Scaffold-based Asynchronous Distributed Self-reconfiguration by Continuous Module Flow

Pierre Thalamy, Benoit Piranda, Frédéric Lassabe, Julien Bourgeois

SCAFFOLD-BASED ASYNCHRONOUS DISTRIBUTED SELF-RECONFIGURATION BY CONTINUOUS MODULE FLOW

Pierre Thalamy[†], Benoît Piranda[†], Frédéric Lassabe[†] and Julien Bourgeois[†]

***Abstract*— Distributed self-reconfiguration in large-scale modular robots is a slow process and increasing its speed a major challenge. In this article, we propose an improved and asynchronous version of a previously proposed distributed self-reconfiguration algorithm to build a parametric scaffolding structure. This scaffold can then be coated to form the desired final object. The scaffolding is built through a continuous feeding of modules into the growing shape from an underneath reserve of modules which shows a reconfiguration time improved by a factor of $\sqrt[3]{N}$ compared to the previous and synchronous version of the algorithm, therefore attaining an $O(N^{1/3})$ reconfiguration time, with N the number of modules in the system. Our algorithm uses a local motion coordination algorithm and pipelining techniques to ensure that modules can traverse the structure without collisions or creating deadlocks. Last but not least, our algorithm manages uncertainty in the motion duration of modules without negatively impacting reconfiguration time.**

I. INTRODUCTION

The Programmable Matter Project[1] [1] aims to build objects using modular self-reconfigurable robots [3] composed of micro-robots thus able to change their shape, a process called self-reconfiguration. The Programmable Matter Project is a follow up of the Claytonics project [4] where each micro-robot is called a Claytronics atom (*Catom*). An object made of Programmable Matter can be densely filled with *Catoms* but this creates two major problems: movements can only occur on the surface, which limits the number of parallel movements, and it immobilizes the *Catoms* making up the interior of the volume, which could be put to better use. Reconfiguration time is a crucial parameter for self-reconfiguration. In [12], using a thousand *Kilobots*, a reconfiguration takes from 6 to 12 hours, which limits the usefulness of self-reconfiguration. To improve on this, we propose to build objects using a sparse scaffold structure for their inner part, to which we add a coating for their outer part, thus creating the illusion of a densely filled object at a lesser cost and dramatically speeding up the reconfiguration.

In [15], we defined the scaffolding structure, and proposed a synchronous self-reconfiguration method based on two levels of planning: (i) an ordering in the construction of the scaffold; (ii) local rules for guiding the local motion of modules. It also assumes an underneath reserve of modules, named *sandbox*, from which modules can be called in to take part in the reconfiguration. We showed that this method is able to build square pyramids with n $O(N^{\frac{2}{3}})$ reconfiguration time and using $O(N^{\frac{4}{3}})$ messages without congesting the network, with N the total number of *Catoms*.

However, this work presents some limitations: it relies heavily on a synchronization between the central node of a scaffold tile and the ones of the four tiles feeding it from the *sandbox*, as well as a fixed movement time for all modules, which is unrealistic. Furthermore, it requests modules from the *sandbox* only when they are needed, which slows down the reconfiguration process due to the delay between their request and arrival.

To address these limitations, we propose in this paper an asynchronous distributed algorithm which improves on the algorithm in [15] by continuously injecting modules on the scaffold from the *sandbox*, regardless of the requirements of the goal shape. It therefore uses a surplus of modules which can later be used for the coating and it improves the reconfiguration time of the self-reconfiguration by a factor of $\sqrt[3]{N}$, which corresponds to an $O(N^{1/3})$ reconfiguration time when building the scaffold of a square pyramid, with N the total number of *Catoms*. Furthermore, our algorithm does not assume any synchronization of the system to achieve these results, and is resistant to uncertainty in the rotation time of the modules. Both are made possible thanks to our motion coordination algorithm.

II. RELATED WORKS

Using scaffolding to aid self-reconfiguration is not a novel approach. To the best of our knowledge, it was introduced in [6] with massive scaffold tiles consisting of 54 modules. It was then further investigated in works with cubic modules [13, 14] and [7] with a much simpler scaffold design in which modules could slide through one-module-thick tunnels. Scaffolding is very much alike some of the applications of the concept of meta-modules [9, 2, 17, 5], where modules are grouped into logical units, which can be organized so as to form a porous structure that allows internal module movement through *tunneling*.

