two is at least one module long.

Finally, the first modules of the *Opp* branches (i.e., *OppX1* and *OppY1*) will always have to be grown as early as possible in the construction process, as it might not be possible to insert them once other branches have started their growth. By default, *OppX* is built using the LZ_{EPL} and *OppY* using $RevZ_{EPL}$.

5.2.3. Handling a Variable Number of Ingoing Branches

As previously mentioned, in this class of shapes any outgoing branch from a tile will always have a corresponding ingoing branch right under it. Therefore, the same feeding principles as before can be applied. The difference, however, is that for horizontal standard and *Opp* branches, the preferred feeding branch is not always present and thus local rules guiding motion from any ingoing upward branch to any horizontal branches must be added to the database.

5.3. Analyses

In this section, we will study how our improved reconfiguration algorithm performs theoretically on a cubic shape, and see how these results can then be extended to all the shapes from the *semi-convex* class.

5.3.1. Cube Case Study

Considering a cube of length $l = (h-1) \times b + e$ with h the number of *tile root* modules along one of its edges, b the length of a tile branch, and e the number of modules on each branch of the tiles with incomplete branches ($1 \leq e \leq b$). For example in Figure 9, $l = (4-1) \times 6 + 3 = 21$.

Theorem 3. *The total number of tiles in the cube is h^3, and the number of modules is $N = O(h^3)$.*

The complexity of the reconfiguration time of the $l \times l \times l$ cube is $O(h)$, and as a consequence it is $O(N^{\frac{1}{3}})$.

Proof. Let N be the total number of modules. We express these number as the sum of four groups of modules: modules on even tile layers (N_{even}); modules on odd tile layers (N_{odd}); modules on the top tile layer (N^{top}_{even} if h is even or N^{top}_{odd} otherwise); and *support* modules from the border of the bottom tile layer (N_0). Then we get:

$$\begin{aligned} N =& N_0 + \left\lfloor \frac{h-1}{2} \right\rfloor N_{odd} + \left\lfloor \frac{h}{2} \right\rfloor N_{even} \\ &+ (h \bmod 2)\, N^{top}_{odd} + ((h+1) \bmod 2)\, N^{top}_{even} \end{aligned} \tag{7}$$

For example in Figure 9: $h = 4$, $e = 3$ and $b = 6$ then

$$N = N_0 + 1 \times N_{odd} + 2 \times N_{even} + N^{top}_{even} = 1426$$

We express the components used in Equation 7 in terms of h, b and e:

$$\begin{aligned} N_0 &= 8h - 4 \\ N_{even} &= (6b-1)h^2 + (e - 10b + 9)h + 4b - 4 \\ N_{odd} &= (6b-1)h^2 + 2(5e - 5b - 4)h + 4b - 6e + 6 \\ N^{top}_{even} &= N_{odd} + 4(e-b)(h-1)^2 - 2(e-1)(4h-3) \\ N^{top}_{odd} &= N_{even} + 4(e-b)(h-1)^2 \end{aligned} \tag{8}$$

The length (in number of modules) of the critical construction path l_c of the cube (drawn in red in Figure 9) is obtained by a depth first traversal of one of the critical sub-trees, yielding the expression:

$$l_c = \begin{cases} 4(h-1)b + \frac{b}{2} + e & \text{if } h \text{ is even} \\ 4(h-1)b + e & \text{if } h \text{ is odd} \end{cases} \tag{9}$$

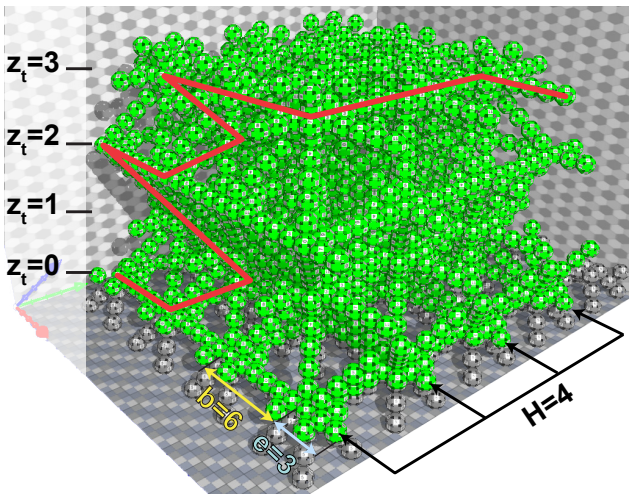

Figure 9: Example of a Cube of length $l = (4-1) \times 6 + 3 = 21$. The critical path l_c of length 78 modules is drawn in red.

