

THÈSE DE DOCTORAT

DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

PIERRE THALAMY

Distributed Algorithms and Advanced Modeling Approaches for Fast and Efficient Object Construction Using a Modular Self-reconfigurable Robotic System

(Algorithmes distribués et méthodes de modélisation avancées pour une construction rapide et efficace d'objets avec un robot modulaire auto-reconfigurable)

Thèse présentée et soutenue à Montbéliard, le 9 Octobre 2020

Composition du Jury :

NICOLAS ANDREFF	Professeur à l'Université de Franche-Comté	Président
HAMANN HEIKO	Professeur à l'Université de Lübeck, Allemagne	Rapporteur
STØY KASPER	Professeur à IT University of Copenhagen, Danemark	Rapporteur
MARTINOLI ALCHERIO	Professeur Associé à l'École Polytechnique Fédérale de Lausanne, Suisse	Examineur
BOURGEOIS JULIEN	Professeur à l'Université de Franche-Comté	Directeur de thèse
PIRANDA BENOÎT	Maître de conférences à l'Université de Franche-Comté	Codirecteur de thèse

N°

X	X	X
---	---	---

ACKNOWLEDGEMENT

First of all, I would like to thank all thesis committee members for having accepted to take the time to review my dissertation and provide valuable feedback out of their certainly busy lives.

Then, I would have never been able to achieve the research discussed in this manuscript were it not for the excellent and mutually complementary guidance of my thesis advisors Prof. Julien Bourgeois and Dr. Benoît Piranda. I am grateful to Benoît for the countless hours he has spent creating stunning visuals illustrating my work and having to deal with my excessive meticulousness in that process. Moreover, I owe much to Julien, who has had a very positive influence on my life over the past 6 years, through his generous and countless advice, the trust he has invested in me, and by having offered me several opportunities to travel the world and discover scientific research. I consider my relationship with both to now go beyond just simple collaboration, into the realm of friendship. I wish them the best of luck in their research and to find the finest PhD students so they can realize their groundbreaking vision of programmable matter.

Furthermore, I am seizing this opportunity to thank all other members of the OMNI team at FEMTO-ST for their support and interest at various points of my thesis. I am grateful to Frédéric Lassabe in particular, who was involved for some time in the discussions surrounding my work.

These past three years would not have been the same without the good humor of my fellow PhD students and office mates Florian Pescher, Thadeu Tucci, and André Naz. In addition to their helpful comments and advice, our regular foosball tournaments have been a great help in relieving stress and fostering a positive workplace atmosphere. This also includes our interns. Special thanks to André Naz, whom I look up to, and in whose steps I have been walking for the past few years, providing reassurance.

I am also grateful to the French National Research Agency (ANR), for financially supporting the project entitled *“Hardware and software for creating programmable matter”* (ANR-16-CE33-0022) to which my thesis work contributes.

Another person who has inspired me to pursue a career in scientific research is Prof. Seth Coppen Goldstein, to whom I am grateful because he has sparked my interest for computer science while I was at Carnegie Mellon University for a summer internship on the Claytronics project in 2014.

Let us not forget the crucial supporting role my friends have played throughout this challenging professional and personal experience. I wish to thank my international friends from Montbéliard, some of them also PhD students, with whom we have mutually supported each other during the first half of my thesis.

[N.B.: the rest of the acknowledgment is written in French.]

Si ma dernière année de thèse a pu être aussi productive que sympathique, je le dois à la rencontre de mes amis ingénieurs de l'UTBM (et du fameux Bricard) qui ont fortement contribué à mon excellent moral. Merci aussi à Estelle et Quentin qui m'ont chaleureusement hébergé pendant le confinement dû au COVID-19. Cette période si étrange aura pu être particulièrement agréable grâce à eux, m'aidant à préserver un état d'esprit propice à l'avancement de mon travail de thèse.

Je remercie aussi mes amis de plus longue date, qui se sont préoccupés de l'avancement de mes recherches et ont su m'épauler malgré la distance. Merci à Tina Nikoukhah pour nos nombreux échanges sur nos expériences de thèse ; je lui souhaite bonne chance pour mener à bout son doctorat à l'ENS.

Finalement, je tiens à remercier ma famille, qui a toujours cru en moi et su m'apporter un soutien et un amour sans faille. Ses encouragements m'auront redonné de l'élan plus d'une fois. Je lui dédie donc ce manuscrit. Merci à ma Mimi pour sa patience, et son éternel soutien face aux nombreux doutes et incertitudes qui ont pu m'habiter durant ma thèse. Merci à ma mère, qui m'a plus que quiconque donné les moyens de réussir, en dépensant une énergie folle pour me soulager et rendre ma vie plus facile et plus heureuse. Je remercie mon père, le Dr. Thalamy original, pour avoir su constamment challenger mon esprit scientifique, et pour nos discussions mettant en parallèle son monde de la biologie et le mien. De plus, les moments de retrouvailles avec mon frère Louis auront aussi été un bol d'air pour m'évader temporairement du monde de la recherche et me ressourcer. Enfin, je pense à Édouard, mon grand-père, que je sais très fier de moi tout comme ma grand-mère, pour qui ces quelques mots en Français seront sûrement les seuls qu'il trouvera intelligibles de ce manuscrit.

To all the people that I have just mentioned and to everyone else who has contributed in one way or another to the realization of this work:

Thank You. Merci à vous.

ABSTRACT

Humans have always been on a quest to master their environment. But with the arrival of our digital age, an emerging technology now stands as the ultimate tool for that purpose: Programmable Matter. While any form of matter that can be programmed to autonomously react to a stimulus would fit that label, its most promising substrate resides in modular robotic systems. Such robotic systems are composed of interconnected, autonomous, and computationally simple modules that must coordinate through their motions and communications to achieve a complex common goal.

Such programmable matter technology could be used to realize tangible and interactive 3D display systems that could revolutionize the ways in which we interact with the virtual world. Large-scale modular robotic systems with up to hundreds of thousands of modules can be used to form tangible shapes that can be rearranged at will. From an algorithmic point of view, however, this self-reconfiguration process is a formidable challenge due to the kinematic, communication, control, and time constraints imposed on the modules during this process.

We argue in this thesis that there exist ways to accelerate the self-reconfiguration of programmable matter systems, and that a new class of reconfiguration methods with increased speed and specifically tailored to tangible display systems must emerge. We contend that such methods can be achieved by proposing a novel way of representing programmable matter objects, and by using a dedicated reconfiguration platform supporting self-reconfiguration.

Therefore, we propose a framework to apply this novel approach on quasi-spherical modules arranged in a face-centered cubic lattice, and present algorithms to implement self-reconfiguration in this context. We analyze these algorithms and evaluate them on classes of shapes with increasing complexity, to show that our method enables previously unattainable reconfiguration times.

Keywords: Programmable Matter, Distributed Algorithms, Modular Robotics, Self-reconfiguration, Multi-agent Systems

RÉSUMÉ

Les humains ont de tout temps cherché à contrôler leur environnement. Mais avec l'arrivée de l'ère numérique, une technologie émergente promet de devenir l'outil ultime de cette quête : la matière programmable. Bien que toute forme de matière pouvant être programmée pour réagir de façon autonome à un stimulus puisse prétendre à cette dénomination, son substrat le plus prometteur réside dans les systèmes robotiques modulaires. Ces systèmes robotiques sont composés de modules interconnectés, autonomes, et aux ressources limitées, devant se coordonner par leurs communications et leurs mouvements afin d'accomplir des tâches complexes.

La matière programmable pourrait être utilisée pour réaliser les systèmes de représentation de demain: des affichages tangibles et interactifs en 3D, qui promettent de révolutionner la façon dont nous interagissons avec le monde virtuel. Des ensembles de robots modulaires composés de plusieurs milliers de modules peuvent s'organiser pour former des objets tangibles capables de se transformer à l'infini sur demande. D'un point de vue algorithmique, cependant, ce processus d'autoreconfiguration représente un défi considérable à cause des contraintes cinématiques, temporelles, de contrôle, et de communication, auxquelles sont soumis les modules.

Nous défendons dans cette thèse qu'il existe des moyens d'accélérer la reconfiguration des systèmes de matière programmable, et qu'une nouvelle classe de méthodes de reconfiguration plus rapide et mieux adaptée aux systèmes de représentation tangibles doit voir le jour. Nous soutenons qu'il est possible de parvenir à de telles méthodes en proposant une nouvelle façon de représenter les objets faits de matière programmable, et en utilisant une plateforme d'assistance dédiée à l'autoreconfiguration.

Par conséquent, nous proposons un cadre pour réaliser cette approche innovante sur des ensembles de modules quasi-sphériques arrangés en structures cristallines cubiques à faces centrées, et présentons des algorithmes permettant d'implémenter l'autoreconfiguration dans ce contexte. Nous analysons ces algorithmes et les évaluons sur des cas de construction de formes de complexité croissante, afin de montrer que notre méthode permet d'arriver à des durées de reconfiguration jusqu'ici inatteignables.

Mots-clés : Matière Programmable, Algorithmes Distribués, Robotique Modulaire, Autoreconfiguration, Systèmes Multi-agents

LIST OF ABBREVIATIONS

API: Application Programming Interface	MEMS: MicroElectro-Mechanical-Systems
CA: Cellular Automata	M RTP: Modular Robot Time Protocol
CAD: Computer-Aided Design	MSR: Modular Self-reconfigurable Robot
CR: COORDINATOR_READY	OOP: Object-Oriented Programming
CSG: Constructive Solid Geometry	PGP: PROVIDE_GOAL_POSITION
DES: Discrete-Event Simulation	PLS: PROBE_LIGHT_STATE
DOF: Degree Of Freedom	PM: Programmable Matter
EPL: Entry Point Location (of a scaffold tile)	R: Root (scaffold tile component)
FCC: Face-Centered Cubic (lattice)	RGP: REQUEST_GOAL_POSITION
FTR: FINAL_TARGET_REACHED	SC: Square Cubic (lattice)
GLO: GREEN_LIGHT_ON	SOPS: Self-Organizing Particle Systems
GUI: Graphical User Interface	SR: Self-Reconfiguration
HUD: Heads-Up Display	STL: STereoLithography
IBR: INGOING_BRANCH_READY	TCF: TILE_CONSTRUCTION_FINISHED
IF: INITIATE_FEEDING	TIR: TILE_INSERTION_READY
IoT: Internet of Things	UCM: Unit-Compressible Modules
IoE: Internet of Everything	
MDP: Markov Decision Process	

CONTENTS

Introduction	3
Contributions	13
Dissertation Outline	15
 I Context and State of the Art	 17
 1 Programmable Matter	 19
1.1 The Programmable Matter Project	19
1.2 Modular Robotic Model	21
1.2.1 The <i>3D Catom</i> Modular Micro-Robot	21
1.2.2 Programming Model and Assumptions	27
1.2.3 First Hardware Prototype	28
1.3 Dedicated Simulation Framework	29
1.3.1 Related Simulation Platforms	30
1.3.2 VisibleSim	32
 2 State of the Art of Self-Reconfiguration in 3D Lattices	 45
2.1 Classification of Self-Reconfiguration Approaches	46
2.1.1 Bottom-Up Approach	48
2.1.2 Top-Down Approach	51
2.1.3 Theoretical Approach	59
2.2 Analysis of 3D Lattice Self-Reconfiguration Algorithms	62
2.2.1 Planning Under Mechanical Constraints	64
2.2.2 Free-Space Requirements and Obstacles	64
2.2.3 Collision and Deadlock Prevention Mechanisms	65