Reconfiguration can also be aided through communication. For instance, in [8], the authors propose a reconfiguration algorithm for 2D cylindrical robotic modules that uses message-based coordination to ensure that modules moving in parallel are not impeding on their respective motions. In practice, this method creates *trains* of rotating modules that move and stop in unison, always keeping one free lattice cell between them. Yet, the generalization of this method to 3D reconfiguration is non-trivial, as the added dimension prevents the identification of a common direction of motion for modules moving in parallel. Nevertheless, with

[†]All authors are with Univ. Bourgogne Franche-Comté, FEMTO-ST Institute, CNRS, 1 cours Leprince-Ringuet, 25200, Montbéliard, France. {first}.{last}@femto-st.fr

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

the addition of a scaffolding structure such as ours, the navigation on a single branch of the scaffold can be again reduced to a 2D problem through pipelining, thus restoring its benefits.

III. PREREQUISITES

A. THE 3D CATOM MODULAR ROBOT

We consider a modular robot made of a very large number of *3D Catom* [11] robotic modules, arranged in a *Face-Centered Cubic* (*FCC*) lattice. *3D Catoms* are micro-scale quasi spherical modules that are under development. Each *3D Catom* can connect to up to 12 neighbor modules thanks to the connectors positioned on each of its sides. They communicate locally to their immediate neighbors through these same connectors, also supplying power.

3D Catoms move by rotating on the surface of a neighbor module using electrostatic actuators, acting as a *pivot* and providing the necessary actuation. They can use either one of the two faces (*hexagonal* and *octagonal*) to rotate on their surface (see video in footnote[2]), depending on their local motion constraints.

We assume that our modules are able to react through their programming to events related to their state or local environment, such as the arrival or departure of a neighbor (*ADD_NEIGHBOR* / *REMOVE_NEIGHBOR*), or the end of a rotation (*ROTATION_END*).

Due to the geometry of *3D Catoms*, a position can only be filled if the two opposite positions around it are free. Filling a gap in a line of modules is therefore impossible [16]. This is why it is crucial to enforce a methodical construction strategy in order to avoid the occurrence of deadlocks.

Furthermore, additional motion constraints prevent a *3D Catom* from performing a motion if that motion will result in a collision with another module in its first- or second-order neighborhood.

B. SCAFFOLD MODEL

We define our scaffold model as an arrangement of structured groups of modules named *tiles*. A scaffold *tile* consists of a single module at the core, referred to as *tile root*, and acting as the *Coordinator* of the construction of the tile; one or two horizontal branches along the $\overrightarrow{x}$ and $\overrightarrow{y}$ axes; from one to four vertical branches named *ZBranch*, *LZBranch*, *RevZBranch*, *RZBranch*; and four *Support* modules between the vertical branches, allowing modules coming from incident vertical branches to traverse the tile vertically. Each branch is made of b modules including the *tile root*, such that $4 < b < m$, with b a parameter of our model and m a constant defined by the mechanical strength of *3D Catom* connectors.

Modules navigate the scaffolding structure by **vertically** flowing from tile to tile, rotating on *Support* modules or the modules making up the branches of the tiles, acting as *pivots*. Each module or position comprising the structure of a tile can be referred to as a *Tile Component*. In addition to components, we define special positions around the tiles to

[2] *3D Catoms* motion video: `https://youtu.be/IZh-5p1dbKk`

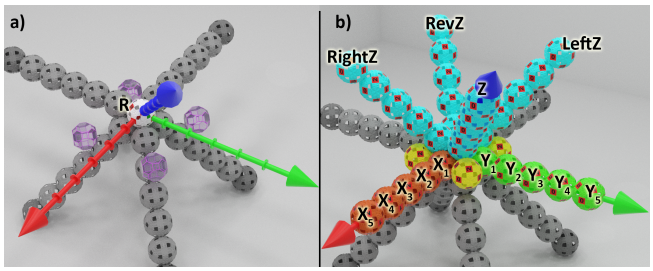

Fig. 1. 3D structure of a tile with $b = 6$. a) Pink transparent cells represent the four entry points into the tile that we use, and white module represents the root R of the new tile, origin of the local coordinates system; b) 3D position of support (yellow) and branch modules of the tile.

which we will refer as *Entry Point Positions*, or *EPL*. These are the lattice cells through which modules entering a tile must pass. In this work we use the 4 *EPL* located centrally on each of the incoming vertical branches, over the second to last module of each incident branches (see Figure 1.a)

When a module enters a tile through one of its *EPL*, it has to request a *goal* position within that tile from the local *Coordinator* of that tile, which will either direct it to one of the children tile above, or a destination tile component to be claimed, if the tile is still under construction.