Then considering that there are moving modules at each step of the simulation along this critical path, we can deduce that

the reconfiguration time is proportional to l_c. Following the critical path for the cube in the same manner as in Section 4.4, we obtain $O(h)$ as the height of the tree and thus an $O(N^{\frac{1}{3}})$ reconfiguration time for cubic shapes. □

5.3.2. Generalization to Semi-convex Shapes

Theorem 4. *The complexity of the reconfiguration time into any semi-convex shape is $O(h)$ relative to the dimensions h of the shape in number of tiles, and $O(N^{\frac{1}{3}})$ relative to the number of modules in the shape N.*

Proof. Let (l_x, l_y) be the dimensions of the base of the shape in number of modules, and l_z its height. We can express each dimension in the same manner as we did for the length of the cube, but with different h and e values for each dimension, which gives:

$$\begin{aligned} l_x &= (h_x - 1) \times b + e_x \\ l_y &= (h_y - 1) \times b + e_y \\ l_z &= (h_z - 1) \times b + e_z \end{aligned}$$

This shape can fit into a cubic bounding box of size $l_{max} \times l_{max} \times l_{max}$, where $l_{max} = max(l_x, l_y, l_z)$. Furthermore, the length of the critical path l_c of the target shape is guaranteed to have a length equal in the worst case to that of the bounding cube, which is $O(h_{max})$, where $h_{max} = max(h_x, h_y, h_z)$. We can hence conclude that the reconfiguration time of our method is also $O(l_{max})$ for any shape currently supported by our algorithm.

Furthermore, as the number of modules in this shape also cannot be greater than in the worst case that of the $l_{max} \times l_{max} \times l_{max}$ bounding cube, which is in $O(h_{max}^3)$ as shown in the previous section, the reconfiguration time relative to the number of modules is still $O(N^{\frac{1}{3}})$. □

5.4. Experiments

In this section, we provide various indicators to evaluate the performance of our algorithms, obtained from simulations on the *VisibleSim* simulator. First, we study a set of canonical shapes. Second, we provide a study on a larger and composite use case.

5.4.1. Comparison between canonical shapes

In the following pages, we compare the reconfiguration times with global and relative indicators[4] . We compare canonical shapes—i.e., pyramid, cube, cylinder, and half-sphere—with sizes ranging from $d = 3$ to $d = 9$ tiles wide. Figure 10 shows the number of modules required to build each shape for varying sizes expressed as a number of tiles. Note that due to the FCC lattice and staggered vertical modules layers, the height of d tiles placed vertically is different from the length of d tiles horizontally in the real-world coordinate system. We name this height D, where $D = \frac{\sqrt{2}}{2}d$. When increasing d, we also increase the height and the depth of the shape accordingly, so that, for instance, a cube of height d will actually be a $d \times d \times D$ cube. The CSG description of these objects is, therefore:

[4] Visual comparison with $d = 6$: youtu.be/DjLwsrzA0MI?t=89.

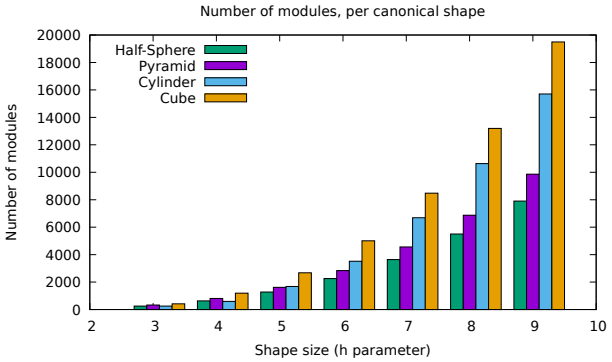

Figure 10: Number of modules in canonical shapes, with varying sizes.

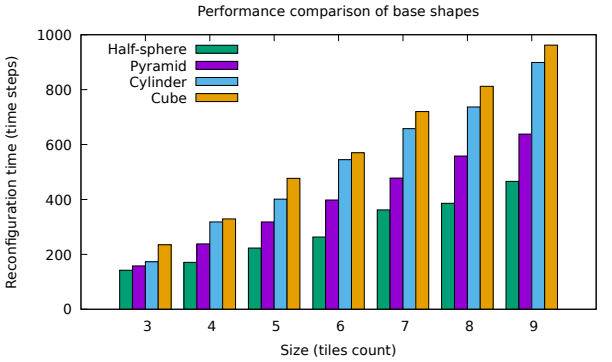

Figure 11: Global reconfiguration time, with varying sizes.

- `translate([`$\frac{d\times b}{2}$`,`$\frac{d\times b}{2}$`,`$\frac{D\times b}{2}$`])`
  `cube([`$d\times b$`,`$d\times b$`,`$D\times b$`], center=true);`
- `translate([`$\frac{D\times b}{2}$`,`$\frac{D\times b}{2}$`,0]) sphere(radius=`$\frac{D\times b}{2}$`);`
- `translate([`$\frac{D\times b}{2}$`,`$\frac{D\times b}{2}$`,0]) cylinder(height=`$D\times b$`,`
  `radius=`$\frac{D\times b}{2}$`, center=false);`

The first comparison is presented in Figure 11 and shows the total time to reconfigure the modules into various dimensions of the shapes under study. The conclusions we can draw from this figure are the following:

- In terms of raw performances, i.e., the time required to complete the reconfiguration, the half-sphere performs faster than the pyramid, which in turn performs faster than the cylinder, which performs better than the cube.
- The time increase is linear with the d parameter.

