Fault-Tolerance Mechanism for Self-Reconfiguration of Modular Robots

Jad Bassil, Perla Tannoury, Benoît Piranda, Abdallah Makhoul, Julien Bourgeois
Univ. Bourgogne Franche-Comté
FEMTO-ST Institute, CNRS, Montbéliard, France
{first}.{last}@femto-st.fr

***Abstract*—A Modular Self-Reconfigurable Robot (MSR) is an Internet of Robotic Things object (IoRT) composed of an ensemble of independent communicating robotic modules that can self-reconfigure to change their initial shape into a goal one. Self-reconfiguration is known to be an intricate and complex task and faults such as broken connections, loss of power, incomplete motions... are likely to occur during the self-reconfiguration process. However, existing work on self-reconfiguration considers fault-free robotic modules and does not apply any fault-tolerance mechanisms.**

In this paper, we propose a fault-tolerance mechanism that can be applied to a broken interface which results in communication failures in the context of the self-reconfiguration of a *3D Catom* robot using the deterministic scaffold assembly algorithm. We introduce a new module role: the *Helper* module. The *Helper* module serves as a communication bridge between two modules attached by a broken interface. We showed in simulation the efficiency of our approach dealing with communication failures caused by broken interfaces.

***Index Terms*—Fault-Tolerance, Modular Robots, Self-Reconfiguration, Internet of Robotic Things**

I. Introduction

In nature, some very robust organisms can be found. For example, the human body is able to regain health after bones or skin breaks. One of the reasons for the success of the human body is the trillions of cells making up the body. There is no single point of failure for multi-cellular organisms. This can be ensured by the redundancy of cells which may separate, relocate, or fix the life form they make. Robots, in actuality, are overall not very fault-tolerant. The passing of a sensor, an actuator or a piece of mechanics will in most cases leave the robot completely defenseless and incapable to play out its capacity.

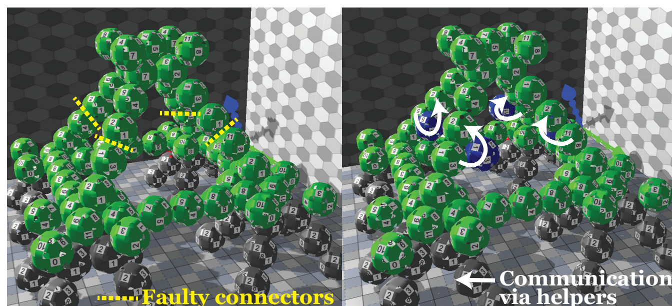

Fig. 1. Presentation of the problem: in the left picture yellow lines shows the broken connection and in the right picture the solution using *Helper* module in blue to maintain the communication.

Inspired by multi-cellular organisms, modular self-reconfigurable robots (MSR) consist of many interconnected robot modules [1] forming an intranet of robotic things. Considering the ensemble as a whole, the set of modules is a complete Internet of Robotic Things (IoRT) object that can participate in the Internet of Things (IoT) ecosystem. They are able to communicate to coordinate their tasks and move to self-reconfigure from their current shape into a goal one. However, modules are prone to failures. Faults such as a broken interface, a loss of power, an incomplete motion, a faulty docking... can probably occur during the self-reconfiguration process. Moreover, self-reconfiguration is known to be an intricate task that requires complex planning and coordination between modules to move and reach their goal positions. Many self-reconfiguration algorithms exist in the literature, yet, to the best of our knowledge, they don't consider modules failures during the self-reconfiguration process. Therefore, in order to tolerate faults, some mechanisms must be implemented to guarantee the completeness of a self-reconfiguration algorithm to successfully achieve the desired goal shape.

Programmable matter is a matter with the ability to change its physical properties such as its color and shape on demand or due to a certain stimuli from its environment. MSRs can be used to achieve such matter by building objects composed of an ensemble of connected quasi-spherical micro-robots capable of computing, communicating and moving around each other to coordinate their tasks and self-reconfigure to change the shape of the object. In this context, Thalamy et. al [2] envisions to a achieve a programmable matter by representing an object as a porous internal scaffold formed with quasi-spherical modules called *3D Catoms*. The scaffold is then coated with a thin layer of modules to better represent the goal object [3].