This work focuses on the construction of structures where all tiles of the scaffold are resting on four incident vertical branches, and therefore 4 parent tiles underneath, namely, h-pyramids. An h-pyramid is a square pyramid shape with an h-tile large square base and a height of h tiles. This simplification is useful as modules can flow through the structure vertically without having to coordinate the horizontal traversal of tiles by modules flowing from different incoming vertical tiles. This restriction will be lifted with further coordination works, and we are confident this can be done without affecting on the present results.

C. SANDBOX

As mentioned earlier, we assume that underneath the reconfiguration scene is located a reserve of modules which we will refer to as *sandbox*. The sandbox is a cubic volume organized into our scaffolding structure and holding a surplus of modules on its branches, and provides both power and the description of the goal shape to all modules. It allows to introduce *3D Catoms* at various ground locations of the reconfiguration space, by having them rotating from the sandbox using the scaffold branches just as in our reconfiguration algorithm.

IV. SELF-RECONFIGURATION ALGORITHM

A. ROLES

Throughout this paper, we consider the following module roles for explaining the algorithm: *Coordinator* is the root module of a tile, acting as a local coordinator guiding the construction process of its tile; *FreeAgent* modules are introduced from the sandbox underneath and navigate the structure until they encounter a tile with a scaffold position

to be claimed; *Beam* modules are any final scaffold components that are not a *Coordinator* (thus, branches or support modules).

B. MESSAGING

Our reconfiguration algorithm uses a number of different message types. Some of them are used by the high level reconfiguration process to direct the growth of the structure. Others have to do exclusively with the coordination of modules navigating the structure in parallel. Below is a summary of the messages used in our algorithm, their function, and their data.

1) Scaffold construction messages:

MESSAGE_NAME (**ACRONYM**) [**DATA**]

- *REQUEST_GOAL_POSITION* (**RGP**) [sender]: Sent to the local *Coordinator* by a *FreeAgent* arriving into a new tile, to request its next destination.
- *PROVIDE_GOAL_POSITION* (**PGP**) [recipient, goal]: Response to *RGP* sent by a *Coordinator* to a requesting *FreeAgent*. Includes a goal position that can either be an unfilled component of the current tile or the EPL of a tile above.
- *TILE_INSERTION_READY* (**TIR**) [$\emptyset$]: Signals to the modules waiting on the EPL of a future tile that construction can proceed. Sent by the last arrived module from the last arrived incident branch to that tile.
- *COORDINATOR_READY* (**CR**) [$\emptyset$]: Sent by a newly arrived coordinator to instruct *FreeAgent* modules waiting on an *EPL* that it is ready to process *RGP* messages.

For additional details on *RGP*, *PGP*, and *TIR*, see [15].

2) Motion coordination messages:

- *PROBE_LIGHT_STATE* (**PLS**) [sender, nextPos]: Sent by a module about to move in order to obtain an authorization of moving from the pivot module to which it will latch after its motion. It holds the position of the sender for routing the reply, and the position to which it seeks to move (*nextPos*). The latter is used by a receiver to dynamically determine the pivot that should answer.
- *GREEN_LIGHT_ON* (**GLO**) [recipient]: This is a reply to *PLS*, sent back to the module which made the request by the pivot module that is ready to receive it as a neighbor.
- *FINAL_TARGET_REACHED* (**FTR**) [$\emptyset$]:
  Pivot modules monitor changes in their neighborhood caused by moving modules in order to update their state and block or resume the flow of modules moving to their location. The flow of modules is usually resumed by a pivot when a moving module leaves the neighborhood of the pivot. However, if a moving module claims a scaffold position that is next to the pivot, a moving module will enter its neighborhood but never leave it. In this scenario, *FTR* is sent by the module that has just arrived to its claimed scaffold position module to instruct the pivot to resume the flow of modules even though it is still in its neighborhood.

We use a distributed message passing paradigm to deliver messages from one module to another non-neighboring module. Intermediate modules (between the sender and the recipient) leverage the geometrical regularity of the structure and their local knowledge to deduce the next hop according to the message type.