Figure 12 shows more clearly that the reconfiguration time is stable according to the size of the target shape.

Figure 13 shows how many modules are converging by time step on average—the higher, the better. From this figure, we draw two conclusions:

- As the size of the shape increases, its performance in terms of module placement also increases. It is easily explained by the ability of the algorithm to move more modules in parallel with a wider base.

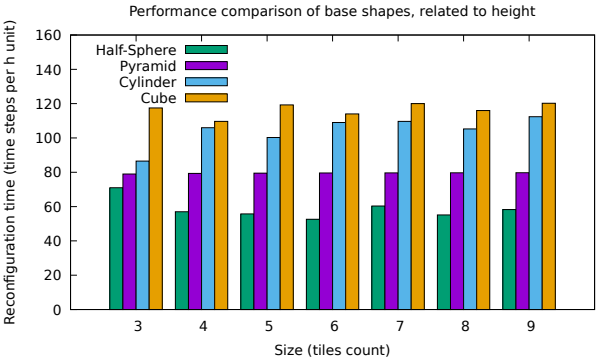

Figure 12: Reconfiguration speed in time steps per height level.

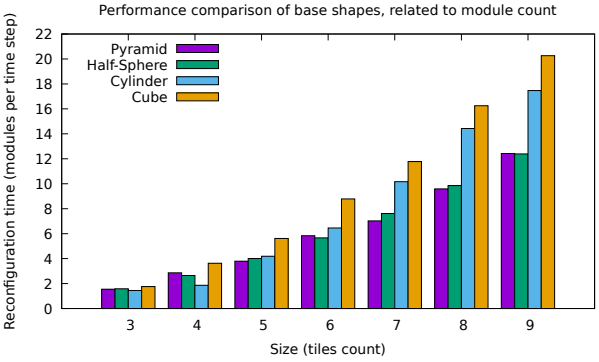

Figure 13: Reconfiguration speed in modules per time step.

- The ranking of the shapes reverses with the cube being first and the pyramid being last. It is partly bound to the fact that, for a given size, the cube contains many more modules than the pyramid. Even though the full cube reconfiguration takes longer, its per-module performance is better. The second part of this behavior is the parallel nature of the cube when compared to the pyramid: both start with a $d \times d$ base, but as the tiles are stacked, the pyramid size decreases, while the cube continues to build $d \times d$ layers, therefore it remains strongly parallel. The same goes when comparing the cylinder to the half-sphere.

Figure 14 shows the convergence rate of our 4 canonical

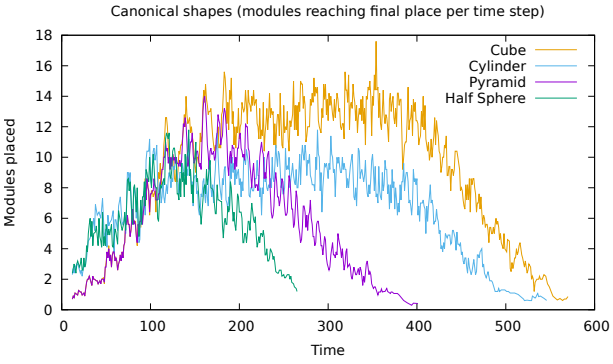

Figure 14: Instant modules placement for canonical shapes with $d = 6$.

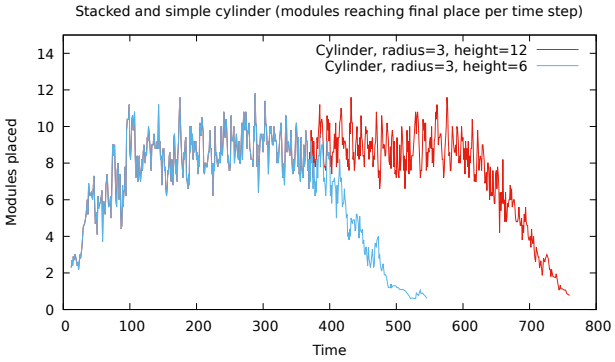

Figure 15: Instant modules placement: comparing simple and stacked cylinder. Both curves overlap until t = 380 time steps.

shapes as a number of modules in place given current simulation time. In this figure, we use as parameter $d = 6$. Several observations are interesting:

- The point where each curve stops marks the end of the simulation, i.e. when the shape is completely built. It shows the differences in terms of modules required to build a given shape as well as the corresponding reconfiguration time.

- The trend of the curve reflects the amount of parallelism in the reconfiguration. The higher the trend, the more parallel. Without surprise, it shows that the cube is the most parallel shape, followed by the cylinder, then the pyramid and last the half-sphere.

- We also observe a 3-step progression for all shapes: first, a steady increase of the parallelism, then a peak or a plateau, followed by a steady decrease until the end of the reconfiguration. The second step is a peak for the half-sphere and the pyramid, while it is a plateau for the cylinder and the cube, confirming the previous observation.