In [4], an algorithm is described for building the scaffold. In which, modules flow continuously in the upward direction starting from a reserve of modules placed underneath the scene called *Sandbox*. During their motions, modules use a message-passing motion coordination algorithm inspired by traffic-light systems to keep enough space between them and other flowing modules. They also exchange messages to plan their motions. Therefore, a communication failure can interrupt the flow of

modules and cause an incomplete scaffold construction.

In this paper, we aim to propose a fault-tolerance mechanism to prevent incomplete scaffold construction in case of a communication failure caused by a broken interface between two adjacent modules. For this purpose, we introduce the *Helper* module: a module that will serve as communication bridge between two adjacent modules attached by a broken interface. We describe a method that can be easily added to the motion coordination algorithm to put the *Helper* module in its place to reactivate the communication as shown on Figure 1.

The remainder of this paper is organized as follows. Section 2 starts by introducing the related work, followed by an overview of the fundamentals of our method in Section 3. Section 4 describes the proposed solution. Lastly, Section 5 presents results and simulations of our method performed on the VisibleSim [5] simulator for modular robots, before concluding the paper in Section 6.

II. Related work

Modular robots are made up of multiple modules that are communicating with one another [6]. A fault can originate from any module in the system, regardless of where the unexpected system failure occurs.

Once the fault in a module actually manifests itself and becomes active, it causes some unexpected behavior. Different faults will result in different behaviors, but the common thread here is that they are unexpected. An expected behavior in our system effectively means that some part of our system did something that we did not plan for. These robots are becoming increasingly prevalent in our digital era, fault-tolerant research has begun to shed light on the internal failures of the robot's hardware and software systems [7] [8]. The focus is shifting to enable the robot to prolong its working life and maintain as much of its functionality as possible with minimal cost and less human intervention. In order to support these internal fault-tolerant capabilities, there is a need to develop and master robotic technology to detect and engrave failures. Before digging in more in the modular robot faults, the concept of scaffolding should be introduced.

Scaffolding in self-reconfiguration is not a novel approach, it started in [9], where the ensemble could be made porous by constructing multi-module structures that allowed modules to flow freely internally, simplifying design, lowering the amount of modules that needed to be displaced, thus speeding up completely the reconfiguration. The scaffolding concept was further studied in [10] and [11] using a very simple scaffold geometry thanks to the cubic sliding and rotating robotic model's simplicity. They devised an effective deterministic reconfiguration method, based on cellular automata and simple gradients. The same scaffold geometry later was the inspiration of [12], with a similar model but resolving reconfiguration using a max-flow search to optimize the flow of the modules between the boundaries of the initial shape and those of the goal shape. They both achieved self-reconfiguration using a number of individual movements linear in the number of modules that are presented in the system.

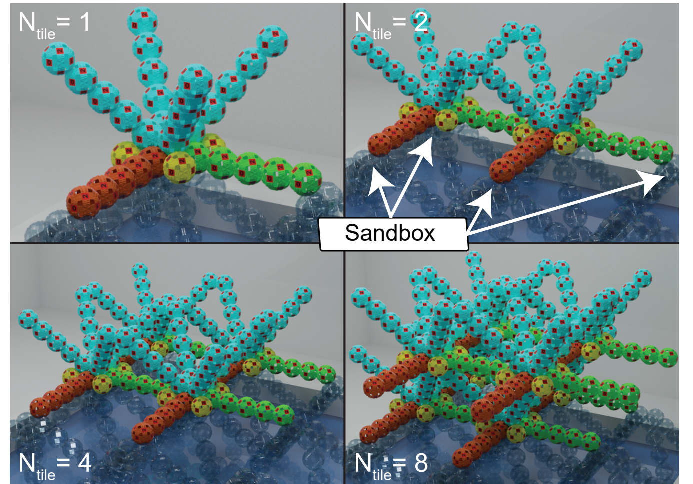

Fig. 2. Anatomy of the entire scaffold: Breakdown of a sample scaffold consisting of an arrangement of 8 tiles with all branches grown, directly over the sandbox [4].