2.2.4	Goal-Shape Representation	66
2.2.5	Solution Methods	67
2.2.6	Surface Movements vs. Internal Movements	67
2.2.7	Motion Parallelism and Convergence	68
2.2.8	On the Complexity of Self-Reconfiguration	69
2.2.9	Simulation Environments	69
2.2.10	Evaluation Methods	70
2.2.11	Validation Methods and Analyses	71
2.3	Discussion on Programmable Matter	72
2.3.1	Self-Reconfiguration Criteria	72
2.3.2	Relevance of Existing Works	73
2.3.3	Perspectives	74
II	Contribution	77
3	Introduction to Engineering Faster Self-Reconfiguration	79
3.1	Scaffolding and Structural Engineering	86
3.2	A Dedicated Self-Reconfiguration Platform	88
3.3	Visual Aspect Preservation Through Coating	89
4	Sandbox and Scaffold-based Self-reconfiguration Algorithms	93
4.1	Fundamentals	95
4.1.1	Scaffold Construction Principles	95
4.1.2	High-level Planning: Tile Construction Scheduling	99
4.1.3	Low-level Planning: Module Navigation	105
4.1.4	Motion Coordination Algorithm	107
4.2	Building Simple Pyramids	109
4.2.1	Motivations	110
4.2.2	Assumptions	110
4.2.3	Self-Reconfiguration	111
4.2.4	Analysis	112

4.2.5	Simulations	118
4.3	Semi-Convex Generalization	121
4.3.1	Motivations and Challenges	122
4.3.2	Updated Model and Assumptions	123
4.3.3	Analyses	124
4.3.4	Simulations	127
4.4	Generalization	133
4.4.1	Motivations and Challenges	134
4.4.2	Updated Assumptions	134
4.4.3	Main Idea	135
4.5	Discussion	136
5	A Simple Coating Assembly Algorithm	139
5.1	Coating Self-Assembly	141
5.1.1	High-Level Assembly Strategy (Bottom-Up Layering)	141
5.1.2	Standard Layer Assembly Strategy (The Tucci Algorithm)	142
5.1.3	Support Layer Assembly Strategy (Border Completion)	143
5.2	Results	146
5.2.1	Preservation of message complexity	146
5.2.2	Simulations	147
5.2.3	Complexity	149
5.3	Discussion	150
5.3.1	Limits of the Current Method	150
5.3.2	Discarded Coating Strategies	152
5.3.3	Module Dispatch From the Sandbox	153
5.3.4	Towards More Efficient Coating Methods	153
	Conclusion	157
	Summary of the PhD thesis	158
	Discussion	161
	Perspectives	167

INTRODUCTION

INTRODUCTION

What makes humans such fantastic animals is their ability to hold complex concepts in their mind and share them to other members of their species. This makes human societies more intelligent than the sum of their parts and this collective intelligence has been the driving force of human prosperity, growth, and flourishing through the ages. Yet, understanding and entertaining complex concepts in one's mind is infinitely easier than sharing them with one's kin. To that purpose, humanity has invented languages, a descriptive tool that can spread thoughts and ideas from mouths to ears (through talking), and later from hands to eyes (through writing). However, preserving the integrity and accuracy of an idea or concept described by language is a nearly impossible task. Another way humans have managed to communicate the content of their “teetering bulbs of dread and dreams”, as poet Russel Edson would call brains, is through the use of what we will name *display systems*. Display systems encompass all the ways that human ingenuity has conceived to visually represent some information.

One of the first recorded display systems is perhaps paintings, or rather cave paintings, as those of the Chauvet cave in France, dated to earlier than 30 000 BCE. It is likely that human prosperity has been enabled by increasingly complex and powerful display systems, supporting human collaboration by helping individuals express concepts with increasing accuracy. Yet, even though humans live in a *three-dimensional* (3D) world, most display systems through human history have represented the world in *two dimensions* (2D). For instance, drawings, paintings, photography, and even screens are all attempts to capture our 3D world on a 2D surface. Unfortunately, this means that a substantial amount of information is lost along the way. 3D display systems have nevertheless existed for the most part of human history, but considerable skill and effort were required to represent the world through them with high *fidelity*. Their production has thus been reserved to an elite. Wood, stone, and metal carvings and sculptures are instances of such high effort ways of representing the world. However, beside dimensional data, not all display systems have been able to represent the world at the same level of fidelity. Another interesting aspect of display systems is that even though they all provide visual information, some are also *multi-sensory*. Sculptures can be touched for instance, which provides additional tactile information and sensations. Furthermore, some are *static*, in the sense that the information that they represent or communicate never changes (e.g., paintings, sculptures, drawings, 3D-printed objects), while others are dynamic and hence their content can evolve (e.g., screens). Finally, some display systems are *interactive*, and provide

users with a way to modify the displayed data (e.g., screens, especially touchscreens), while others are not.



Figure 1: **Some instances of ancient, modern, and future display systems (from left to right):** cave paintings; sculpture; touchscreen; holographic interface from the movie *Avatar*; programmable matter display system from the Claytronics (Goldstein et al., 2004) project

For the most part of human history, display systems have remained static, two-dimensional, and lacked interactivity. Nonetheless, recent advances in technology have given us much more powerful systems, now high fidelity, low effort, and multi-sensory.

Even the display systems of the far and near future have existed for some time in the collective imagination of our species, and have been featured in numerous speculative science-fiction books, TV shows, movies, and video games. Let's consider holography and holograms for instance, whose first reference probably dates back to Isaac Asimov's *Foundation Trilogy*, but which have since then become commonplace thanks to cinematic successes such as *Star Wars*, *Star Trek's Holodeck*, the artificial intelligence Cortana in the *Halo* game franchise, or the holographic consoles and Heads-Up Displays (HUD) in the movie *Avatar*. Holographic systems promised a new era of three-dimensional display systems, where the displayed data fully integrates with its environment, can be inspected from any angle, and which is fully embodied, even though it remains matterless and impalpable. However, existing holographic technologies are still a long way from the ubiquity, fidelity, and real-time performance found in science fiction, as alternative technologies such as virtual reality and augmented reality have proven technically easier to implement and more versatile, despite their dependence on wearable devices. All of those, however, lack a particularly stimulating property: *interactivity*.

In our work, we envision a different class of future display systems, which would be three-dimensional, dynamic, and tangible, as an application of *Programmable Matter (PM)*. Programmable matter is usually defined as matter that is able to dynamically alter its state (such as its shape, color, density, conductivity, etc.) in a programmable fashion, as a response to internal or external stimuli either from user interaction or sensed events.

The term has been first coined by (Toffoli et al., 1991), and then largely popularized by promising flagship projects such as the Claytronics project (Goldstein et al., 2004, 2005), now discontinued. In its most advanced representations, programmable matter consists of artificial atoms that can be rearranged at will, thus ushering a new era of synthetic reality where humans have finally achieved their ultimate quest to master their environment and any object in the environment can morph into any other desired object. Such an extreme version of programmable matter surely appears far-fetched, but our current technological trends certainly tend towards programmable environments, with the rise of the Internet of Things (IoT), or even the Internet of Everything (IoE). In our case, we aim to achieve a programmable matter system that can represent any three-dimensional object or scene with high fidelity (3D visual display), alter and update the displayed data (dynamism), and offer multi-sensory tactile interactions (tangibility and interactivity). The displayed data could hence be manipulated and altered by touch, and the changes could be reflected into the underlying data model. Such technology would be somewhat akin a tangible version of the holographic displays envisioned in the aforementioned science fiction work. Such system has also been proposed by Goldstein et al. (2009), who imagined the ultimate communication technology, named *pario*, that went beyond audio and video. The idea was to capture in real time the audio and video data of a person and send that data over the internet to a programmable matter system in a distant location that would recreate a 3D clone the captured person — enabling real-time tangible 3D *pario*-conferencing. While we are doubtful that such real-time performance and accuracy will ever be attained, the idea to physically instantiate anything that can be modeled on a computer seems powerful enough for exciting applications in areas such as education and training, entertainment, or interactive computer-assisted design.

In parallel with the innovation in display systems, the human condition has been completely transformed for the better thanks to the mechanization of society and the workforce, reducing suffering by allowing humans to move away from the fields and food production, to which most human lives were unfortunately confined since the beginning of agriculture. This has contributed to a reduction in human suffering caused by a short life of (hard, physical) work, and to powering a cultural, intellectual and artistic revolution as humans could now focus let the machines do the work and open to new areas of life. Robotics has been one of the main drivers of this mechanization in recent decades, opening entirely processes to mechanization, by enabling the programmability of machines and instilling them with specialized intelligence. Robots were until recently thought as big, monolithic entities, but advances in miniaturization, control, and integration have enabled a new generation of robots to emerge that are modular, reconfigurable, potentially lightweight, and thus, versatile.

Though many technologies could potentially be used as the basis for programmable matter, building programmable matter using Modular Self-reconfigurable Robots (MSR) rep-

resents the most promising endeavor, as it is the only technology that manifests all four desired properties: **evolutivity**, **programmability**, **autonomy**, and **interactivity** (Bourgeois et al., 2016). Alternative forms of programmable matter include origami-style foldable materials Hawkes et al. (2010), 4D printing technologies (Tibbits et al., 2014), and DNA machines (Ke et al., 2012; Kim et al., 2011). Modular robotic-based programmable matter therefore seeks to use robotic *modules* in place of atoms as the basic structure of matter, which have benefit of being easily programmable and producible (though their scale is enormously greater than the one of atoms). Such programmable matter systems are also increasingly represented in the popular culture: the shape-shifting T-1000 android in the movie Terminator, reconfigurable furniture and walls made from nano-robot in the TV shows Doctor Who and Altered Carbon, or, with even more semblance to actual research, the nano-robots swarm of the child robotics prodigy Hiro in Disney's Big Hero 6.

Modular robots are robotic systems composed of interconnected electromechanical modules that can rearrange in order to best adapt to their task-environment or recover from failures (Stoy et al., 2010). Individual units can have various levels of autonomy and have to coordinate through sensing and communication to achieve their tasks. Their promise is to realize robotic systems that are more versatile, affordable, and robust than their conventional monolithic counterparts, at the cost of a probable reduced efficacy for extremely specific tasks. Though many modular robots have been realized in hardware and their capabilities demonstrated, Stoy et al. (2011) argued in 2011 that no existing modular robots are actually self-reconfigurable, physical, distributed, and autonomous, as desired — this does not seem to have changed since. This concept was first introduced in the late 1980s as *cellular robotic systems* by T. Fukuda, later physically realized in the CE-BOT modular robot by Fukuda et al. (1990). Since then, the field has been renamed *modular robotics*, and various robotic architectures have been proposed (Ahmadzadeh et al., 2016; Tan et al., 2020). Modular robots differ from traditional swarm robotic systems by the fact that all individual robots (usually referred to as *modules*) in a system must remain connected to each other at all times, as they commonly rely on their interconnection for communication and power distribution whereas swarm robots are usually mobile robots with full motion and power autonomy — albeit some of these systems are sometimes referred to as *mobile modular robotic systems* (e.g., Kilobot (Rubenstein et al., 2012), e-puck (Mondada et al., 2009)). Furthermore, if modular robotic systems can be seen as tightly-linked swarm robotic systems, this depends on their mode of control, as swarm intelligence only emerges from the distributed and local interactions of its simple computational devices (Hamann, 2018; Hamann et al., 2007; Ijspeert et al., 2001).

The first type of architecture is *chain-type* modular robots, where chains of modules make up the structure of the robots, with the modules always arranged in a tree-like fashion. Modules therefore act as joint or links, with few degrees of freedom (DOF), and create a

robot with high DOF by assembling with each other. Figure 2 shows a selection of notable chain-type modular robots from the robotics literature.

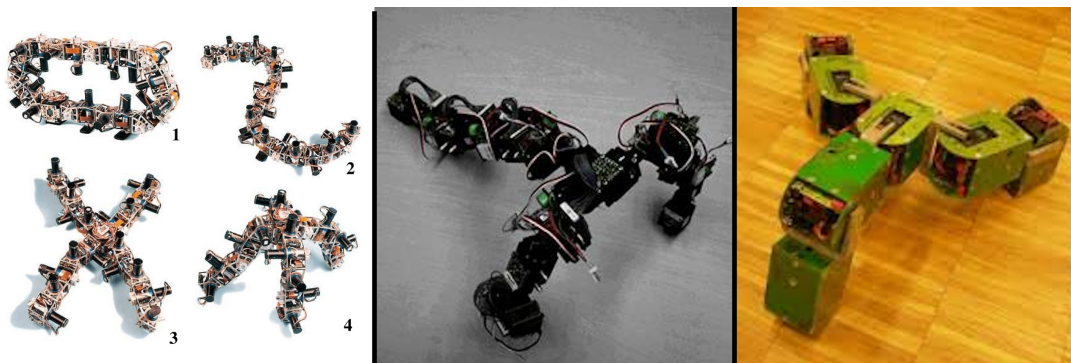


Figure 2: **Examples of chain modular robots:** (Left) Polybot (Yim et al., 2000) self-reconfigurable robot in various chain configurations. (Center) Tripod configuration of CONRO (Castano et al., 2000) modules. (Right) Five YaMoR modules (Moeckel et al., 2006) in a tripod configuration.