Figure 15 compares the parallelism between a cylinder of size $d = 6$ and $h = \frac{\sqrt{2}}{2}d$, and a cylinder twice as high (dimensions: $d = 6$, $h = \sqrt{2}d$). We clearly see a longer plateau for the highest cylinder, whose stable section lasts longer than the shorter one.

In the previous experiments, we studied various metrics to quantify the performance of our reconfiguration method on canonical shapes. We showed that our algorithms scale well: although the full reconfiguration time is asymptotic to a linear equation relative to d, it must be considered that when d increases, it actually increases the volume of the shapes by an order of d^3. These results support the theoretical analysis performed in Section 5.3. We also show that some shapes (i.e., cubes and cylinders) are inherently more parallel than others (i.e., pyramid and half-sphere) since they keep the same buildable section almost from start to end. Nonetheless, if we were to express parallelism as a function of the number of modules in the target shape, it would appear that all semi-convex shapes have an equal *useful parallelism*, as our method consistently

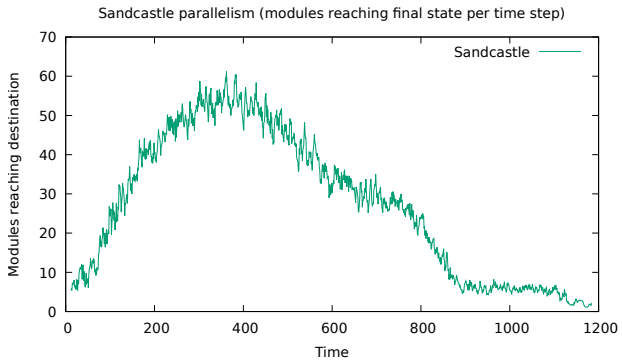

Figure 16: Sandcastle reconfiguration speed, modules in place per time step.

provides the optimal throughput to all the tiles of the shape since module flows are never divided from the sandbox to their destination.

5.4.2. *Complex and composite shape*

In this section, we study a complex shape composed of canonical shapes to build an actual object. The case study is a sandcastle, whose complete shape contains around 32,000 modules.

Figure 16 shows the reconfiguration speed as the instantaneous number of modules placed at each time step. We observe the same global behavior than with canonical shapes, i.e. a slow start when the building starts and is limited by the current section size. Then, the reconfiguration speeds up to a peak before slowly decreasing. We also see that, although there are far more modules than in a cube (5,000 modules), the overall time is limited: only 1,200 time steps for the sandcastle against nearly 600 time steps for the cube. That's a 3× speedup per module on average. We explain this by two factors: first, as we showed in the study of the canonical shapes, our algorithm is very scalable in terms of modules placed. Second, this shape can be viewed as two disconnected parts: the central tower on the one hand, and the cornering towers and their attached walls on the other hand. These shapes are independent and do not need to synchronize.

6. Generalization

The main limitation of this reconfiguration method in its current state remains its narrow scope in terms of the shapes it can build, as the semi-convex class of shapes—even though it can produce complex shapes as shown in Section 5.4.2—is still too restricted for most objects that a user may want to represent. Nonetheless, it is still worth mentioning that there may not be a single best solution for all self-reconfiguration cases, as the preferable solution to general self-reconfiguration might consist of a set of highly specialized algorithms.

While this is still ongoing work at the time of writing, we detail in this section how the previous method can be fully generalized to any shape, by dropping the constraint on the absence of concavities (both within the shape and between the sandbox and the shape) from the last section.

6.1. *Motivations and Challenges*

The full generalization again raises a number of problems, which have been briefly mentioned previously, and that are further discussed below:

1. Tiles can now have no ingoing upward branches. As these were previously the only way of feeding modules into the tile, new solutions must be found for that purpose.

2. While the restricted generalization from Section 5 introduced reverse growth for horizontal branches and tiles, new cases now emerge that will require vertical reverse growth—i.e., growing previously upward branches from the top down (which thus makes them now *downward* outgoing branches), and feeding modules to the tiles below.

3. The previously studied class of shapes did not allow intermediate configurations that could threaten the mechanical stability of the system (e.g., a line or mass of modules hanging in the air), but this could now happen. An ideal planning method would take mechanical constraints into account.

4. Current algorithmic complexities are unlikely to be maintained for all shapes, as the current class of shapes guarantees the maximum transfer rate across all branches of the scaffold due to the one-to-one match between upward ingoing and outgoing branches, as well as exclusive vertical feeding. New reconfiguration cases might now involve splitting the flow of modules from one branch into several ones, each time dividing the module transfer rate.

6.2. *Updated Assumptions*

Our proposed solutions to the challenges of generalization are briefly introduced in this section.

Firstly, in order to address the feeding of tile with no ingoing upward branches, we propose to use the ingoing horizontal branches to feed the modules. For this purpose, we introduce 8 new entry points to the existing 4 vertical EPLs, whose exact location is shown in Figure 8c.