However, these scaffold approaches have considered an initial form as a pre-built scaffolding, but none responded to the construction of the scaffolding structure from a mass of modules, which it was introduced and developed in [4]. This work showed that scaffolds with complex geometries can be considered as a very powerful tool to facilitate the self-reconfiguration of massive modular robots.

In order to have fault-tolerant robots using the scaffold method, one must be able to detect a failure or an error in their systems. Many fault detector algorithms were developed in [13] [14] [15] [16] [17] to identify failures quickly without depending on a human's senses, allowing the robot to rely only on the information it can glean from its existing sensors.

In our case, we are interested in tolerating communication failure caused by a broken interface on one of the modules forming the scaffold to void the interruption of the construction.

III. Prerequisites

A. 3D Catom modular robot

Our solution focuses on the self-reconfiguration of modular robots. These robots are made of quasi-spherical rotating modules named *3D Catoms* [18]. The *3D Catoms* have a quasi-spherical geometry consisting of 12 flat squares (named connectors) linked by curves.

These 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, see Figure 3. The *3D Catoms* are placed in a face-center cubic lattice and each *3D Catom* can be connected to up to 12 neighbors. All *3D Catoms* share the same coordinate system and each *3D Catom* stores in its internal memory its own coordinates and update them after each movement. A *3D Catom* can move to a free adjacent position by rotating on a fix module that serves as a pivot. *3D Catoms* motions must be coordinated by message-passing to avoid collisions and

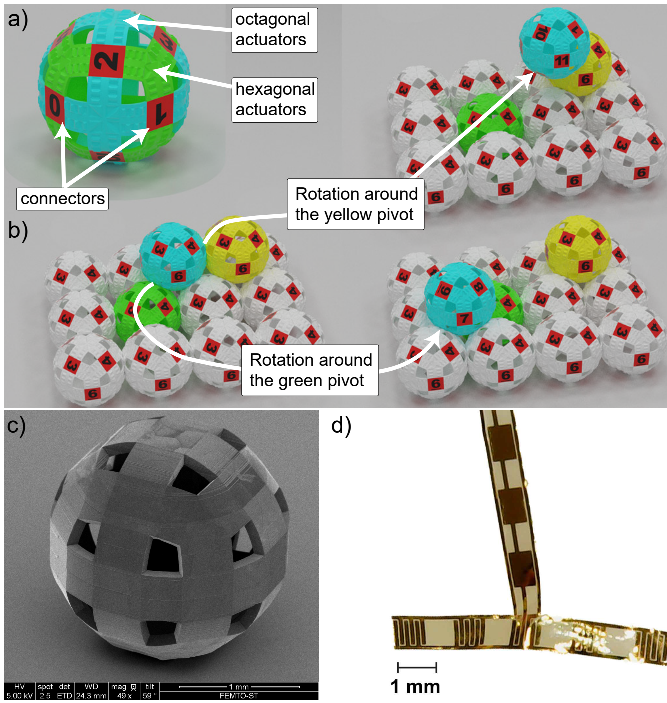

Fig. 3. The *3D Catom*, a) shape and actuators, b) the two kind of rotations, c) first prototype of mm size printed structure and d) electrostatic actuator.

blocking. They can detect and react to several internal and external events such as the reception of a message, an added or removed neighbor, an interruption event...

In addition, a *3D Catom* can detect the presence of a connected neighbor to one of its interfaces. In the context of this paper, the *3D Catom* can also detect that a connection is broken after a motion due to a faulty docking procedure.

B. Scaffold model

To accelerate the self-reconfiguration of a programmable matter based on modular robots, Thalamy et al. [2] described a novel 3D self-reconfiguration approach for programmable matter composed of *3D Catoms*. The idea is to represent the object using an internal scaffold coated by thin layer of modules. The advantage of using a scaffold model is to leave internal holes inside the shape thus facilitating coordination and parallel motions inside the structure while requiring less number of modules to represent the goal shape.