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

```
Msg Handler REQUEST_GOAL_POSITION(srcPos):
    epl = getEPLForPosition(srcPos);
    if plan.isOver() then
        goalPos = getAlternateBranchEPLfor(epl);
    else if plan.nextComponentIsSourcedBy(epl) then
        goalPos = plan.popNextComponent();
    else
        moduleWaitingOnEPL[epl] = true; return;
    sendMsg(sender, PGP(srcPos, goalPos));
    checkModulesWaitingOnEntryPoints();

Function checkModulesWaitingOnEPL:
    do
        moduleAwoken = false;
        foreach epl ∈ getAllEntryPoints() do
            if plan.nextComponentIsSourcedBy(epl) then
                goalPos = plan.popNextComponent();
                sendMsg(sender, PGP(srcPos, goalPos));
                moduleAwoken = true;
    while moduleAwoken = true;
```

C. OVERVIEW

Our reconfiguration method operates at two levels of planning.

1) High-Level Planning: The higher level of planning consists in the construction of the scaffold at the tile level, in other words, the construction of the tiles relative to each other in term of their precedence. A construction order is enforced (roughly 3D diagonal growth) based on a very simple set of rules: *(i)* scaffold construction must start from a single ground point; *(ii)* A new tile can only start undergoing construction once all its incident branches are complete. The tile construction process is then directed by the first module of a tile placed in the root position, whose role is to direct incoming module flows to the next position to be filled in the tile, following a predetermined construction plan based on the specific location of that tile within the goal shape. When their tile is complete, *Coordinators* are responsible for directing all modules arriving to a branch, to the branch directly above that branch.

Algorithm 1 illustrates how this behavior is implemented. See Algorithm 2 for the complementary viewpoint of *FreeAgent* modules.

During the construction of the scaffold, modules are continuously flowing through the structure **vertically**, from the sandbox below, and they only stop when they are blocked by a *Coordinator* after entering an *EPL* of its tile. The motion coordination algorithm explained in the next section allows

the propagation of the halt of the flow through message passing, in addition to preventing potential collisions.

When a *Coordinator* gets into its final position, it initializes a queue data structure holding a list of couples $\langle SC_i, EPL_j \rangle$ where SC_i is a scaffold component, $i \in [0, 5b+4]$, and EPL_j is the *EPL* from which a module claiming the component SC_i should come, $j \in [0,3]$. This list therefore represents the ordered construction plan of the tile, computed dynamically by the *Coordinator* when it arrives into the tile root position and based on its position within the overall structure. This plan enforces a precedence in the placement of modules coming from different directions, in order to minimize the risk of collisions. Upon reception of an *RGP* message, the *Coordinator* checks whether the request comes from the entry point EPL_i of the next component in the plan and responds with a *PGP* message or puts the module on hold otherwise. Whenever the construction advances through the claim of the next component by a *FreeAgent* module, the *Coordinator* checks whether it previously put on hold a module waiting on the *EPL* for the next component to be filled, and so on. Once the construction of the tile is over, the flow of incoming modules is left uninterrupted, and all modules are routed towards the *EPL* of the branch directly over the one on which they arrived.

2) Low Level Planning: Conversely, the low level planning consists in the mechanisms of the flow of modules themselves, or in other words, how *FreeAgent* modules navigate the tile structure once they have been assigned a goal position within a tile. Just like in [15], each *3D Catom* is given an identical set of local rules that describes the sequence of motions to be taken in order to navigate from each *EPL* of a tile to any component locations within that tile, or *EPL* of children tiles. Every time a module has to perform a motion towards a goal position, it attempts to match its local neighborhood, the number of motions since its entry into the tiles, and its goal position to an entry in its local rules database. The result is the displacement induced by its next motion towards its goal. It then searches its neighborhood for a pivot that can be used to perform a rotation representing this movement. If it cannot find a rule to match or a pivot to perform the motion, it waits and periodically checks for these conditions until they are satisfied.

D. MOTION COORDINATION ALGORITHM

We find that managing a large numbers and a high density of modules across the structure is too slow and too complicated as moving modules tend to block each other. We therefore force all modules moving concurrently to always keep a free position between each other, in a manner similar to [8].

A major difference with our previous work is that module motions now are asynchronous and can have different stochastic durations. In order to support both, we propose a distributed and local motion coordination algorithm, described in this section.