Furthermore, and in the same manner as done previously for the *OppX* and *OppY* branches, we introduce 4 new outgoing branches to the set of **outgoing** branches of the tile: *OppZ*, *OppRevZ*, *OppLZ*, and *OppRZ*, following axes $(0, 0, -1)$, $(1, 1, -1)$, $(1, -1, -1)$, and $(-1, 1, -1)$, respectively. Again, these are practically the same as the **ingoing** upward branches, but they belong to and are grown from the tile above instead of the tile below them. Also, note that incomplete versions of these branches already appeared in semi-convex cases, though they were not grown, for the sake of simplicity and at the cost of a lesser amount of details in the shape.

Besides, regarding the mechanical aspect of self-reconfiguration, we assume for now that all intermediate configurations are stable, as efficiently ensuring the mechanical stability of reconfiguration is an ongoing intractable problem (Hołobut and Lengiewicz, 2017).

6.3. Main Idea

Most of the self-reconfiguration process would remain unchanged, except for portions of the goal shape whose growth was not previously supported. Indeed, it would now be needed to add rules to detect tiles that could not be constructed previously: if a tile has no ingoing upward branch, then it will need to be constructed from either the top tiles or through the lateral tiles. If a tile has a lateral neighbor that is opposite to the growth direction of this portion of the shape then it will be constructed from this tile, or from the tile above otherwise.

If a tile detects that it has to feed the growth of a lateral neighbor through a horizontal branch, it would then send modules from one of its vertical EPLs to a target horizontal EPL. This means that again the set of local rules needs to be greatly expanded to cover all possible cases, which shows the current limits of this local motion method and points at the necessity to find a better alternative, so as to avoid the tedious design work and overloading the memory of modules. In addition, a new tile construction scheduling would need to be carefully designed for these new cases.

Once a tile receives a module through one of its horizontal ingoing branches, it will direct this module to one of its vertical EPLs. We decided to proceed that way so as to reuse our previous tile construction method. It might nonetheless be required to design a novel coordination strategy in order to avoid collisions between modules moving from horizontal EPLs to the vertical ones below.

Again, this process and the earlier ones are to be repeated until the shape is complete.

7. Discussion

Through the various formal analyses from Sections 4.4 and 5.3 and simulation results presented in Sections 4.5 and 5.4, we aimed to give an account of capabilities and significance of our algorithm, which are further discussed here.

With the sandcastle, a complex shape consisting of nearly 40,000 modules, we have shown that our method was able to correctly converge into complex objects even given a massive robotic ensemble. This ability to converge is a crucial aspect of a self-reconfiguration algorithm, and even if we are unable to provide a formal proof of convergence for the algorithm due to its complexity, it can be known exactly for which classes of shapes the method will converge (*semi-convex* cases), and for which it will fail. Furthermore, we have hinted at what could be done to transcend this limitation in Section 6.

It appears that the total duration of the reconfiguration strongly correlates with the height of the target shape. This is indeed very intuitive, as adding height to the shape does not add any new module sources from the sandbox, as enlarging the other dimensions would—in *semi-convex* shapes at least. The width of the object matters also insofar as synchronization is required to respect the *bridging constraint*. In the general case, however, it is not just the width of the object that will matter, but more importantly the size of its base, which connects it to the sandbox. Indeed, the entire module rate of the self-reconfiguration will be determined by the number of tiles that are connected to the sandbox, much like now, but this time this total traffic might have to be split in order to feed different connected sub-parts of the shape. Consequently, the placement of the shape regarding the sandbox is a very important parameter of self-reconfiguration, and will be even more in the general case, as this determines its maximum throughput.

Furthermore, we have highlighted that the other driving factor of the reconfiguration time of our method is related to synchronization points (in the form of waiting times for the construction of parents in new tiles to be grown), their amount, and specific environment. The impact of synchronization on the self-assembly of *3D Catoms* systems has been further studied in (Tucci et al., 2018).

Finally, while this method breaks away from previous work in self-reconfiguration and swarm self-assembly by modular robots due to the presence of a sandbox environment and the geometrical complexity of the model, which makes comparison difficult, a number of pertinent observations can be made. It has been mentioned in Section 2 that this precise module geometry could be reconfigured in linear time from a flat disk of modules into various other shapes (Yim et al., 2001), but at the cost of a lack of a guarantee of convergence. Furthermore, the fastest results in self-reconfiguration using scaffolding could also achieve linear time reconfiguration with simpler module geometries (Støy and Nagpal, 2007; Lengiewicz and Holobut, 2019), and leveraging translation motions through narrow tunnels to achieve such speeds. However, self-reconfiguring from a prebuilt shape into another rather than from a sandbox-like reserve raises the additional problem of resource allocation—i.e., where to pick modules that will be used in a particular area of the goal shape from—and adds complexity to the task. Therefore, while our current result cannot be directly compared to these other solutions, we considered that reaching a sub-linear cubic-square reconfiguration time with such a level of parallelism is already an admirable achievement, and we are confident that extending this method to shape-to-shape reconfiguration will yield results that can rival with those and open new frontiers in term of scalability for self-reconfiguring modular robotic systems.