Their proposed scaffold is composed of tiles as its basic building units. Each tile can have up to six branches: two horizontal branches placed along the $\overrightarrow{x}$ and $\overrightarrow{y}$ axis and four upward branches as seen in Figure 2. In [2] a distributed self-reconfiguration algorithm that builds the scaffolding structure is described. *3D Catoms* continuously flow upward at various ground location of the scene from a reserve of modules placed underneath the structure called *Sandbox* to construct the tiles that corresponds to a given goal shape. Modules flow following the same paths along the branches in a train-like fashion. A motion coordination algorithm detailed in the next Section III-C that uses a message-passing traffic-light style is used to keep a safe space between flowing modules to avoid blocking. Our solution is a plugin to be added to the motion.

C. Motion coordination algorithm

As mentioned in the previous section, during the construction, modules flow inside the structure to build the scaffold. The motion coordination algorithm is used to keep enough space between flowing modules to prevent blocking and collision. During the scaffold construction, modules take different roles. We are concerned with two roles:

1) *FreeAgent* modules: are flowing modules inside the structure.
2) *Beam* modules: are modules already in their final positions forming the branches of the tiles. They are used as pivot for *FreeAgent* modules.

Figure 4 shows the traffic-light style state transitions during the motion coordination algorithm to keep safe distance between moving *FreeAgent* modules. Before a motion, a *FreeAgent* must query the pivot and its next latching points to check that their light states are green. The light state of a beam changes as follow: when a *FreeAgent* attaches to a *Beam*, the beam changes its light state to red blocking the flow of next modules. It switches back to green when the *FreeAgent* is removed so the flow can resume. Below is the summary of the messages used in the motion coordination algorithm, their function, and their data. They are exchanged between *Beam* modules and *FreeAgent* modules:

1) PROBE_LIGHT_STATE (sender, nextPos) (PLS): Sent by a *FreeAgent* about to move to ask permission by checking the light states of its next latching points. It has the sender's position for routing the reply, as well as the position it wants to shift to (nextPos). A receiver uses the latter to forward the message to all modules that will be attached to the *FreeAgent* after its motion.

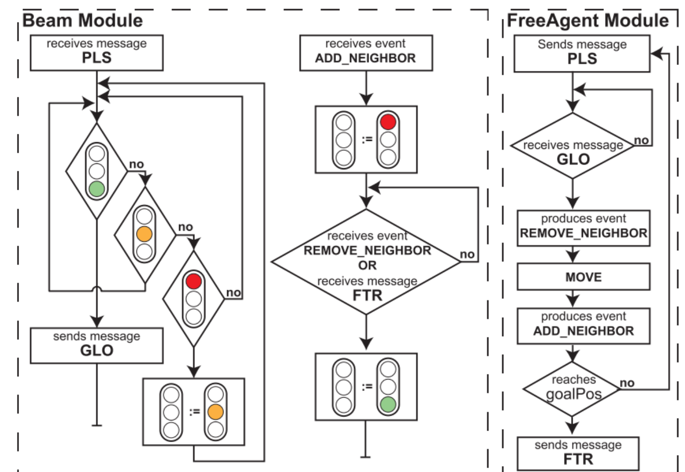

Fig. 4. Light state transition during the fault-free motion coordination algorithm [4].