The other main architecture type is *lattice-type* modular robots, composed of an ordered arrangement of modules, residing on a regular structure named a *lattice*. Lattice modular robots are easier to model in software as they reside in a discrete grid, and they can be more easily controlled in parallel than chain modular robots. There exist a number of lattice structures, each based on a particular geometry of their cells, and differing by their packing density, number of dimensions, and number of neighboring positions at a given location (Naz et al., 2018; Piranda et al., 2018). Figure 3 introduces various modular robots from various lattice geometries. This will be the modular robotic architecture under consideration in this manuscript, where the state of the art in the self-reconfiguration of 3D lattice-based metamorphic systems will be brought into focus.

Finally, some modular robots can both exhibit the features of lattice-type and chain-type modular robots, and are commonly referred to as *hybrid*. Several such systems are shown in Figure 4. There are arguably few exceptions of modular robots that do not exactly fit any these architectures, but this is out of the scope of this manuscript (Ahmadzadeh et al., 2015), like the FireAnt system (Swissler et al., 2018).

In this thesis, we will focus exclusively on modular robotic systems where modules are endowed with self-locomotion capabilities, nor does it address modular mechatronic artifacts such as the Pebbles Gilpin et al. (2010), Programmable Parts Bishop et al. (2005), or origami robots Miyashita et al. (2017), where motion is provided by external agitation and control on the latching. We are therefore not interested in the very large literature on self-assembly where modules do not allow full motion controllability and rely on external actuation such as Haghighat et al. (2017). However, as opposed to self-locomoted modular robotic systems where most hardware implementations are in the decimeter range, all these works went down to concrete physical implementations in the millimeter-centimeter

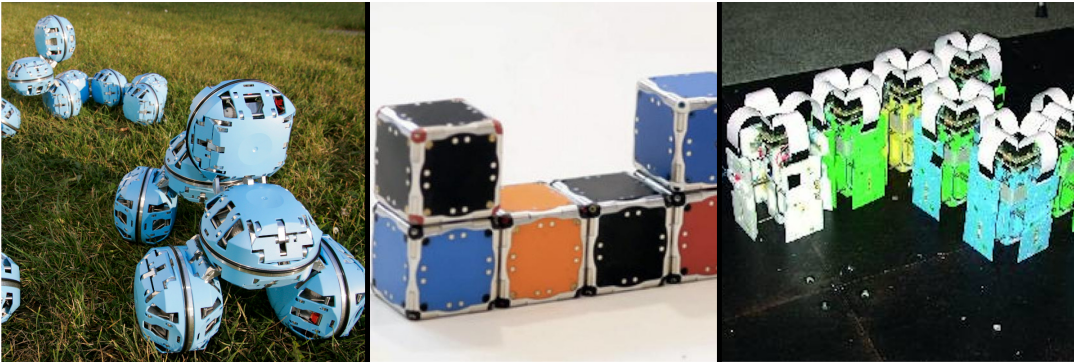


Figure 3: **Examples of lattice modular robots:** (Left) ATRON (Jorgensen et al., 2004) modular robots in different configurations. (Center) Rotating M-Block (Romanishin et al., 2013) modules. (Right) 2D Crystalline (Rus et al., 2000) modular robots with extensible arms.

range, much closer to what we are aiming for.

Modular robotic systems have been considered for a wide range of applications (Ahmadzadeh et al., 2016), such as search and rescue, exploration, inspection, mapping, or adaptive tool sets. These applications require various behaviors from modular robotic systems such as shape formation, locomotion (Hamann et al., 2010), manipulation (Martinioli et al., 2004), balancing, and supporting. Then, these behaviors can be implemented thanks to operational primitives such as gait, grasping, self-repair, self-reconfiguration, self-assembly, self-adaptation, collective action, etc.

While the design of modular robotics hardware poses considerable challenges, implementing these operations and realizing these behaviors in software is no easier task. Indeed, researchers seek to achieve autonomy and decentralization in modular robotic systems, which means that desired behaviors must *emerge* from the collective actuation of constituent modules rather than from a centralized control of the system. Intelligence must therefore be distributed across the system, and modules must resort to communication to coordinate and commonly achieve a task. This requirement stems from the fact that, if all modules are identical and independent, they can easily but added, discarded, and replaced without any additional programming, leading to increased scalability and robustness.

Among all the operations that have been mentioned, the most striking feature of modular robotic systems is, therefore, their ability to change their morphology on-the-go, a process named *self-reconfiguration* (*SR*). The self-reconfiguration problem is usually defined as finding a sequence of individual module motions that transforms an *initial* configuration \mathcal{I} of a modular robot into a *goal* configuration \mathcal{G} . A *configuration* is a particular arrangement of the modules in a modular robotic system. It can be modeled mathematically as a graph $G(E, V)$, where V represents the modules of the system, and E their interconnection

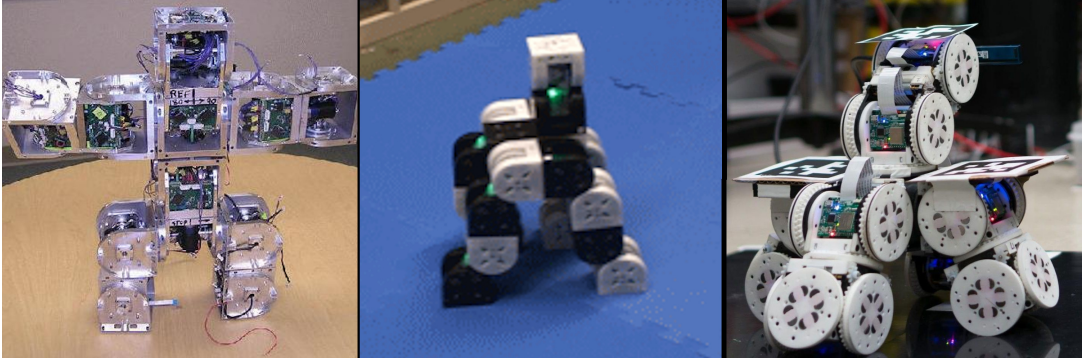


Figure 4: **Examples of hybrid modular robots:** (Left) Superbot (Salemi et al., 2006) modular robot in a humanoid configuration. (Center) M-TRAN (Kamimura et al., 2002) modular robot in 4-legged configuration. (Right) SMORES (Davey et al., 2012) modular robot undergoing self-reconfiguration.

through their connectors. In this manuscript, we will distinguish self-reconfiguration from *self-assembly*, which we will define as follows: in modular robotic self-assembly, the goal is to find a construction sequence for building a shape made of modular robotic modules without regard to the motions of the modules that self-reconfiguration considers. Furthermore, the structures that we are aiming to assemble are very well defined, and accuracy is non-negotiable in this context — unlike in works where the purpose of self-assembly is to react to environmental change (Kernbach et al., 2009), or where the target shape can afford a small margin of error.

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 et al., 1991). The exploration space for reconfiguration between two random configurations is therefore exponential in the number of modules, which prevents complete optimal planning to be found for all but the simplest configurations.

Self-reconfiguration planning is thus a hard problem, as traditional search methods are ineffective due to the size of the configuration space increasing exponentially with the number of modules in the system. It has been proved to be NP-complete for chain-type MSR by reduction to 3-PARTITION (Hou et al., 2010) and Probabilistic SATisfiability (PSAT) (Gorbenko et al., 2012), and is also suspected to be NP-complete for lattice MSR.

Not only is self-reconfiguration computationally intractable when computing it in a centralized fashion, but as mentioned, modular robotic systems must avoid relying on cen-

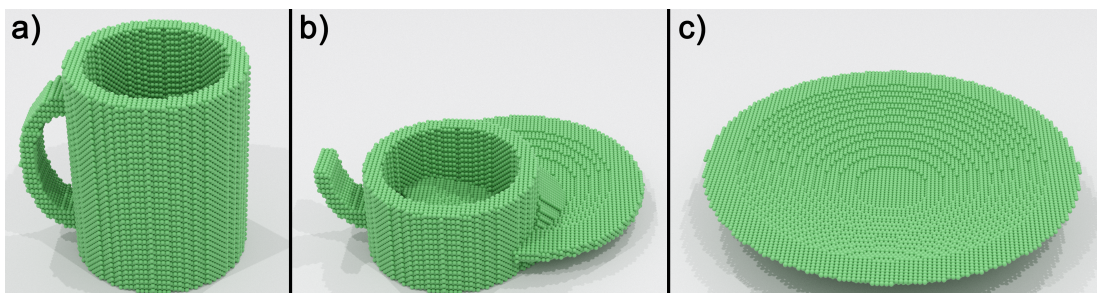


Figure 5: Sample self-reconfiguration of about 38,500 3D Catom modules from a cup into a plate. **(a)** *Cup* initial configuration; **(b)** An intermediate configuration from the self-reconfiguration process; **(c)** *Plate* goal configuration.

tralized computing and use a distributed paradigm instead. The difficulty of distributed self-reconfiguration algorithms then also comes from the incredible level of coordination required for many modules to move in parallel while avoiding collisions between each other, and creating *deadlocks* during the construction process, situations where part of the construction cannot be realized because some modules are blocking part of the goal shape, perhaps because of an erroneous construction scheduling.

In fact, a self-reconfiguration algorithm could well proceed without any parallel motion of the modules, but the critical parameter that must be optimized in self-reconfiguration is *reconfiguration time*. That is the time to transform the initial configuration \mathcal{I} into the goal configuration \mathcal{G} . This reconfiguration time is usually expressed in number of discrete time steps, and as a function of the number of modules in the configurations. It is evidently much slower to perform the transformation one module at a time than with parallel motions, and sequential algorithms are hence mostly impractical due to their slow speed. Other metrics for self-reconfiguration algorithms include the total number of motions performed by the modules, the memory cost of the algorithm, or the number of messages exchanged.

Several research communities have become interested in the self-reconfiguration problem, with very different viewpoints. On the one end, roboticists seek to design algorithms to implement specific behaviors in their hardware systems, on the other, theoretical computer scientists in study this problem from the standpoint of computational geometry (Michail et al., 2017), mostly on 2D problems until now, and finally, other research communities such as in distributed systems study self-reconfiguration algorithms without necessarily being involved in hardware research. We mostly belong to the latter, even though as we will see we are now making progress in the design and manufacturing of hardware micro-scale modular robots.

On the robotics side, 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 et al., 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 based on 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 et al., 2013)—based on a Markov decision process to optimize the number of connections/disconnections of modules—have proven very promising as they can easily be 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).

The self-reconfiguration problem evidently stands as one of the major foundational issues of programmable matter. In this context, algorithmic solutions are required to exhibit some particular properties that will be discussed further on through an analysis of the state of the art of self-reconfiguration juxtaposed to the expectations of programmable matter. Returning to the objective of this research, let us introduce what we believe are desirable properties of a modular robotic system for object and 3D data representation with programmable matter, though this will be discussed in more details further on. First, we are interested in large-scale robotic ensembles, constituting hundreds to thousands of modules, at the millimeter scale or lower. This is due to the resolution of our represented objects, which is a function of the size of the modules and their number. Consequently, reconfiguration speed is absolutely critical in our application, since large-scale systems might have much more matter (in the form of modules) that must be moved during reconfiguration, and thus require very high reconfiguration speeds to remain practical. Furthermore, a display system where a user has to wait 10 minutes for its data to load seems unlikely to gain a lot of attention in our increasingly fast-paced world. This therefore leads

to the following thesis, which will be defended in this manuscript:

Thesis Statement

In this thesis, I defend the idea that there exist ways to accelerate self-reconfiguration of programmable matter systems, and that a new class of reconfiguration methods with increased speed and specifically tailored for this purpose must emerge. I contend that such methods can be achieved by proposing a novel way of representing programmable matter objects, and using a dedicated reconfiguration platform.