8. Conclusion

We have introduced a novel way of representing objects made of a micro-modular-robot swarm arranged in an FCC-lattice structure, by discretizing the object into a set of regular and porous tiles that can be deterministically constructed and leave holes in the structure for motion. Moreover, we have proposed a framework for constructing these scaffolded shapes from an underneath reserve of modules, in sub-linear time and with high parallelism, using a deterministic construction scheduling, local-rules-based motion planning, and collision avoidance through distributed messaging. This sort of self-reconfiguration task by a massive swarm of autonomous and independent agents would traditionally have required advanced

optimization methods. Still, this work shows that deterministic, rule-based methods can be equally suited for this task.

The performance of this self-reconfiguration method has been evaluated through analyses and simulations for a number of case studies with increasing complexity, showing that an $O(N^{1/3})$ reconfiguration time could be achieved for a large class of shapes that do not have concavities—with N the number of modules in the system. Expanding this method to all remaining shapes is an ongoing work, but the associated challenges have been introduced, and a number of solutions have been proposed in this paper nonetheless.

Overall, this work stands as proof that large-scale reconfiguration can be performed in a reasonable time (relative to the number of modules, hardware capabilities will define it in absolute terms) using adequate methods and supporting systems such as our sandbox. It also shows that scaffolds with complex geometries can be considered, at least in theory, and confirms once again that it is a very powerful tool to facilitate the self-reconfiguration of massive modular robots.

9. Future Work

There are a number of areas for improvement in the current version of this work that have been identified in this paper, as well as perspectives for extending it; they are reminded and commented below:

First, the current framework needs to be extended to all classes of shapes to truly be on par with other general reconfiguration methods. This raises a number of challenges raised in Section 6. Current complexity results are unlikely to remain as they are now for all shapes due to the decreasing module throughput of the reconfiguration with increasing shape complexity and the number and size of concavities.

Generalization is likely to require improvements on the current local-rule-based local motion planning solution, with the rules required for generalization becoming too numerous, potentially filling the scarce memory of modules, and rendering the hand-design of rules laborious and troublesome. This could either be replaced by an alternative and better-suited motion planning method or benefit from improvements in design and compactness.

Then, reuniting with traditional self-reconfiguration methods, this framework will be extended to shape-to-shape reconfiguration. Resource allocation is likely to proceed both by exchanges between the initial and goal shapes as well as between the sandbox and the goal shape. It would also allow for reconfiguration between shapes with different cardinalities, another previously unstudied aspect of the self-reconfiguration problem.

Regarding the scaffold itself, a rigorous mechanical analysis will have to be performed in order to validate the current tile and scaffold models, along with providing a definite upper bound on the length of branches. Similarly, further research and engineering work is required for designing all the procedures and algorithms necessary for the sandbox to operate.

Finally, the chief issue with scaffolding in its current state is that it deteriorates the external aspect of the object. Thus, we would like to add one or several layers of modules on the external surface of the scaffold during reconfiguration so that it would look no different than a compact object. A coating algorithm is therefore required, that would guide modules to goal locations on the surface of the shape according to a strict construction plan.

10. Acknowledgments

Acknowledgment

This work was partially supported by the ANR (ANR-16-CE33-0022-02), the French Investissements d'Avenir program, the ISITE-BFC project (ANR-15-IDEX-03), and the EIPHI Graduate School (contract ANR-17-EURE-0002).

References

Ahmadzadeh, H., Masehian, E., Asadpour, M., 2016. Modular Robotic Systems: Characteristics and Applications. Journal of Intelligent & Robotic Systems 81, 317–357. URL: https://doi.org/10.1007/s10846-015-0237-8, doi:10.1007/s10846-015-0237-8.

Bäck, T., Fogel, D.B., Michalewicz, Z., 1997. Handbook of evolutionary computation. CRC Press.

Barraquand, J., Latombe, J.C., 1991. Robot Motion Planning: A Distributed Representation Approach. The International Journal of Robotics Research 10, 628–649. URL: https://doi.org/10.1177/027836499101000604, doi:10.1177/027836499101000604.

Beni, G., 2005. From Swarm Intelligence to Swarm Robotics, in: Şahin, E., Spears, W.M. (Eds.), Swarm Robotics, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 1–9.

Bie, D., Wang, Y., Zhang, Y., Liu, C., zhao, J., Zhu, Y., 2018. Parametric L-systems-based modeling self-reconfiguration of modular robots in obstacle environments. International Journal of Advanced Robotic Systems 15, 1729881418754477. URL: https://doi.org/10.1177/1729881418754477, doi:10.1177/1729881418754477.

Butler, Z., Kotay, K., Rus, D., Tomita, K., 2002. Generic decentralized control for a class of self-reconfigurable robots, in: Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on, IEEE. pp. 809–816.

Chung, H., Shin, K.s., 2019. Genetic algorithm-optimized multi-channel convolutional neural network for stock market prediction. Neural Computing and Applications , 1–18.

Dewey, D.J., Ashley-Rollman, M.P., Rosa, M.D., Goldstein, S.C., Mowry, T.C., Srinivasa, S.S., Pillai, P., Campbell, J., 2008. Generalizing metamodules to simplify planning in modular robotic systems, in: Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on, pp. 1338–1345. doi:10.1109/IROS.2008.4651094.