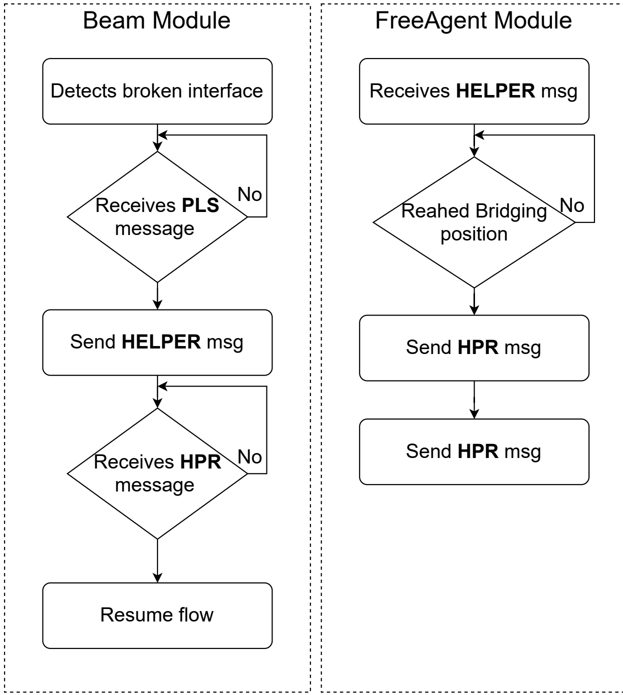

Fig. 5. Beam and FreeAgent flow chart: Simplified view of the behavior of *Beam* and *FreeAgent* modules states in case of a broken interface detected at the *Beam*.

2) GREEN_LIGHT_ON (GLO) [recipient]: This is a response to PLS, sent back to the module which made the request if the light state is green before moving.
3) FINAL_TARGET_REACHED () (FTR): Sent by the module that has just arrived at its claimed scaffold position to instruct the pivot to change its light state to green to continue the flow of modules even if it is still attached to it in this circumstance.

For a more detailed description of the motion coordination algorithm the reader can refer to [4].

IV. Proposed Method

To achieve fault tolerance for self-reconfigurable modular robots in case of a communication failure between two modules, some modifications were added to the motion coordination algorithm. We recall that modules are continuously flowing upward starting from the $Sandbox$. Therefore, when an interface between two *Beam* modules breaks, the flow is interrupted which affects the construction of a tile's branch which results in an incomplete construction. We solve this issue by introducing a new module's role: the *Helper* module. The *Helper* module will be placed between two modules connected with a broken interface to serve as a communication bridge between them as shown in Figure 6. All messages that need to be sent via a broken interface are then handled by the *Helper*.

In order for a *Helper* to take its place, we added two new messages to the motion coordination algorithm:

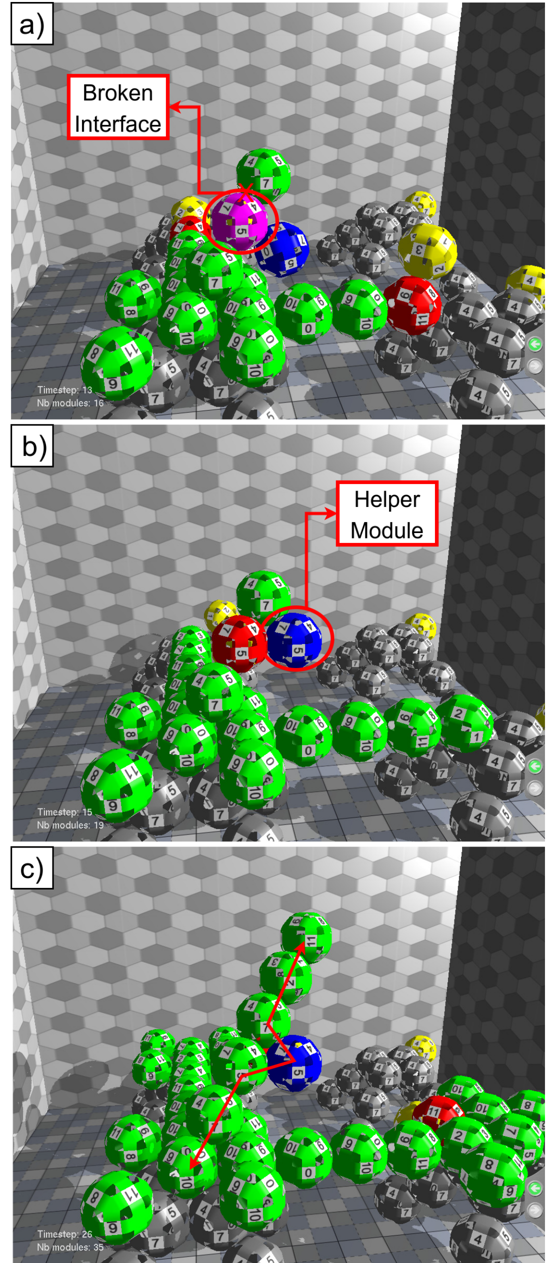

Fig. 6. A simulation example for handling a broken interface. a) a broken interface is detected. b) the *Helper* module creates a bridge of communication between disconnected modules. c) communications through the broken interface are passed through the *Helper*.