A straightforward condition to ensure that two mobile modules have always a space between them is to prevent any module that could potentially act as a pivot from having two mobile modules connected to it at the same time. In the context of the scaffolding, pivot modules are all *Beam* modules, immobile modules constituting the components of the scaffold. These pivot modules can be in two states *GREEN_LIGHT*, and *RED_LIGHT*. *GREEN_LIGHT* means that the pivot is accepting that a new mobile module latches onto it, while *RED_LIGHT* means the opposite. The transition between states occurs every time a mobile module latches onto a pivot or unlatches from it, as it can be seen on Figure 2. However, if the mobile module is on its final motion to its goal position, it will send send a *FTR* to instruct the pivot to turn back green, as explained in Section IV-B.

Every time a module wants to move, it sends a *PLS* message to the pivot it plans to use for the motion. The pivot then evaluates if it should be the one responding to the request. The target pivot is one of the pivots to which the module will connect at the end of its motion. It is the one that is the farthest along the path of the moving module. When a pivot receives a motion request, it either responds directly with a *GLO* message if it is in the *GREEN_LIGHT* state, or waits for its state to change back to *GREEN_LIGHT* before responding, therefore holding the motion of the module until it is safe to proceed. This could be modeled as an *ORANGE_LIGHT* state.

Algorithms 2 and 3 show how these mechanisms are implemented.

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

```
Event ROTATION_END: ARRIVED_FROM_SANDBOX:
    if myPos == goalPos then
        if isTileComponent(myPos) then
            agentRole = agentRoleForComponent(myPos);
        else reachedNewTileEntryPoint() ;
    else
        step++;
        planNextRotation();

Function reachedNewTileEntryPoint():
    coordinatorPos = getNearestTileRootFrom(myPos);
    nextHop = findSupportOrBranchTipNeighbor();
    sendMsg(nextHop, RGP());

Function planNextRotation():
    ngbh = getNeighborhood();
    {nextPos, pivot} = matchRules(ngbh, goalPos, step);
    sendMsg(pivot, PLS(myPos, nextPos));

Msg Handler PROVIDE_GOAL_POSITION(rcvdPos):
    step = 0; goalPos = rcvdPos;
    planNextRotation();

Msg Handler GREEN_LIGHT_ON(rcvdPos):
    rotate(nextPos, pivot);
```

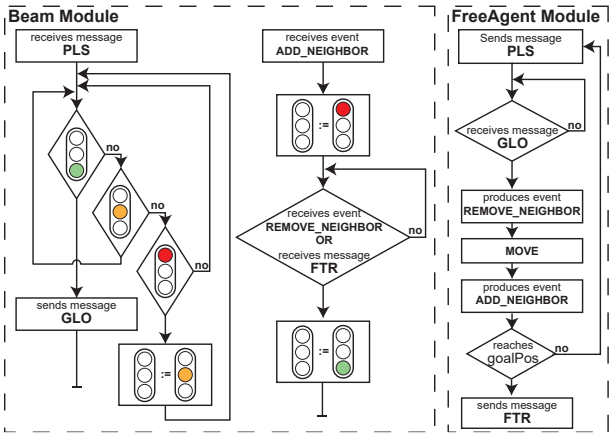

Fig. 2. Pivot light states transition diagram. The two *Beam* routines are executed concurrently on pivot modules.

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

Function *setGreenLightAndResumeFlow()*:
  if *state* == *ORANGE* **then**
    sendMessage(sender, GLO(waitingModulePos));
  state = GREEN;

Event Handler *ADD_NEIGHBOR*: state = RED ;

Event Handler *REMOVE_NEIGHBOR*:
  setGreenLightAndResumeFlow();

Msg Handler *REQUEST_GOAL_POSITION(RGPmsg)*:
  forwardMsgTowardsCoordinator(RGPmsg);

Msg Handler *PROVIDE_GOAL_POSITION(PGPmsg)*:
  forwardMsgTowardsRecipient(*PGPmsg*);

Msg Handler *PROBE_LIGHT_STATE(PLSmsg)*:
  if *state* == *GREEN* **then**
    sendMsg(sender, GLO(PLSmsg.source));
  else
    state = ORANGE ;
    waitingModule = PLSmsg.source;

Msg Handler *FINAL_TARGET_REACHED(FTRmsg)*:
  setGreenLightAndResumeFlow();

V. ANALYSIS

In this section, we study the number of modules used to construct the scaffold, and the time complexity of the reconfiguration method in the case of a pyramid shape with a height of h tiles. We also compare this work with the synchronous construction time of the scaffold presented in [15].

The total number of modules engaged in the scaffold construction algorithm corresponds to the number of modules constituting the scaffolding structure $N_{scaffold}$ added to the excess modules sent by the algorithm to anticipate future constructions.