To solve the reconfiguration 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, based on previous research (Kotay et al., 2000; Ünsal et al., 2001a; Støy et al., 2007; Støy, 2006; Lengiewicz et al., 2019) 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 to restore 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. Furthermore, while traditional self-reconfiguration research assumes systems where all the number of modules in the initial and goal configurations are equal, we drop this constraint and assume that our self-reconfigurations take place in a dedicated environment named a *sandbox*, that acts as a reserve of modules placed underneath the reconfiguration scene and that can supply and discard modules from the reconfiguring ensemble. We introduce in this manuscript various algorithms to establish the foundations of this new self-reconfiguration paradigm, and show simulation results supporting our thesis.

As an anecdotal example of reconfiguration speeds, an estimate of the reconfiguration time for our sliding blocks (Piranda et al., 2013) and using the metrics introduced in (Zhu et al., 2019) is 12 h for 800 blocks. This is just a rough estimate to stress that time and parallelism is really an issue in self-reconfiguration algorithms. Besides, it was shown in practice that it could take 11.66 h to reconfigure a 2D ensemble of 1 000 Kilobots (Rubenstein et al., 2012) — an extraordinary result by itself due to the very large number of modules involved in this real-world experiment. Given a rotation time of 20 ms, as gauged by our latest hardware experiments with our quasi-spherical rotating 2 mm *3D Catom* modules, 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. Such a tremendous re-configuration time decrease can seem highly suspicious. This is indeed more of an order of magnitude than a precise result, as reconfiguration time also has to factor in the time spent processing and communicating by the modules, control loops, synchronization, etc. This figure is thus most likely a very low approximation of expected self-reconfiguration time. Nonetheless, even if the actual duration of ends up $\times 50$ times greater, the total reconfiguration time for more than a thousand modules would still be a matter of minutes, not hours. This hints that together, electrostatic actuation and the reconfiguration method we propose can dramatically reduce self-reconfiguration time, enabling practical applications for programmable matter.

CONTRIBUTIONS

In this thesis, we first propose a classification of 3D self-reconfiguration algorithms for lattice modular robots based on their relationship to their underlying hardware model. Then, we introduce a new paradigm of self-reconfiguration, based on the combination of scaffolding techniques, a dedicated platform for self-reconfiguration with variable cardinality, and a coating routine. We show that this approach is essentially a trade-off where flexibility is sacrificed (due to being confined to a platform) to achieve unprecedented reconfiguration speeds.

- Classification of 3D Lattice Self-reconfiguration Methods:** While researchers envision exciting applications for metamorphic systems like programmable matter, current solutions to the shape formation problem are still a long way from meeting their requirements. To dive deeper into this issue, we propose an extensive survey of the current state of the art of self-reconfiguration algorithms and underlying models in modular robotic and self-organizing particle systems. We identify three approaches for solving this problem and we compare the different solutions using a synoptic graphical representation. These approaches are named *Top-Down*, *Bottom-Up*, and *Theoretical*, and express the relationship of the reconfiguration software with regard to its hardware model. More specifically, we find that *Theoretical* works abstract away all the kinematic and communication constraints of modules, and seek only to determine the minimum requirements of the model to achieve various self-reconfiguration performance. *Bottom-Up* works are at the opposite end, starting from a very concrete physical hardware and seeking the optimal self-reconfiguration planning for this model. Finally, the *Top-Down* approach stands halfway in between and consists in working with relatively abstract models that do not necessarily exist in hardware or instead synthesize several possible hardware architectures. Their kinematic constraints are somewhat laxer than

Bottom-Up works. We thoroughly discuss existing *Top-Down* works and confront existing methods to our vision of programmable matter. We derive from this comparison and discussion a number of future research directions for getting closer to practical modular-robot-based programmable matter systems.

- **The Sandbox and Scaffold Self-Reconfiguration Paradigm:**

- ◇ Based on the current state of the art of self-reconfiguration and the very restrictive motion constraints of the *3D Catom* model, we propose to alter the traditional self-reconfiguration model in the following ways:
 1. Change the way an object is defined to its boundary representation, i.e., the internal structure of an object does not matter as long as its external aspect remains unchanged.
 2. Consequently, propose to build a sort of skeleton, or *scaffolding*, of the internal structure of objects, as inspired by previous work by Kotay et al. (2000); Ünsal et al. (2001a); Støy (2006); Støy et al. (2007); Lengiewicz et al. (2019). We therefore propose an innovative scaffold design that is parameterizable and has a face-centered-cubic lattice structure made from our rotating *3D Catom* micro-modules. It is built from a regular arrangement of similar multi-module structures named *tiles*, that can be built deterministically in constant time. These tiles enable the *pipelining* of modules inside the shape, easing coordination between modules and allowing for very high motion parallelism.
 3. Furthermore, we contend that the external aspect of the scaffolded object can be preserved by the addition of a thin layer of *coating* made from the same *3D Catom* modules. We therefore introduce the *coating problem* of modular robotics for building this modular robotic surface.
 4. Assume the presence of a reserve of modules underneath the reconfiguration scene, named a *sandbox*, and that is able to supply modules at multiple ground locations regularly placed at the base of the scene. Consequently, assume that *anisonumeric* reconfigurations can occur, that is self-reconfiguration between shapes with different cardinality.
- ◇ 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. 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. Our analyses of the method and simulations on shapes with an increasing level of intricacies 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 extended to any shape and improved further.

- ◇ We introduce a basic method for constructing the coating of a class of scaffolds layer by layer. This method is based on previous work on self-assembly by Tucci et al. (2018) and is partly inspired by theoretical work on the Amoebot model by Derakhshandeh et al. (2017). It suffers, however, from a clear lack of parallelism compared to our scaffold assembly, which is why we hint at various alternative strategies and improvements for extending and improving this work. We also show that even with a straightforward algorithm, our scaffolding and coating combo uses much fewer modules and can achieve a linear reconfiguration time in the number of modules.
- ◇ Thanks to all the aforementioned contributions, we establish the foundations of scaffold-to-scaffold self-reconfiguration, and enunciate that with our framework, scaffold-to-scaffold self-reconfiguration can be achieved in sublinear time and shape-to-shape self-reconfiguration in linear time for our class of shapes, even with our current severely limited coating method.

DISSERTATION OUTLINE

This dissertation is organized as follows: In Chapter 1, we introduce the context and basis of our work, namely the Programmable Matter Consortium to which it is integral, and the *3D Catom* hardware model that it is based on. We also touch on *VisibleSim*, our dedicated simulator for our modular robotic system research, to the development of which the Author has significantly contributed, and that has supported all experiments discussed in this dissertation. Then, in Chapter 2 we propose a survey and discussion of the state of the art of self-reconfiguration algorithms and methods, focusing for the most part on 3D lattice-based self-reconfiguration works. From our conclusion drawn from the two preceding chapters, Chapter 3 proposes an innovative approach to transcend the current limits of previous work and our model, and achieve faster self-reconfiguration speeds at a reasonable cost. Then, Chapters 4, and 5 introduces various algorithmic primitives to implement this approach on the *3D Catom* model, and present various experiments to quantify the current performance and gain of the methods. In closing, this dissertation concludes on a critical discussion of our approach and model, seeking to derive additional guidelines and insights from this work, before compiling a list of directions and perspectives for future work and guiding advances in the field of self-reconfigurable robotics.

I

CONTEXT AND STATE OF THE ART

PROGRAMMABLE MATTER

Contents

1.1 The Programmable Matter Project	19
1.2 Modular Robotic Model	21
1.2.1 The <i>3D Catom</i> Modular Micro-Robot	21
1.2.2 Programming Model and Assumptions	27
1.2.3 First Hardware Prototype	28
1.3 Dedicated Simulation Framework	29
1.3.1 Related Simulation Platforms	30
1.3.2 VisibleSim	32

1.1/ THE PROGRAMMABLE MATTER PROJECT

The work presented in this manuscript is part of a much larger effort to realize the concept of programmable matter, as presented in the introduction. A consortium around this effort has been built under the name of *The Programmable Matter Consortium*, and has rallied numerous partners from major research institutions (see Figure 1.1), industrial leaders and manufacturers, and even art studios tasked with exploring the concept from their perspective.

This partnership goes beyond the mere financial support of research efforts through research grants, as it enables frequent real-world interactions and collaboration between partners. Within the past three years, and while researching the work synthesized in this manuscript, I have been taken part in several visits from our team to work hand in hand with partners and mutualize our knowledge: attempts to sketch and then design our first joint hardware prototypes while visiting Prof. David Blaauw's team at the University of Michigan (with Prof. Yoshio Mita of the University of Tokyo joining the discussions through late-night videoconferencing from half a world away); or discussing the algorithmics and complexity theory of self-reconfiguration with Prof. Othon Michail's team in

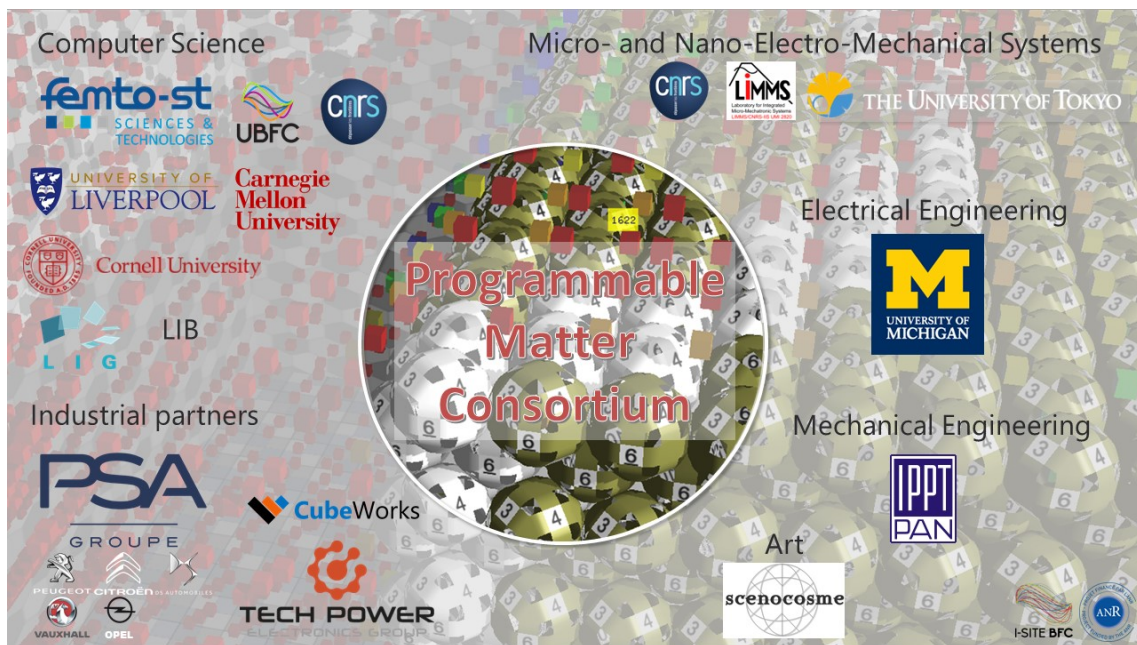


Figure 1.1: The *Programmable Matter Consortium* partners

Liverpool. The Dagstuhl seminars on the algorithmic foundations of programmable matter (that took place in 2016 (Fekete et al., 2016) and 2018 (Berman et al., 2019), and to which I have participated in 2018), have also greatly contributed to the inception of the project. During these seminars, researchers working on various flavors and levels of programmable matter get to meet and exchange ideas for a few days in a former medieval convent now turned into an authentic “computer science monastery.” The value of this partnership is, therefore, the open and global exchange of expertise between research teams with diverse and complementary specialties, each capable of addressing some of the many challenges standing between the current state of technology and the real-world programmable matter of our future.