1) *HELPER(bridgingPosition)*: Sent by a module that has a broken interface to the sender of a PLS message to change its role from *FreeAgent* to *Helper*. The *Helper* module will keep rotating until it reaches the bridging position between the broken module and its neighbor. The bridging position is chosen in way to not interfere the flow of the other modules. Once arrived, it will send a "HELPER_POSITION_REACHED" message to the broken module.
2) HELPER_POSITION_REACHED() (HPR): HPR is sent

by the module that has just arrived to its bridging position to instruct the broken *Beam* to continue the flow of modules.

Algorithm 1: Motion algorithm pseudo-code for the *Beam* module role in case of a broken interface.

```
Msg Handler PROBE_LIGHT_STATE(sender, nextPos):
  | dst ←computeLightPivotForTarget(motionTarget);
  | if dst is brokenInterface then
  |   | state ← RED;
  |   | send HELPER(bridgingPos) to sender;
  | else
  |   | Execute the original motion coordination
  |   |   algorithm;
  | end
end
Msg Handler HELPER_POSITION_REACHED():
  | setGreenLightOnandResumeFlow();
end
Function setGreenLightAndResumeFlow():
  | if state = ORANGE then
  |   | send GREEN_LIGHT_ON() to
  |   |   waiting module;
  | end
  | state ← GREEN;
end
```

Algorithm 1 and 2 shows the algorithms executed on a *Beam* and a *FreeAgent* in case of a broken interface on the *Beam*. When the beam detects a broken interface it waits until it receives a PLS message. Then, instead of executing the original motion coordination algorithm, it sends a HELPER message containing the bridging position to the sender. On reception, the *FreeAgent* rotates to the bridging position to become a *Helper* then sends a HPR message to the broken *Beam*. The *Beam* modules connected to the *Helper* can then resume the flow and use the *Helper* as a bridge to send and receive messages.

V. Simulation

The proposed algorithm was implemented and evaluated using *VisibleSim*: a discrete-event simulator for modular robots.

Algorithm 2: Motion algorithm pseudo-code for the *FreeAgent* module role in case of a broken interface

```
Msg Handler HELPER(bridgingPosition):
  | rotateTo(bridgingPosition);
  | send HELPER_POSITION_REACHED() to
  |   brokenbeam;
end
Msg Handler GREEN_LIGHT_ON():
  | rotateTo(nextPos);
end
```

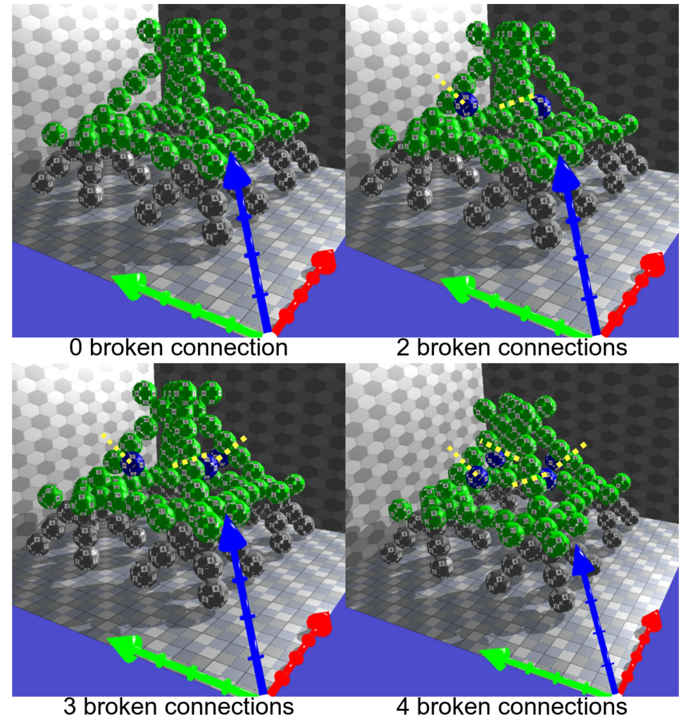

Fig. 7. Four complete constructions of a pyramid made of four tiles while having 0,2,3 and 4 broken interfaces (in yellow) on upward branches. *Helper* modules are in Blue serving as a communication bridge.