The expression of $N_{scaffold}$ and of the number of tiles N_{tiles} in a pyramid of height h tiles are calculated in [15], and reminded below:

$$N_{scaffold} = (2b - \frac{1}{3})h^3 + (\frac{9}{2} - 2b)h^2 + \frac{5}{6}h$$
$$N_{tiles} = \frac{h^3}{3} + \frac{h^2}{2} + \frac{h}{6}$$

Excess modules are present along the paths formed by the ascending branches of the structure. In the case of the pyramid, for each level $i \in [2..h]$ there are $(h-i+1)^2$ tiles, each receiving 4 ascending branches from the lower level. Therefore, the total number of ascending branches is:

$$N_{Zbranch} = \sum_{i=2}^{h} 4(h-i+1)^2 = \frac{4h^3 - 6h^2 + 2h}{3}$$

For each branches of length b, excess modules will fill the *EPL* cell at the extremity of the branch, and a number of internal position along the branch, keeping one free position between any two mobile modules. Thus, we obtain $e = \frac{b}{2} - 2$, the average number of modules per branch plus 4 modules on *EPL* cells for each tile.

We then deduce the total number of modules $N_{modules}$:

$$\begin{aligned} N_{modules} &= N_{scaffold} + 4N_{tile} + e \times N_{Zbranch} \\ &= \frac{h^3}{3}(8b+5) + h^2(\frac{25}{2} - 3b) + \frac{h}{6}(2b-3) \end{aligned}$$

We can conclude that in the case of a pyramid of height h, and in spite of excess modules, the complexity of N_{module} is still $O(h^3)$.

In the same way as in [15], we base our analysis on the construction tree of the scaffold, consisting of the tiles of the scaffold as vertices, and expressing the precedence in the construction of these tiles through its edges. The first tile of the pyramid, located at $(0,0,0)$, is the root of the tree.

By studying displacement rules, we can observe that the time needed to place the root module of the first tile of each level i of the pyramid takes $16(b-1)$ time steps (ts). We can deduce that the sum of the waiting time and the motion time necessary for a tile root module to reach its position depends on its height i in the construction tree, which can be expressed as:

$$T_{tile} = [16(b-1)] \times ts$$

T_{tile} does not depend on its height in the construction tree i, while in the previous version with waiting times, we had $T_{tile}(i) = 24 + 4b + 2b \times i$. This is the reason why the time complexity of this version is linear, as proven below.

Theorem 1: The time complexity of the asynchronous construction of the scaffold is $O(N^{\frac{1}{3}})$.

Proof: Considering that the height of the construction tree is $O(h)$, we have:

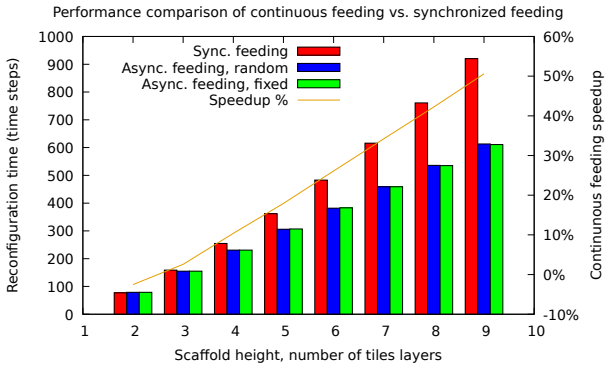

Fig. 3. Synchronous vs. continuous feeding comparison

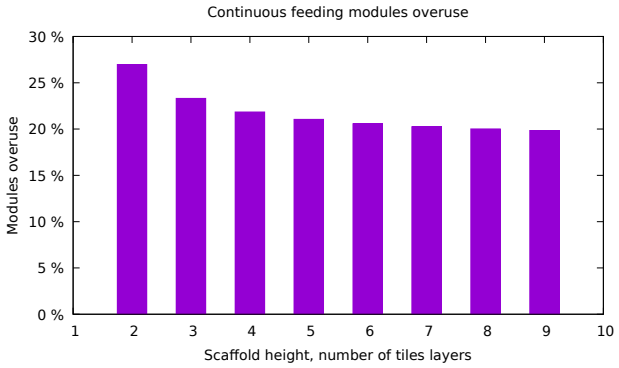

Fig. 4. Modules overuse by continuous feeding

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

$$T_{old} = \sum_{i=1}^{h} 24 + 4b + 2b \times i = 24h + b(5h + h^2) \qquad (1)$$

Then, in the case of the pyramid of height h, considering that the number of modules is $N_{module} = O(h^3)$, we can conclude that the complexity of the reconfiguration time is $O(N_{modules}^{\frac{1}{3}})$. ■

It is worth noting that for the same configuration of height h, our method admits a time complexity of $O(h)$, a factor of h lower than our previous algorithm with waiting times admitting $O(h^2)$ time steps, and without relying on any synchronization.