Dorigo, M., Maniezzo, V., Colorni, A., 1996. Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics) 26, 29–41. URL: http://ieeexplore.ieee.org/document/484436/, doi:10.1109/3477.484436.

Fitch, R., Butler, Z., Rus, D., 2003. Reconfiguration planning for heterogeneous self-reconfiguring robots, in: Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on, pp. 2460–2467. doi:10.1109/IROS.2003.1249239.

Fitch, R., McAllister, R., 2013. Hierarchical Planning for Self-reconfiguring Robots Using Module Kinematics, in: Distributed Autonomous Robotic Systems 10, pp. 477–490. doi:10.1007/978-3-642-32723-0_34.

Goldberg, D.E., Holland, J.H., 1988. Genetic Algorithms and Machine Learning. Machine Learning 3, 95–99. URL: https://doi.org/10.1023/A:1022602019183, doi:10.1023/A:1022602019183.

Hołobut, P., Lengiewicz, J., 2017. Distributed computation of forces in modular-robotic ensembles as part of reconfiguration planning, in: Robotics and Automation (ICRA), 2017 IEEE International Conference on, pp. 2103–2109. doi:10.1109/ICRA.2017.7989242.

Kawano, H., 2015. Complete reconfiguration algorithm for sliding cube-shaped modular robots with only sliding motion primitive, in: Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on, pp. 3276–3283. doi:10.1109/IROS.2015.7353832.

Kennedy, J., Eberhart, R., 1995. Particle swarm optimization, in: Proceedings of ICNN'95 - International Conference on Neural Networks, IEEE, Perth, WA, Australia. pp. 1942–1948. URL: http://ieeexplore.ieee.org/document/488968/, doi:10.1109/ICNN.1995.488968.

Kotay, K.D., Rus, D.L., 2000. Algorithms for self-reconfiguring molecule motion planning, in: Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on, pp. 2184–2193. doi:10.1109/IROS.2000.895294.

Kurokawa, H., Murata, S., Yoshida, E., Tomita, K., Kokaji, S., 1998. A 3-D Self-Reconfigurable Structure and Experiments, in: Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications, p. 6.

Lengiewicz, J., Holobut, P., 2019. Efficient collective shape shifting and locomotion of massively-modular robotic structures. Auton. Robots 43, 97–122. URL: https://doi.org/10.1007/s10514-018-9709-6, doi:10.1007/s10514-018-9709-6.

Naz, A., Piranda, B., Bourgeois, J., Goldstein, S.C., 2016. A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots, in: Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on, IEEE. pp. 254–263. URL: http://ieeexplore.ieee.org/abstract/document/7778628/.

Park, M., Chitta, S., Teichman, A., Yim, M., 2008. Automatic Configuration Recognition Methods in Modular Robots. The International Journal of Robotics Research 27, 403–421. URL: http://journals.sagepub.com/doi/10.1177/0278364907089350, doi:10.1177/0278364907089350.

Piranda, B., 2016. VisibleSim: Your simulator for Programmable Matter, in: Römer, K., Scheideler, C., Fekete, S.P., Richa, A.W. (Eds.), Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. volume 6 of *Dagstuhl Reports*, p. 12. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6759. doi:10.4230/DagRep.6.7.1.

Piranda, B., Bourgeois, J., 2018. Designing a quasi-spherical module for a huge modular robot to create programmable matter. Autonomous Robots 42, 1619–1633. URL: https://doi.org/10.1007/s10514-018-9710-0, doi:10.1007/s10514-018-9710-0.

Piranda, B., Laurent, G.J., Bourgeois, J., Clévy, C., Möbes, S., Fort-Piat, N.L., 2013. A new concept of planar self-reconfigurable modular robot for conveying microparts. Mechatronics 23, 906–915. URL: http://linkinghub.elsevier.com/retrieve/pii/S0957415813001633, doi:10.1016/j.mechatronics.2013.08.009.

Rajasekhar, A., Lynn, N., Das, S., Suganthan, P.N., 2017. Computing with the collective intelligence of honey bees – A survey. Swarm and Evolutionary Computation 32, 25 – 48. URL: http://www.sciencedirect.com/science/article/pii/S221065021630027X, doi:https://doi.org/10.1016/j.swevo.2016.06.001.

Rubenstein, M., Ahler, C., Nagpal, R., 2012. Kilobot: A low cost scalable robot system for collective behaviors, in: 2012 IEEE International Conference on Robotics and Automation, IEEE, St Paul, MN, USA. pp. 3293–3298. URL: http://ieeexplore.ieee.org/document/6224638/, doi:10.1109/ICRA.2012.6224638.

Ser, J.D., Osaba, E., Molina, D., Yang, X.S., Salcedo-Sanz, S., Camacho, D., Das, S., Suganthan, P.N., Coello, C.A.C., Herrera, F., 2019. Bio-inspired computation: Where we stand and what's next. Swarm and Evolutionary Computation 48, 220 – 250. URL: http://www.sciencedirect.com/science/article/pii/S2210650218310277, doi:https://doi.org/10.1016/j.swevo.2019.04.008.