Our simulations consisted of constructing a pyramid with 2x2 tiles with a branch size of 6 modules shown in Figure 7. We evaluated our solution by breaking one interface on each of the upward branches. Figure 6 shows simulation snapshots that demonstrate how a module handles a broken interface that disables the communication with the next adjacent module in the branch during the construction of a tile.

The video[1] introduces the construction of the pyramid scaffold with 4 simulated broken interfaces on each upward branch of the 4 tiles. It shows the placement of the helper module and how it allows the continuity of communication between the elements of the branch. Finally, it shows in comparing the construction of the same pyramid with 3 and 4 broken connection the effect of the algorithm on the global time of reconfiguration.

Figure 8 shows how the number of exchanged messages varies when the number of faults increases. Each fault requires a fixed number of messages exchanged between the faulty *Beam* module and the *Helper* module. Hence, the number of exchanged messages increases linearly in the number of faults.

Figure 9 shows how the global simulation time varies when the number of faults increases. A fault in a tile's branch only affects the time of construction of the branch. Tiles are constructed in a specific order as explained in [4] and can be seen in video[1]. The construction order can be represented as a tree where nodes are tiles and edges represent the precedence of construction. The tree is rooted at a tile placed at one

[1] Youtube video link: https://youtu.be/llxhcyfpBa0

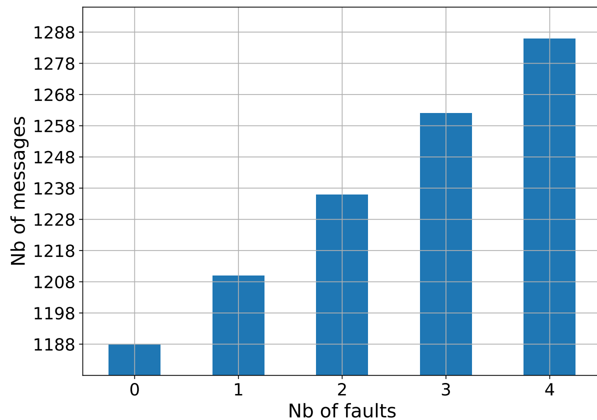

Fig. 8. Number of exchanged messages vs the number of faults.

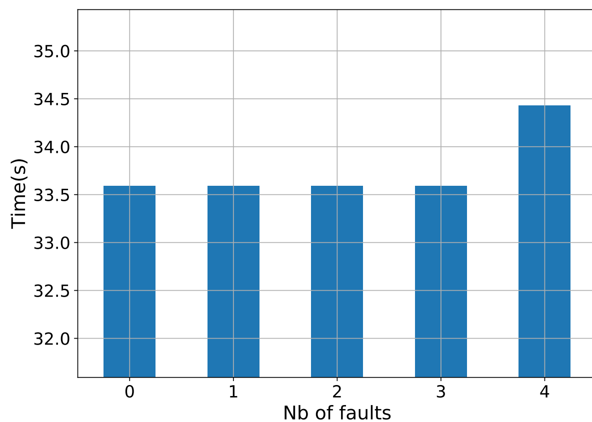

Fig. 9. Time in seconds vs the number of faults.

corner of the goal shape. Therefore, when a fault occurs, the construction time of the whole scaffold is affected if it occurs in a branch connecting a parent tile to one of its child tiles or in the last tile being constructed (a leaf tile). The simulated example presents the latter case when the fourth fault occurs in a leaf tile.

VI. Conclusion and Future Work

In this paper, we adapted the deterministic scaffold assembly algorithm to deal with communication failures caused by a broken interface during the construction of the scaffold. The presented solution uses a special module as a communication bridge between two modules attached by a broken interface to guarantee that other modules continue their flow to reach their goal position. Simulated experiments show that the emergence of a bridging module in case of a communication failure is efficient to complete the construction of the goal shape.