VI. SIMULATION

We conduct various simulations to evaluate our contributions using *VisibleSim* [10], a modular robot simulator. We study the following:

- Compare synchronous and asynchronous self-reconfiguration algorithms with varying scaffold sizes. We focus on studying the impact of the asynchronous algorithm in terms of modules count and total reconfiguration time. We measure the reconfiguration speedup and modules usage as performance indicators compared to the synchronous algorithm.
- We compare an ideal fixed-time module movement model, with a more realistic model, where modules have a pseudo random movement duration defined as a normal distribution $X \sim \mathcal{N}(\mu, \sigma^2)$, where μ is the fixed value, and σ can be configured for simulating of varying movement reliability.
- We run those tests for various scaffold height h, where h is the number of tiles layers.

In Figure 3, we compare the construction time of a scaffold for various scaffold heights. The figure shows the construction time in simulator time steps in Y-axis for several h values in X-axis. Three algorithms are compared: synchronous feeding, and two variants of continuous feeding, one with fixed movement time, one with pseudo random varying movement time. We make several observations from the results: first, continuous feeding performs faster than synchronous feeding, with a speedup increasing as the scaffold height increases (This can be seen in the video showing the side-by-side execution of the two algorithms, accessible from footnote[3]. Second, both variants of continuous feeding perform almost identically, which shows that our motion coordination algorithm allow modules to synchronize with their predecessors in a very efficient manner.

Figure 4 shows the percentage of module overuse due to continuous feeding, this can be compared to synchronous feeding, which has no overuse. These modules are not lost since they can be sent back to the sandbox, or used for further operations. What seems interesting is that this unused quantity starts at 36% for a small scaffold and quickly drops and stabilizes to about 25% when $h \geq 6$. We provide an analysis of the convergence of the surplus as the size of the structure increases below.

Theorem 2: The rate of modules in excess has an infinite limit lower than 25%.

Proof: We express the number of modules in excess $E(h)$ depending on the height of the pyramid by:

$$E(h) = N_{Zbranch} \times e$$

$$\frac{E(h)}{N_{modules}(h)} = \frac{(b-4)(2h^3 - 3h^2 + h)}{h^3(8b+5) + 3h^2(\frac{25}{2} - 3b) + h(b - \frac{1}{2})}$$

If we calculate $\lim_{h\to\infty} \frac{E(h)}{N_{modules}(h)}$, we get:

$$\lim_{h\to\infty} \frac{E(h)}{N_{modules}(h)} = \frac{1}{4} - \frac{37}{32b+20}$$

We can conclude that the rate of modules in excess is less than 25% for large size pyramids. ■

As the construction time gain increases and modules overuse remains stable as the size of the construction increases, we conclude that our new algorithm, continuous feeding, scales better than synchronous feeding. It is a key

[3]Side-by-side comparison video of synchronous feeding and async: `https://youtu.be/XpG20m7waJk`

property when dealing with programmable matter, since we aim at building shapes based on micro-robots which will require an enormous amount of robots.

VII. CONCLUSION

In this paper, we propose an improved and asynchronous version of our algorithm for building a scaffold using micro-robots. This algorithm uses a continuous feeding of the modules into the goal shape thus speeding the completion time by a factor of $O(N^{1/3})$ compared to the previous version, owing to the removal of waiting times. We also propose a coordination algorithm to avoid modules collisions during the reconfiguration. This algorithm is expressed as a set of callbacks, raised by events and messages, which sets a traffic light-like module inner state indicating when it must move and when it must wait.

Our proposal is experimented on a simple goal shape: a pyramid, defined by its global height and its individual tile size, and we provide results for varying values of both parameters. From the results, we show that our algorithm is promising: it is scalable and the modules overuse is, in percentage, stable. Furthermore, the modules unused by the scaffold could still be useful for further operations, for instance to fill the outer envelop of the goal shape. Our results also show that small motion time variations have a negligible impact on the whole reconfiguration time, making our algorithm robust to physical variations in module movements.