Our consortium is involved in research on both software and hardware aspects of the technology. On the software side, the objectives of the consortium are to establish strong theoretical and algorithmic foundations (synchronization, leader election, distributed coordination, etc.) for programmable matter systems, and further our understanding of the capabilities of metamorphic self-reconfiguring systems—which is where the topic of this work fits.

On the hardware side, the current main focus of this joint enterprise is to push against the current limits of miniaturization of modular robotic systems and start a new era of systems at the micro-millimeter scale. This requires major innovations in several areas of research from the fields of micro- and nano-electro-mechanical systems and electrical engineering, such as computer miniaturization and integration, electrostatic actuation, nanoscale 3D printing technologies, etc. The robotic architecture under development is also the one

that is taken into consideration in this thesis: the *3D Catom*.

If we believe that miniaturization is such an important aspect of modular-robot-based programmable matter, it is because the size of the modules has such a tremendous impact on the fidelity of programmable matter object with regard to their inert counterpart. To better illustrate the influence of the geometry and size of the modules, Figure 1.2 shows a visual comparison of programmable matter cups made from cubic and spherical modules at various sizes.

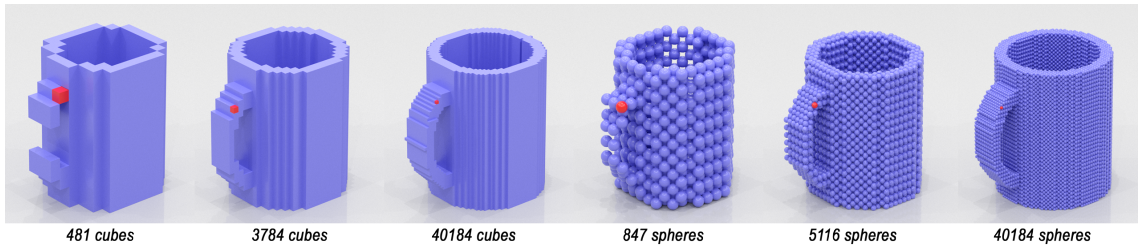


Figure 1.2: Visual comparison of cups made of modular-robot-based programmable matter with cubic and spherical modules and at various resolutions in terms of the size of the modules.

1.2/ MODULAR ROBOTIC MODEL

In this section, we introduce the model that stands as the basis of this work, both in terms of the robotic and computing model. The exact characteristics of the envisioned *3D Catom*, its underlying theoretical model, and its current state of hardware development, are thoroughly discussed.

1.2.1/ THE *3D Catom* MODULAR MICRO-ROBOT

As it has just been pointed out, the distributed algorithms presented in the following chapters consider the self-reconfiguration of modular robots named **3D Catoms**. We will thereafter refer to a single unit from this modular robot as a *3D Catom* module, or simply as a **module**. *3D Catoms* are quasi-spherical rotating modules, with an expected diameter in the micro-millimeter range. These modules can attach to each other and rotate around one another without moving parts through electrostatic actuation.

Geometry *3D Catoms* have a quasi-spherical geometry which consists of 12 flat squares (the connectors, drawn in red in Figure 1.3a, 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) lattice** (cf. (Piranda et al., 2018) for relative contact points

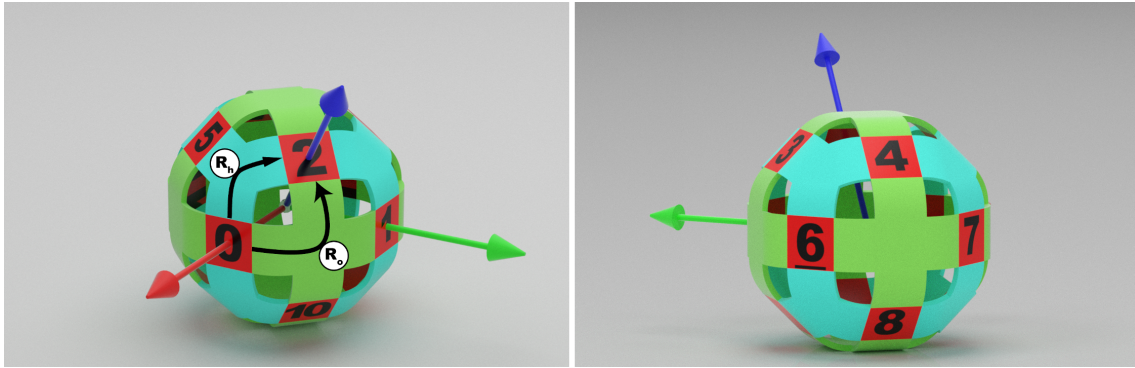


Figure 1.3: **The 3D Catom:** geometry from two opposite angles, skewed coordinate system, and the two possible paths for the motion of a neighbor on its surface, using and a hexagonal actuator (R_h) or an octagonal actuator (R_o).

coordinates). There are two kinds of curves that 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 1.3a) 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 1.3a) is made of a square and 4 sections of the body of a cylinder.

3D Catoms assemble by latching onto each other using one of the 12 electrostatic connectors (also, interfaces) on their surface (numbered from #0 to #11 for identification purpose), and form FCC lattice structures—see Figure 1.4 for the positions of all 12 neighbors (modules directly connected to a given module) of a *3D Catom*, across three layers. Each position within a *3D Catom* lattice can also be referred to as a *lattice cell* (or simply *cell*), and is assigned a unique discrete coordinate. We assume that modules can only sense the presence or absence of their immediate neighbors, through their interfaces.

Furthermore, as can be seen in Figure 1.4, modules on adjacent horizontal layers of modules are staggered. For that reason, different coordinate systems can be pertinent depending on the task at hand. We choose to use a coordinate system with a skewed \vec{z} axis (cf. Figure 1.3)—as it circumvents the trouble of a straight \vec{z} axis caused by having different top and bottom neighbor positions depending on the parity of the current horizontal layer.

Rotations *3D Catoms* move around the FCC grid by rotating on the surface of the surrounding modules, using their orthogonal or hexagonal actuators. 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*. Note that a module acting as a pivot is not allowed to perform a motion while it is actuating for another module, and that a moving module cannot carry another during its motion.

Figure 1.5 shows a simple rotation from one connector to another on a neighbor pivot,

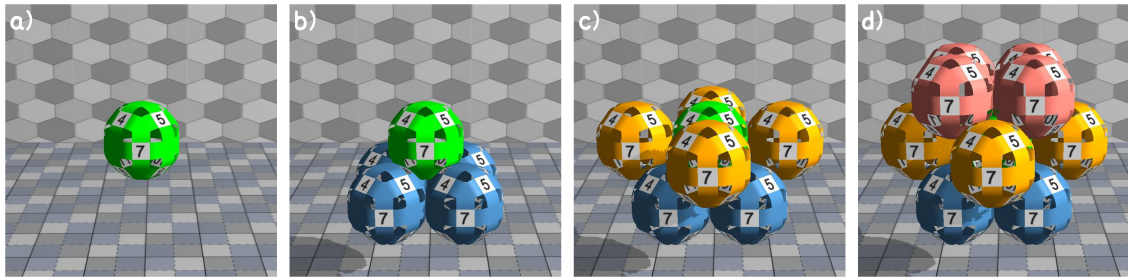


Figure 1.4: Arrangement of a 1-ball of *3D Catoms* in a Face-Centered Cubic (FCC) Lattice: **(a)** A single *3D Catom* at the center of the ball; **(b)** The four bottom neighbors of the center; **(c)** The four horizontal neighbors of the center; **(d)** The four top neighbors of the center module.



Figure 1.5: Five snapshots of two rotations of the orange module on a pivot: the first motion connects connector #11 of the orange module to #5 of the *pivot* and second #10 to #7 of the *pivot*.

and Figure 1.6 shows the two possible ways of performing rotations, using an octagonal actuator on the left (rotating around the green pivot), and using a hexagonal actuator on the right (rotating around the yellow pivot). Each rotation displaces the rotating module from one cell to an adjacent one within the FCC lattice. More complex motions comprised of several steps are therefore sequences of individual rotations on the surface of neighbor modules. All lattice cells are not accessible to every module, these restrictions are the result of motion constraints.

First, a module can perform a motion from its current position to an adjacent one if and only if one of its immediate neighbors can act as a pivot for that motion. However, it does not suffice that the pivot has no module connected to the destination connector of the moving module, as there could be another module blocking the motion in the neighborhood of the pivot. For instance, if a module is attached to connector #0 of the *3D Catom* in Figure 1.3 and seeks to move to connector #2 of the pivot, it would not be able to do so using the octagonal actuator if the pivot has a neighbor on its interface #5. Accordingly, using the hexagonal actuator of the pivot would not be possible either if it has a neighbor on its interface #10, in which case that module would not be an appropriate pivot for that motion.

It thus becomes apparent that locally planning motion is not a trivial task for a *3D Catom*. We will say that a module is *mobile* if it can perform at least one motion in any direction. 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* (as illustrated in the rightmost part of Figure 1.6). In terms of the module, this means that a module cannot move if it has neighbors connected to two of its interfaces that are opposite, such as connectors #2 and #8, or #1 and #7, in Figure 1.3. (The number of the connector opposite to # N is always $\#N_{opp}$, where $N_{opp} = (N + 6) \bmod 12$.)

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 using *3D Catoms*, 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**.

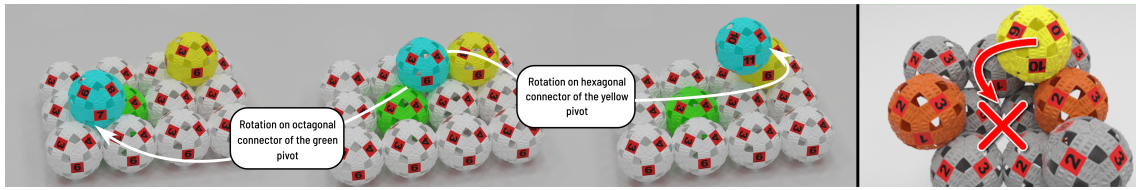


Figure 1.6: **(Left)** Two additional sample motions on octagonal and hexagonal actuators. **(Right)** The *bridging constraint*, in which the yellow module is unable to reach its destination because of the two blocking modules in orange.

Furthermore, any motion that would result in a collision with another module is prohibited. There are two possible scenarios in which that could happen. On the one hand, there is the presence of a module that is not in the local neighborhood (i.e., the 12 immediate neighbors) of a mobile module, and that nonetheless blocks this module from entering its target cell, resulting in a collision. While a module can directly sense its immediate neighbors to ensure that none might impede on its motion, there is no local way of doing such verification on a wider radius in our model. We will refer to this problem as the **remote blocking conundrum**. On the other hand, a collision could occur between two mobile modules if the two have planned concurrent and intersecting motions, which is also a scenario that cannot be prevented solely from the local scope of a module. This is the **motion coordination challenge**. We will discuss potential solutions to these problems after having introduced the communication capabilities of *3D Catoms*. Figure 1.7 illustrates these two problems: On the left, the motion of the green module will collide with the orange module even though its local neighborhood and the one of its pivot (magenta) are clear for this motion; On the right, both yellow modules are attempting to enter the same lattice position, as no local constraint prevents it, which will result a collision between the two.