Strumberger, I., Tuba, E., Bacanin, N., Jovanovic, R., Tuba, M., 2019. Convolutional Neural Network Architecture Design by the Tree Growth Algorithm Framework, in: 2019 International Joint Conference on Neural Networks (IJCNN), IEEE, Budapest, Hungary. pp. 1–8. URL: https://ieeexplore.ieee.org/document/8851755/, doi:10.1109/IJCNN.2019.8851755.

Støy, K., 2006. Using cellular automata and gradients to control self-reconfiguration. Robotics and Autonomous Systems 54, 135 – 141. URL: http://www.sciencedirect.com/science/article/pii/S0921889005001521, doi:https://doi.org/10.1016/j.robot.2005.09.017.

Støy, K., Nagpal, R., 2007. Self-Reconfiguration Using Directed Growth, in: Distributed Autonomous Robotic Systems 6, pp. 3–12. URL: https://doi.org/10.1007/978-4-431-35873-2_1, doi:10.1007/978-4-431-35873-2_1.

Sung, C., Bern, J., Romanishin, J., Rus, D., 2015. Reconfiguration planning for pivoting cube modular robots, in: 2015 IEEE International Conference on Robotics and Automation (ICRA), IEEE. pp. 1933–1940. URL: http://ieeexplore.ieee.org/document/7139451/, doi:10.1109/ICRA.2015.7139451.

Thalamy, P., Piranda, B., Bourgeois, J., 2019a. Distributed Self-Reconfiguration using a Deterministic Autonomous Scaffolding Structure, in: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, Montreal QC, Canada. pp. 140–148. doi:10.5555/3306127.3331685.

Thalamy, P., Piranda, B., Bourgeois, J., 2019b. A survey of autonomous self-reconfiguration methods for robot-based programmable matter. Robotics and Autonomous Systems 120, 103242. URL: https://linkinghub.elsevier.com/retrieve/pii/S0921889019301459, doi:10.1016/j.robot.2019.07.012.

Thalamy, P., Piranda, B., Lassabe, F., Bourgeois, J., 2019c. Scaffold-Based Asynchronous Distributed Self-Reconfiguration By Continuous Module Flow, in: 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4840–4846. doi:10.1109/IROS40897.2019.8967775.

Tuba, M., Bacanin, N., Beko, M., 2015. Multiobjective RFID Network Planning by Artificial Bee Colony Algorithm with Genetic Operators, in: Tan, Y., Shi, Y., Buarque, F., Gelbukh, A., Das, S., Engelbrecht, A. (Eds.), Advances in Swarm and Computational Intelligence, Springer International Publishing, Cham. pp. 247–254.

Tucci, T., Piranda, B., Bourgeois, J., 2017. Efficient Scene Encoding for Programmable Matter Self-reconfiguration Algorithms, in: Proceedings of the Symposium on Applied Computing, pp. 256–261. URL: http://doi.acm.org/10.1145/3019612.3019706, doi:10.1145/3019612.3019706.

Tucci, T., Piranda, B., Bourgeois, J., 2018. A Distributed Self-Assembly Planning Algorithm for Modular Robots, in: International Conference on Autonomous Agents and Multiagent Systems) (AAMAS), Association for Computing Machinery (ACM), Stockholm, Sweden. pp. 550–558.

Yim, M., Zhang, Y., Lamping, J., Mao, E., 2001. Distributed Control for 3D Metamorphosis. Autonomous Robots 10, 41–56. URL: https://doi.org/10.1023/A:1026544419097, doi:10.1023/A:1026544419097.

Yoshida, E., Murata, S., Kurokawa, H., Tomita, K., Kokaji, S., 1998. A distributed method for reconfiguration of a three-dimensional homogeneous structure. Advanced Robotics 13. doi:10.1163/156855399X00234.

Zhu, L., El Baz, D., 2019. A programmable actuator for combined motion and connection and its application to modular robot. Mechatronics 58, 9–19. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957415819300029, doi:10.1016/j.mechatronics.2019.01.002.

Zhu, Y., Bie, D., Wang, X., Zhang, Y., Jin, H., Zhao, J., 2017. A distributed and parallel control mechanism for self-reconfiguration of modular robots using L-systems and cellular automata. Journal of Parallel and Distributed Computing 102, 80 – 90. URL: http://www.sciencedirect.com/science/article/pii/S0743731516301824, doi:https://doi.org/10.1016/j.jpdc.2016.11.016.

Ünsal, C., kilivççöte, H., Patton, M.E., Khosla, P.K., 2000. Motion Planning for a Modular Self-Reconfiguring Robotic System, in: Parker, L.E., Bekey, G., Barhen, J. (Eds.), Distributed Autonomous Robotic Systems 4, Springer Japan, Tokyo. pp. 165–175. URL: https://doi.org/10.1007/978-4-431-67919-6_16, doi:10.1007/978-4-431-67919-6_16.