VIII. FUTURE WORKS

We envision as future works to first eliminate the necessity for a module surplus, which stands as the main drawback of this method, and would necessitate that each *Coordinator* geometrically computes the resource requirements of each children tiles. Then, we aim to extend this work to all convex shapes, for which we think our reconfiguration time result can be preserved. This requires supporting the construction of incomplete branches (i.e., where their length $l < b$), and new rules for feeding any growing branch from any vertical branch below. Next, we focus on concave shapes, which would necessitate the design of a coordination mechanism for modules to traverse tiles horizontally from any incident branch, as well as a way to construct tile branches in the opposite direction. This jump to generic shapes would also require a systematic method for generating a scaffold structure from the 3D description of an object. Furthermore, we plan to allow the reconfiguration of an arbitrary scaffold into another, by absorbing part of the structure to construct another in an efficient manner. Finally, we aim to design a coating algorithm that would enable one or several layers of modules to cover the scaffolding structure and therefore preserve the external aspect of the represented object.

ACKNOWLEDGMENT

This work was partially supported by the ANR (ANR-16-CE33-0022-02), the French Investissements d'Avenir program, ISITE-BFC project (ANR-15-IDEX-03).

REFERENCES

[1] Julien Bourgeois, Benoit Piranda, Andre Naz, Nicolas Boillot, Hakim Mabed, Dominique Dhoutaut, Thadeu Tucci, and Hicham Lakhlef. Programmable matter as a cyber-physical conjugation. In *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*, pages 002942–002947. IEEE, October 2016.

[2] D. J. Dewey, M. P. Ashley-Rollman, M. De Rosa, S. C. Goldstein, T. C. Mowry, S. S. Srinivasa, P. Pillai, and J. Campbell. Generalizing metamodules to simplify planning in modular robotic systems. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 1338–1345, September 2008.

[3] T. Fukuda and Y. Kawauchi. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. pages 662–667. IEEE Comput. Soc. Press, 1990.

[4] Seth Copen Goldstein, Jason D. Campbell, and Todd C. Mowry. Programmable matter. *Computer*, 38(6):99–101, 2005.

[5] Hiroshi Kawano. Distributed Tunneling Reconfiguration of Sliding Cubic Modular Robots in Severe Space Requirements. In *DARS 2018, 14th International Symposium on Distributed Autonomous Robotic Systems*, page 14, 2018.

[6] K. D. Kotay and D. L. Rus. Algorithms for self-reconfiguring molecule motion planning. In *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 3, pages 2184–2193, 2000.

[7] Jakub Lengiewicz and Pawel Holobut. Efficient collective shape shifting and locomotion of massively-modular robotic structures. *Auton. Robots*, 43(1):97–122, 2019.

[8] André Naz, Benoît Piranda, Julien Bourgeois, and Seth Copen Goldstein. A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on*, pages 254–263. IEEE, 2016.

[9] An Nguyen, Leonidas J. Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. In *Proc. of 4th International Workshop on Algorithmic Foundations of Robotics*, pages 23–36, 2000.

[10] Benoit Piranda. VisibleSim: Your simulator for Programmable Matter. In *Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271)*. Dagstuhl, May 2016.

[11] Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, February 2018.

[12] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *2012 IEEE International Conference on Robotics and Automation*, pages 3293–3298, St Paul, MN, USA, May 2012. IEEE.

[13] Kasper Støy. Using cellular automata and gradients to control self-reconfiguration. *Robotics and Autonomous Systems*, 54(2):135 – 141, 2006.

[14] Kasper Støy and Radhika Nagpal. Self-Reconfiguration Using Directed Growth. In *Distributed Autonomous Robotic Systems 6*, pages 3–12, 2007.

[15] Pierre Thalamy, Benoît Piranda, and Julien Bourgeois. Distributed Self-Reconfiguration using a Deterministic Autonomous Scaffolding Structure. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 140–148, Montreal QC, Canada, May 2019.

[16] Thadeu Tucci, Benoit Piranda, and Julien Bourgeois. A Distributed Self-Assembly Planning Algorithm for Modular Robots. In *International Conference on Autonomous Agents and Multiagent Systems) (AAMAS)*, Stockholm, Sweden, July 2018. Association for Computing Machinery (ACM).

[17] S. Vassilvitskii, M. Yim, and J. Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 1, pages 117–122 vol.1, 2002.