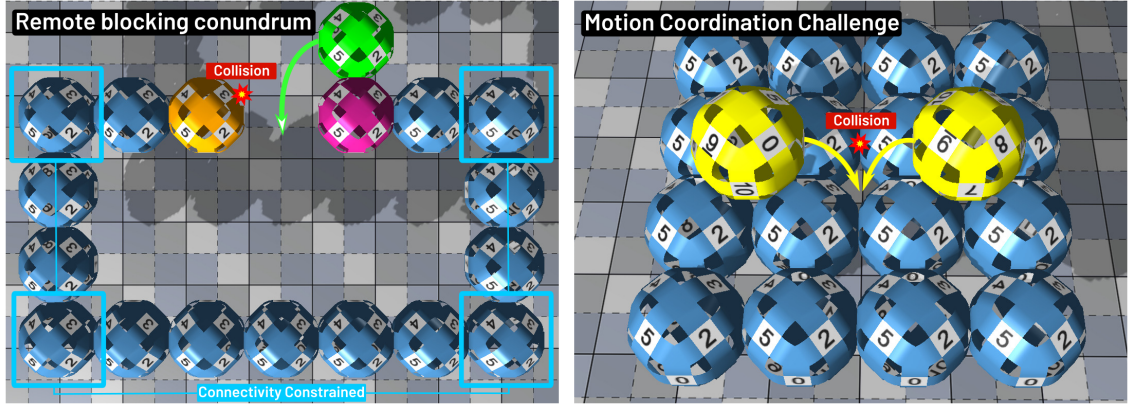


Figure 1.7: The two main motion challenges posed by the *3D Catom* model: the **remote blocking conundrum** and the **motion coordination challenge**.

Communication Latching and motion actuation are not the sole purpose of the electrostatic connectors on the surface of *3D Catoms*. We consider in our *3D Catom* model that all communication is performed locally, according to the distributed communication paradigm, and in a peer-to-peer fashion between connected modules through their electrostatic connectors. Hence, *3D Catoms* cannot rely on global communication to receive essential data about the state of the system, but must instead propagate information distributively.

Motion Constraints As a consequence, when considering motion and communication constraints together, we see that modules can only sense their local neighborhood (immediate neighbors) through the connection of their interfaces, which they can also use to communicate with their first degree neighbors exclusively. However, modules in their second-degree neighborhood (the neighbors of their neighbors), can still cause trouble (cf. the **remote blocking conundrum**). Validating a candidate motion thus involves ensuring that no position in the first and second degree neighborhoods of a module are blocking that motion. The former is done locally, therefore, while the second has to be done through remote communication, distributively searching the graph of the configuration until all lights are green. Unfortunately, this necessarily means performing an entire flooding of the *3D Catom* network every time a motion must occur, which is insanely prohibitive due to the sheer number of messages that such verification would require (as well as the time and energy cost). Besides, the **motion coordination challenge**, that is in itself a *race condition*, can be solved by more local mechanisms, such as the virtual locking of lattice cells surrounding a moving module, much like *mutual exclusion* around a shared resource in programs with concurrency situations. While this is not local to the mobile modules, it is at least regional, as they would only need to lock all the cells in a one-module-thick FCC ball (cf. Figure 1.4d) around their initial position. Nonetheless,

the lack of shared memory primitives between nearby modules still makes such strategy quite burdensome and impractical.

There is one last motion constraint that has not yet been mentioned, named the **connectivity constraint**. It states that all modules in a *3D Catom* ensemble must remain connected as a single ensemble at all times. In other words, the *connectivity graph* $G = (V, E)$ where V is the set of all modules in the system and E the interconnection and between modules through their interfaces. According to the connectivity constraint, G must remain a *connected* graph at all times, that is to say that no module motion that would split the graph G into two disconnected subparts is allowed (even temporarily). An example is given in Figure 1.7, where all the modules outlined with a light blue square are not allowed to move due to the connectivity constraint, even though they are mobile in all other aspects. The purpose of the connectivity constraint is to guarantee that modules can always act as a single global entity (i.e., they can always communicate with each other), and that the ensemble is therefore never split into two or more disjoint parts. Furthermore, in some modular robots, this also ensures that power can always be supplied to all modules in the system. Indeed, some modular robotic systems are likely to require an external source of a power, that would have to be routed to all modules in the system (Daymude et al., 2020)—a daunting challenge.

Much like for the **remote blocking conundrum**, checking the connectivity constraint also requires a massive communication overhead, as it involves evaluating whether a module seeking to move is an *articulation point* of the configuration graph. Støy (2004) showed that this could be one by using a *connection gradient* propagated from modules that are in their final positions, and with an *invariant* on module motion stating that the motion should not alter the connection gradients of the surrounding modules.

We have thus seen that motion constraints exist at two levels for a *3D Catom*: **local** motion constraints and **global** ones. All motion constraints imposed on *3D Catom* modules are summarized below, with for each of them the corresponding condition that has to be evaluated by the module before attempting a motion:

1. Local motion constraints can be directly evaluated by a module:
 - The pivot constraint: *"Can one of my nonmoving neighbors act as a pivot to get me to the cell I am trying to reach?"*
 - The local criterion of the bridging constraint: *"Do I have two neighbors that are on opposite connectors?"*
2. Global motion constraints, however, must be resolved through regional or systemwide distributed communication:
 - The global criterion of the bridging constraint: *"Is there two opposite neighbors*

surrounding the position that I am trying to reach?"

- The remote blocking conundrum: *"Is there a non-connected module that might impinge on my motion path?"*
- The motion coordination challenge: *"Is my motion crossing the path of another module's motion?"*
- The connectivity constraint: *"Will this motion split the 3D Catom ensemble in two disconnected subparts?"*

1.2.2/ PROGRAMMING MODEL AND ASSUMPTIONS

As we have seen, 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. Below are a number of additional assumptions that govern the *3D Catom* ensembles under consideration in our work.

We model a modular robot consisting of a connected ensemble of *3D Catoms* as a distributed system, where:

- The system is **fully distributed** (*3D Catoms* do not receive any command a pre-determined leader from outside the system, nor from inside it, but transient global or local leaders can *emerge* according to the distributed paradigm through *leader election* algorithms.);
- each module is assigned a **unique identifier** beforehand;
- all modules are identical and execute the exact same distributed program—*3D Catoms* thus form a **homogeneous modular robot**;
- the graph constituted by all modules in the systems and their interconnection must remain connected at all times—this is the **connectivity constraint** introduced earlier;
- 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 timed interruption or the start or the end of a motion;
- computation is only performed locally to each *3D Catom*;
- communication is also performed exclusively in a local fashion, with modules only communicating with their immediate neighbors on the FCC grid, and through a **message-passing scheme**;

- message sending and message propagation time are negligible against the rotation time of *3D Catoms*;
- all modules share a common coordinate system and have a global knowledge of the goal shape (Tucci et al., 2017) for self-reconfiguration tasks;
- modules perform everything **asynchronously**.

1.2.3/ FIRST HARDWARE PROTOTYPE

Although this can seem like a strictly abstract model, several partners from the Programmable Matter Project are actively engaged in creating hardware *3D Catoms*, as mentioned in the introduction to this chapter. Achieving the project's goal of producing micro-scale modular robots requires major advancements in all the fields in which partners are involved. For that reason, designing the first hardware *3D Catom* prototype first required collecting the requirements and constraints imposed by the current state of the art of each partner's specialty, and discussing ways to combine them all into an autonomous micro-robot. These discussions concluded that in light of the current state of progress in all the concerned fields of research, a *3D Catom* prototype with a diameter as small as 3.6 mm is the current frontier in miniaturization using today's technology.

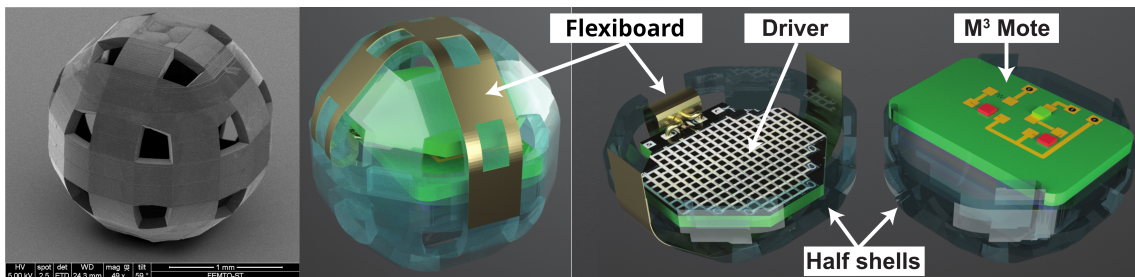


Figure 1.8: First *3D Catom* prototype. From left to right: 3.6 mm catom shell, shell covered with electrostatic actuators, photovoltaic power conversion driver, catom-embedded M^3 Mote. (Left: Nanoscribe picture, courtesy of Gwenn Ulliac - FEMTO-ST.)

The *3D Catom* design under realization therefore consists in a 3D-printed envelope (cf. Figure 1.8a), with electrostatic actuators on its external surface (cf. Figure 1.8b), and with a processor, battery, and drivers for commanding the actuators embedded inside (cf. Figure 1.8c,d). The brain of a *3D Catom* is the Michigan Micro Mote (M^3) (Pannuto et al., 2013), the world's smallest computer at the time of writing, which is a testimony to the cutting-edge nature of the *3D Catom* technology. With the M^3 system comes a battery that provides some energetic autonomy to the modules. Furthermore, at the current scale, a single *3D Catom* would be able to support a vertical load of several dozen modules on a single electrostatic actuator.

The relevance of hardware progress for the work presented in this thesis is that it guides our software research by providing a set of specifications and requirements that describes what is possible in our algorithmic models.

1.3/ DEDICATED SIMULATION FRAMEWORK

In this section, we introduce **VisibleSim**¹, a framework for creating *behavioral* simulators for distributed lattice-based modular robotic systems in a regular 3D environment. VisibleSim is the dedicated simulation platform of the Programmable Matter project, and as such is the tool that facilitated all the simulation experiments presented in this work. It was developed by our FEMTO-ST team in the early 2010s, and has received continuous maintenance and many updates and features since then. Even though I was not involved at the time of its conception, I have spent countless hours developing VisibleSim into what it is today over the years: first performing a complete overhaul of the simulator's architecture and various additional development tasks during a 3-month research internship at The Hong Kong Polytechnic University (where Prof. Jiannong Cao's team from the Internet and Mobile Computing Lab was using our simulator as part of a partnership); then supporting the development of the simulator at all levels and facilitating new user adoptions during the three years of this thesis. As such, while Benoît Piranda is the person in charge of the development of the software, one could say I am its technical specialist (my contribution represents around 1100 *git* commits, or slightly more than half the total number of commits created since *VisibleSim* was added to Github in 2015). This section tries to make the case for the necessity of a tool such as VisibleSim to support our research efforts, describes the simulation model that is used in the rest of the manuscript, and shows that it is particularly well suited for the tasks at hand.

Simulation is an essential part of robotics research. In most cases, it is a tool that is used for testing out prototypes of robotic hardware and controllers quickly, in all kinds of environments, and at a low cost. It is also sometimes used as a training ground to evolve powerful controllers for robotic control through generic programming (Wang et al., 2013; Vonásek et al., 2013), or to perfect the coordination of swarm or multi-robot systems through methods such as reinforcement learning (Varshavskaya et al., 2008). But simulations can also be used as a platform for building the software foundations of future complex systems that are not yet producible. Such a process can guide hardware development by producing a critical set of requirements for those systems.

This distinction is particularly relevant to the field of self-reconfiguring modular robotic systems (Ahmadzadeh et al., 2016). Such systems are made from an arrangement of individual and autonomous modules (from several to thousands in number) attached to each

¹VisibleSim's home page: <https://projects.femto-st.fr/programmable-matter/visiblesim>

other that must coordinate to achieve a common goal, and that can change their morphology through the motions of their parts. Many tasks such as self-reconfiguration (Thalamy et al., 2019b) have been well researched for decimeter- and centimeter-scale modular robots consisting of several modules, using both hardware and simulation, but hardware results are still far away for complex larger-scale systems in the millimeter range or consisting of up to thousands of modules. Enabling their simulation is thus currently essential for imparting researchers with a better understanding of the complex dynamic behavior found in such systems.