Future works include to experiment fault-tolerance on different shapes of modular robots. Then, find solutions to tolerate other types of faults related to motion, docking and loss of power. In addition, we aim to generalize this method to other scaffold structures. Furthermore, we aim to make use of the *Helper* module in coating after the scaffold construction is done.

Acknowledgment

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

References

[1] T. Fukuda and S. Nakagawa, "Dynamically reconfigurable robotic system," in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*. IEEE, 1988, pp. 1581–1586.

[2] P. Thalamy, B. Piranda, and J. Bourgeois, "Engineering efficient and massively parallel 3d self-reconfiguration using sandboxing, scaffolding and coating," *Robotics and Autonomous Systems*, vol. 146, p. 103875, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889021001603

[3] ——, "3D coating self-assembly for modular robotic scaffolds," *IEEE International Conference on Intelligent Robots and Systems*, pp. 11 688–11 695, 2020.

[4] P. Thalamy, B. Piranda, F. Lassabe, and J. Bourgeois, "Deterministic scaffold assembly by self-reconfiguring micro-robotic swarms," *Swarm and Evolutionary Computation*, vol. 58, p. 100722, 2020.

[5] P. Thalamy, B. Piranda, A. Naz, and J. Bourgeois, "Behavioral simulations of lattice modular robots with visiblesim," in *15th International Symposium on Distributed Autonomous Robotic Systems (DARS 2021)*, Kyoto, Japan, jun 2021. [Online]. Available: https://publiweb.femto-st.fr/tntnet/entries/17403/documents/author/data

[6] M. Yim, Y. Zhang, and D. Duff, "Modular robots," *IEEE Spectrum*, vol. 39, no. 2, pp. 30–34, 2002.

[7] A. Hereau, K. Godary-Dejean, J. Guiochet, and D. Crestani, "A fault tolerant control architecture based on fault trees for an underwater robot executing transect missions," in *International Conference on Robotics and Automation (ICRA 2021)*, 2021.

[8] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, "Expert system framework for fault detection and fault tolerance in robotics," *Computers & electrical engineering*, vol. 20, no. 5, pp. 421–435, 1994.

[9] K. D. Kotay and D. L. Rus, "Algorithms for self-reconfiguring molecule motion planning," in *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*, vol. 3. IEEE, 2000, pp. 2184–2193.

[10] K. Stoy, "Using cellular automata and gradients to control self-reconfiguration," *Robotics and Autonomous Systems*, vol. 54, no. 2, pp. 135–141, 2006.

[11] K. Stoy and R. Nagpal, "Self-reconfiguration using directed growth," in *Distributed autonomous robotic systems 6*. Springer, 2007, pp. 3–12.

[12] J. Lengiewicz and P. Hołobut, "Efficient collective shape shifting and locomotion of massively-modular robotic structures," *Autonomous Robots*, vol. 43, no. 1, pp. 97–122, 2019.

[13] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, "Robotic fault detection and fault tolerance: A survey," *Reliability Engineering & System Safety*, vol. 46, no. 2, pp. 139–158, 1994.

[14] F. E. Gelhaus and H. T. Roman, "Robot applications in nuclear power plants," *Progress in nuclear energy*, vol. 23, no. 1, pp. 1–33, 1990.

[15] R. F. Stengel, "Intelligent failure-tolerant control," *IEEE Control Systems Magazine*, vol. 11, no. 4, pp. 14–23, 1991.

[16] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[17] E. Chow and A. Willsky, "Analytical redundancy and the design of robust failure detection systems," *IEEE Transactions on automatic control*, vol. 29, no. 7, pp. 603–614, 1984.

[18] B. Piranda and J. Bourgeois, "Designing a quasi-spherical module for a huge modular robot to create programmable matter," *Autonomous Robots*, vol. 42, no. 8, pp. 1619–1633, dec 2018. [Online]. Available: http://link.springer.com/10.1007/s10514-018-9710-0