Physics simulation is not the purpose of VisibleSim, instead, its aim is to provide the tools for studying the behavior of such systems, through the simulation of the interactions between the robotic modules and their environment (neighbor detection, motions within the lattice, user interactions, etc.), or between the modules themselves (communication between modules through message passing). Each module in the system is assigned a unique identifier, and executes the same controller as all other modules, generating communication or environmental events that are handled deterministically by VisibleSim's discrete-event scheduler. This allows for accurate simulations of complex algorithms on robotic ensembles up to millions of modules in size on a single computer. A simulation with 30 million communicating and moving modules was successfully run in 2020. VisibleSim also doubles as a powerful visualization tool for impactful academic research results thanks to its wide choice of options for media export.

1.3.1/ RELATED SIMULATION PLATFORMS

There are numerous competing general simulation software for autonomous robots, many of which are open source (Kramer et al., 2007). Some of them have been around since the beginning of the millennium and have been widely used for all kinds of robotics projects such as the Player (Gerkey et al., 2003) framework, either running along Stage for simulating large ensembles of robots at low-fidelity in a 2D environment, or coupled with Gazebo (Koenig et al., 2004), better suited for simulations involving a limited number of robots in a realistic 3D environment. Gazebo can also be used as a standalone or interact with a number of other control software, and provides multiple physics engine to customize simulations. Webots (Michel, 2004) is another reference of a commercial open-source simulator, now a leader in this field, with more than 20 years in the making and providing much flexibility its users, allowing them to model all kinds of complex mechanical systems. Other projects include USARSim (Carpin et al., 2007), a hyper-realistic simulator based on the Unreal gaming engine and originally specialized in search-and-rescue simulation; Microsoft Robotics Studio (Jackson, 2008), whose support has since been discontinued; and the more recent ARGos (Pinciroli et al., 2012) simulator for swarm robotics, which can simulate large and heterogeneous multi-robot systems.

While some of these general-purpose simulators have been used for simulating modular robotic systems, such as Webots for the YaMoR (Moeckel et al., 2006) modular robot, they are more often the exception than the norm. Modular robotics research usually relies on dedicated simulation platforms. These simulation platforms are in most cases exclusive to a single modular robotic architecture, though a few generic simulators also exist and will be discussed below. Physics-based and general-purpose simulators are nonetheless complementary to other types of simulators as they are necessary for closing the reality gap between theoretical models and actual hardware implementations.

On the one hand, regarding hardware-specific modular robot simulators, many platforms for simulating self-reconfiguration for lattice modular robots are unnamed simulators with basic features implemented using Java3D (Yim et al., 2001; Støy et al., 2007; Vassilvitskii et al., 2002; Fitch et al., 2003). Even though hardware-specific simulators are generally limited, CubeInterface (Zykov et al., 2008), for the chain modular robot Molecubes, an advanced user interface for designing Molecubes modular robots by hand or through the autonomous evolution of their controllers, as well as for interacting with individual modules.

On the other hand, several generic modular robot simulators have been developed, mostly physics-based and targeting chain or hybrid modular robots: First, there is Rebots (Collins et al., 2016), a high-performance simulator supporting several robotic architectures (Roombots (Spröwitz et al., 2010), Smores (Davey et al., 2012), Superbot (Salemi et al., 2006)), with advanced user interactions such as drag-and-drop, without sacrificing programmability. The same team behind Rebots had also previously introduced ReMod3D (Collins et al., 2013), another high-performance self-reconfigurable robot simulator well suited for a vast number of modular robotics tasks with ATRON (Jorgensen et al., 2004) and Superbot modules. Then, there is Sim (Vonásek et al., 2013), a lightweight simulator designed to run at a low computational cost in headless mode, and targeting both wheeled mobile robots and the Scout modular robot of the SYMBRION and REPLICATOR projects (Kernbach et al., 2008). Also related to the same projects, the Symbricator3D simulator (Winkler et al., 2009) can run simulations of both SYMBRION and REPLICATOR robotic modules at different degrees of dynamism, from single modules to entire swarms. Lastly, USSR (Christensen et al., 2008) is another unified simulator for self-reconfigurable chain- and lattice-based modular robots, supporting the ATRON, Odin (Lyder et al., 2008), and M-TRAN (Kamimura et al., 2002) modular robots. In practice, it is more of a framework for building modular robot simulators.

Nonetheless, there are still a few generic simulators that are designed for lattice modular robots, the type of robots targeted by VisibleSim. For example, SRSim (Fitch et al., 2003) has been used to simulate self-reconfiguration and locomotion of both lattice (Sliding-Cube (Fitch et al., 2003) and Crystalline (Rus et al., 2001) (2D)) and hybrid (Super-

bot (Fitch et al., 2008)) modular robots using Java3D. SRSim has also been extended to support hardware-in-the-loop simulation, for a more realistic setting for controller development. Finally, DPRSim (Ashley-Rollman et al., 2011) has been shown to efficiently simulate ensembles with up to 20 million Catom modules, both in their 2D and 3D forms, by maximally leveraging the potential for multithreading of the simulation and using computing clusters. This simulator has also benefited from advanced integrated debugging features such as simulation replay and distributed tracing, inspection, and visualization of data (Rister et al., 2007).

The common feature among all the aforementioned simulators except perhaps SRSim, is that they are all physics-based, which is useful when developing robotic designs and mechanically evaluating them, evolving controllers, and interacting with real-world complex environments, but might be superfluous and prohibitively costly when researching distributed robotic control from a more fundamental, or *behavioral*, point of view. This is the kind of simulator that VisibleSim thus aspires to be, a framework for performing all kinds of behavioral simulations on lattice-based modular robotic systems with low environmental interactions. Furthermore, it is most similar to USSR and SRSim in its usage, being a framework for developing simulators rather than an actual monolithic executable software where all simulation parameters are interpreted.

1.3.2/ VISIBLESIM

1.3.2.1/ OVERVIEW

VisibleSim is designed for researchers that have computer programming experience as it consists in a **C++ framework** for building lattice modular robot simulators controlled by distributed programming. Several sample modular robot simulators are provided with the software. *VisibleSim* takes the form of an open source project under AGPLv3 license and is available on Github².

In *VisibleSim* lingo, the distributed program that is executed on each module during the simulation is named a *BlockCode*. It is effectively the controller of the modules and where users will describe the behavior of the robot in response to all kinds of events whether external (interactions with the world, reception of a message, etc...), or internal (interruption or timer, initialization, end of a motion, etc...).

Unlike other simulators where each robot is fitted with a number of sensor and actuator components, this distinction is not materialized in *VisibleSim*. Modules from any type of robots are however fitted with a constant number of *interfaces*, depending on the geometry of their lattice, and which can both be used for sensing connected modules (by ex-

²VisibleSim's public Github repository: <https://github.com/VisibleSim/VisibleSim>

amining whether an interface is connected) and communicating with them. In the current state of the simulator and as in most existing work on modular robotics, communication between modules is only natively allowed in a peer-to-peer manner between connected neighbors.

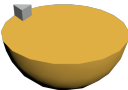

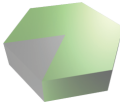


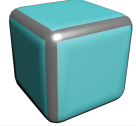
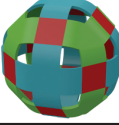

2D Nodes Square lattice (2D) 4 neighbors Motion: Slides along a neighbor or Turns around an edge. Display: Lights in color 	Smart Blocks Square lattice (2D) 4 neighbors Motion: Slides along a vertical border Display: Lights in color and draws numbers on the top 
Hexanodes Hexagonal lattice (2D) 6 neighbors Motion: Turns around a neighbor (pivot) Display: Lights in color 	2D Catoms Hexagonal lattice (2D vertical) 6 neighbors Motion: Turns around a neighbor (pivot) Display: Lights in color 
Blinky Blocks Cubic (3D) 6 neighbors Display: Lights in color Sensor: tap 	Sliding Cubes Cubic (3D) 6 neighbors Motion: Slides along a neighbor or Turns around an edge. Display: Lights in color 
3D Catoms Face-Centered Cubic lattice (3D) 12 neighbors Motion: Turns around a neighbor (pivot) Display: Lights in color 	Datoms Face-Centered Cubic lattice (3D) 12 neighbors Motion: Deforms to turn around a neighbor (pivot) Display: Lights in color 

Figure 1.9: Several shapes of robots proposed in *VisibleSim*.

Previous work on modular robots can be classified based on the shape of the robots and the type of grid in which they are placed. Each grid has a specific number of positions adjacent to one of its cells, which determines the number of neighbors a module in that grid can communicate with. Some instances of 2D and 3D grids are given below.

Some modular robots are placed in cubic lattices like the *Telecubes* (Suh et al., 2002)—consisting of cubic modules that can move thanks to the contraction or expansion of its arms—, the *Miche* (Gilpin et al., 2008)—made of centimeter-scale cubic modules with disassembling capabilities—, and the *Blinky Blocks*, which cannot self-reconfigure by themselves. They can be abstracted and simulated using the *Sliding Cube* model.

Others are confined to a 2D grid, like the 2D square lattice of the *Smart Blocks* (Piranda et al., 2013), whose aim is to build a large distributed modular conveyor belt to convey small and fragile objects. Another 2D grid instance is the hexagonal lattice, used by the *Distributed Flight Array* (Oung et al., 2011), where modular robots are hexagonal drones which are able to assemble into an hexagonal lattice structure and fly together. The *2D Catoms* developed within the *Claytronics* project—cylinders that are able to roll around a neighbor—also use an hexagonal grid, but a vertical one.

Finally, other work propose modular robots in a Face Centered Cubic (FCC) lattice, such as Proteo (Yim et al., 2001), based on a rhombic dodecahedral geometry, the quasi-spherical *3D Catoms* (Piranda et al., 2018), or the deformable *Datoms*.

VisibleSim offers different classes of modular robots across these different lattices, as shown in Figure 1.9.

What characterizes a modular robot in *VisibleSim* is therefore: the geometry and visual aspect of its modules; the lattice in which they belong (hence their number of possible neighbors), and a specific mode of motion. Additional components and state visualization features such as a display, speakers, or tap sensors can however be added.

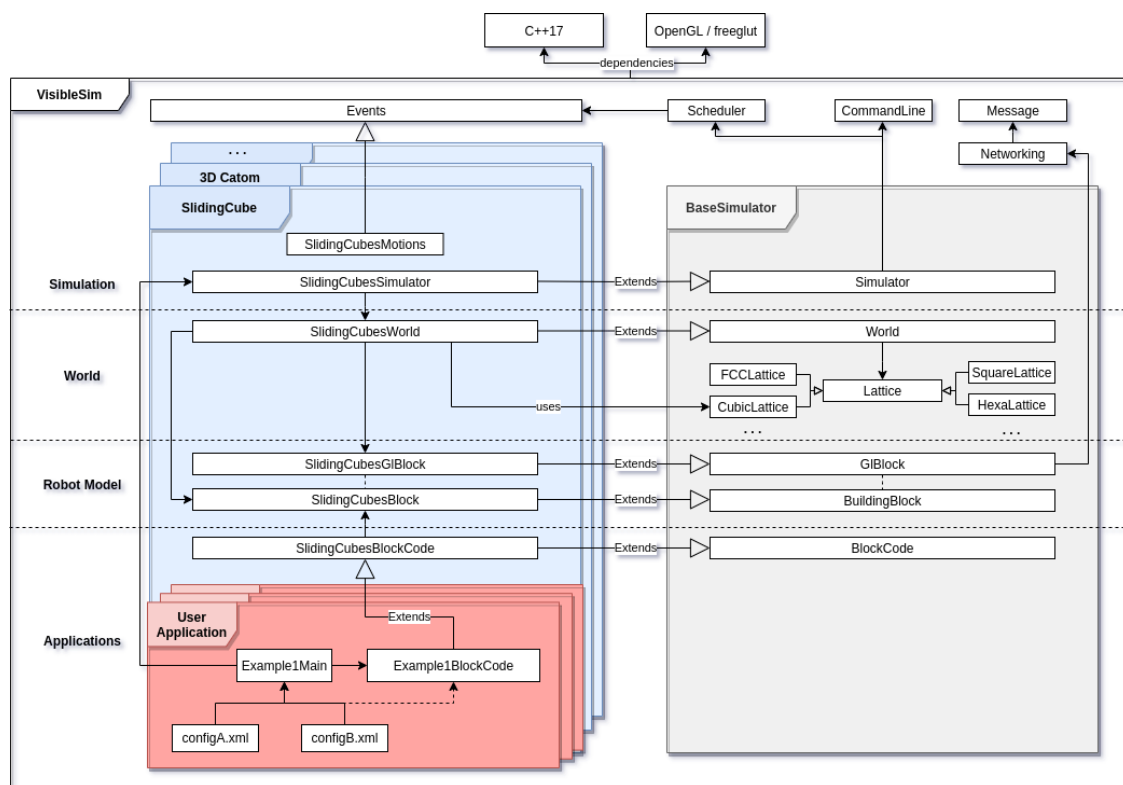


Figure 1.10: **Architecture of the simulator:** *BaseSimulator* framework in grey; framework instantiations for each modular robot in blue; user applications for a given modular robot in red, here with 2 configuration files.

Architecture The architecture of *VisibleSim* heavily relies on object-oriented programming (OOP). The core of the simulator, named *BaseSimulator* consists in a number of abstract C++ classes with a number of pure virtual functions that have to be extended for each modular robot (in grey in Figure 1.10). The result is a full instantiation of the *BaseSimulator* components for each module type (in blue in Figure 1.10), relying on the logic found in the *BaseSimulator* and additional supporting libraries from the core of the simulator. Particular motions and other robot-specific features must also be implemented

as events pertaining to that module type. Then, for an end-user of given modular robot simulator instance (in red in Figure 1.10), creating a distributed program means extending the *[Module]BlockCode* abstract class for that given module, and implementing the functions describing the course of action of a module upon startup, and in response to all relevant events to which it might be exposed. Each application *BlockCode* also requires a main function that will handle the lifecycle of the simulation, and has to be supplied a configuration file that describes modular robotic configuration and other parameters of the simulation.

As previously mentioned, the *BlockCode* element is essentially the controller of the robot. Now, unlike what is often found in continuous time simulators, these controllers do not live in separate threads that are continuously updating the simulation engine, but instead they expose a number of functions executed only in response to the handling of simulation events concerning their host module by the main scheduler thread, discussed thereafter.

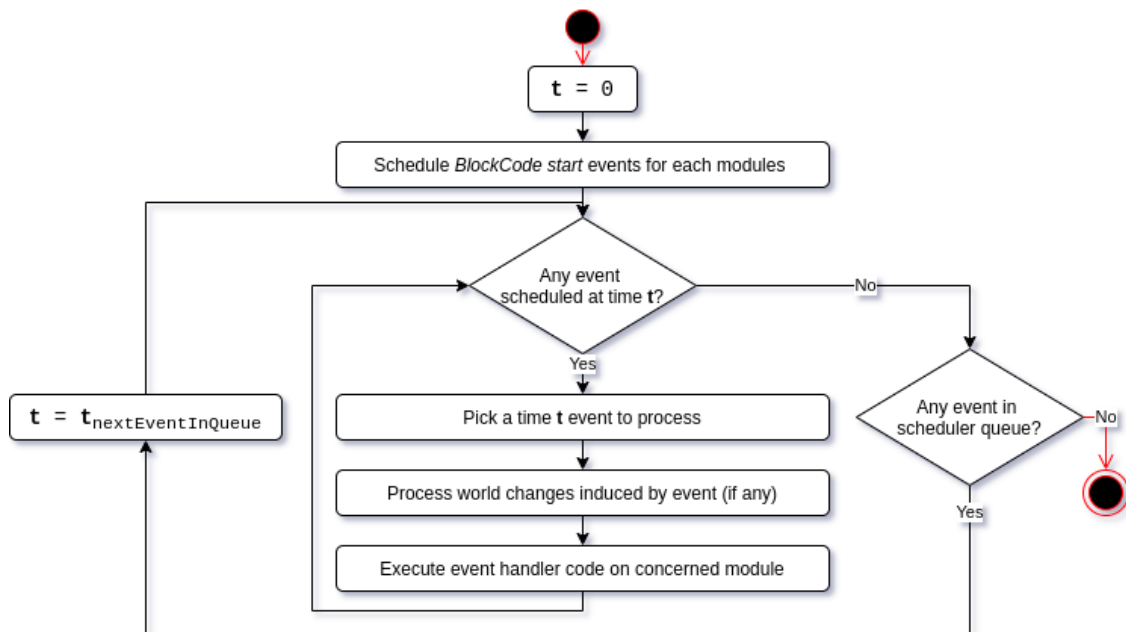


Figure 1.11: Main loop of the *VisibleSim* simulator

Scheduling *VisibleSim* is powered by discrete-event simulation (DES), which means that it relies on a scheduler that executes a sequence of discrete actions over time scheduled at fixed dates. Each event represents an action on the simulated world and potentially causes new events to be scheduled at a later date. It is assumed that all changes in the simulated world and modules are caused by the processing of an event and thus the entire simulation can be reduced to an ordered sequence of events and their effects on the simulated world. Two modes of DES are supported in *VisibleSim*: *fixed-increment time progression* (real-time mode), where time passes linearly by increment of a constant

value, or *next-event time progression* (fastest mode), where the current date of the scheduler is always the date of the next event in queue. This is the foundation of our behavioral simulator and we find that it is particularly well-suited for lattice modular robot simulation, as the modules themselves reside in a grid that is also discrete, thus reducing the possible location of the modules of the system and yielding a much more efficient simulation than what is possible with continuous time simulation. Figure 1.11 shows the main loop of our simulator in next-event time progression mode, essentially always picking the next event in queue, updating the world, and executing the handler code for that event on the controller of the concerned module. When the simulation starts however, an initial *start* event is scheduled for each module in the configuration, which will trigger the initialization code for the controller of each of them.

In a sense this entire procedure could be summarized as a *Load - Process - Terminate* lifecycle: *loading* the initial simulation events from the modules at the start; *processing* these events and scheduling new ones in the process; and when there are no events left to process, *terminating*.

1.3.2.2/ PROGRAMMING ENVIRONMENT AND FEATURES

User Application Demonstration This section presents an example of a *SlidingCube* modular robot application, where a message is broadcast distributively through the robot from a leader module (identified by its identifier) to instruct modules to perform a random motion. Though this application has no practical purpose, it demonstrates concisely the structure of a user application as well as elements of its motion and communication API.

Furthermore, a visual *BlockCode* generator is available online³, which takes a target robotic architecture and a list of messages as input and returns a code template for that setup.

Listing 1.1: **Sample BlockCode:** Broadcast of a message across the robot from a master module and moves upon reception

```
#include "exampleBlockCode.h"

void ExampleBlockCode::startup() {
    addMessageEventFunc(BROADCAST_MSG, std::bind(&exampleBlockCode::onBroadcastRcvd,
    this, std::placeholders::_1, std::placeholders::_2));
    if (module->blockId == 1) { // module #1 is the master
        this->broadcastReceived = true;
        sendMessageToAllNeighbors(new Message(BROADCAST_MSG));
    } else {
        this->broadcastReceived = false;
    }
}
```

³VisibleSim BlockCode generator: <http://ceram.pu-pm.univ-fcomte.fr:5015/visiblesim/doc/codeblock.php>

```

}

void ExampleBlockCode::onBroadcastRcvd(std::shared_ptr<Message> msg,
                                       P2PNetworkInterface* sender) {
    if (not this->broadcastReceived) {
        this->broadcastReceived = true;
        // Propagate broadcast and move to first available location
        sendMessageToAllNeighbors(new Message(BROADCAST_MSG), sender); // ignore sender

        std::list<Cell3DPosition> destinations = getAllPossibleMotionCells();
        if (not list.empty()) initiateMotionTo(destinations.front());
    }
}

```

Listing 1.2: **Sample main file:** initiates and cleans up the simulation

```

#include <iostream>
#include "robots/slidingCubes/slidingCubesSimulator.h"
#include "robots/slidingCubes/slidingCubesBlockCode.h"
#include "exampleBlockCode.h"

int main(int argc, char **argv) {
    // Returns only once scheduler has ended
    createSimulator(argc, argv, ExampleBlockCode::buildNewBlockCode);
    deleteSimulator();
    return 0;
}

```

Listing 1.3: **Sample XML configuration file:** describes the simulated world, the modules within it, and other simulation parameters

```

<?xml version="1.0" standalone="no" ?>
<world gridSize="20,20,20" windowSize="1920,1080">
    <blockList defaultColor="128,128,128" ids="RANDOM">
        <!-- Describe individual modules -->
        <block position="3,4,2" color="127,255,43" />
        <!-- or use Constructive Solid Geometry (CSG) -->
        <csg content="union() { cube([10, 5, 5]); cube([5, 10, 5]); }"/>
    </blockList>
    <targetList> <!-- Goal configuration for reconfiguration -->
        <target format="csg">
            <csg content="sphere(10)"/>
        </target>
    </targetList>
</world>

```

User Interactions In fixed-increment time progression mode, *VisibleSim* supports pausing and resuming of the simulation (programmatically or using the keyboard), which

can be used to inspect the simulated world at any given time. This is especially useful since *VisibleSim* has a built-in console that provides useful information about a number of built-in (messages sent or received, motions, etc.) or custom (any user-implemented event or debugging trace) events. This includes the time of the event and any other useful information that is necessary to retrace what the chain of events that led to the current state of the simulation. Not only can the traces concerning all the modules in the system be shown at once from within the simulation window, but individual threads of events relative to a specific module can be shown by selecting the module from the GUI.

Furthermore, left-clicking a module opens a pop-up for interacting with the simulated world and the module itself. These interactions are the addition and removal of neighbors on the interfaces of a module, motion commands, or a physical event such as an accelerometer tap. Finally, the current world configuration can be exported, making *VisibleSim* both a simulation software and a sandbox for building robotic configurations.

Customization Hooks *VisibleSim* proposes a number of customization hooks that are called at various points of the simulation and that can be used to implement custom behaviors for a given *BlockCode* application. Some of these functions provide greater flexibility to the user, others simply facilitate debugging:

- Parsing custom configuration file elements pertaining to the world or to individual modules.
- Parsing custom command line arguments exclusive to this specific *BlockCode* application.
- Respond to custom keyboard events generated by the user during simulation and specific to that application.
- Drawing custom graphical elements in the OpenGL world every time it is updated.
- Drawing custom text onto the OpenGL window to keep some essential information always visible.
- A custom function that gets called on a module whenever a *VisibleSim* assertion has been triggered for that module, and that can provide critical information on its current state.

Export Tools In fixed-increment time progression mode, *VisibleSim* supports pausing and resuming of the simulation (programmatically or using the keyboard), which can be used to inspect the simulated world at any given time. This is especially useful since *VisibleSim* has a built-in console that provides useful information about a number of built-in (messages sent or received, motions, etc...) or custom (any user-implemented event or debugging trace) events. This includes the time of the event and any other useful

information that is necessary to retrace what the chain of events that led to the current state of the simulation. Not only can the traces concerning all the modules in the system be shown at once from within the simulation window, but individual threads of events relative to a specific module can be shown by selecting the module from the GUI.

Furthermore, left-clicking a module opens a pop-up for interacting with the simulated world and the module itself. These interactions are the addition and removal of neighbors on the interfaces of a module, motion commands, or a physical event such as an accelerometer tap. Finally, the current world configuration can be exported, making *VisibleSim* both a simulation software and a sandbox for building robotic configurations.

Finally, *VisibleSim* provides a fast and simple way to generate image or video captures of simulations, and can even export a robotic configuration to a STereoLithography (STL) file for 3D printing.

1.3.2.3/ USAGE AND EVALUATION

In this section, we highlight a number of different modular robots and applications that have been successfully simulated using *VisibleSim* in published research. Our aim is to highlight different ways *VisibleSim* can be used. We also show that the simulation of existing hardware system can show a high level of fidelity to hardware experiments. Finally, we bring to light the current capabilities of *VisibleSim* in terms of scalability.

Simulation Fidelity In addition to faithfully reproducing algorithm functional behavior, *VisibleSim* also accurately simulates timing. Communication and clock models can be customized and passed to *VisibleSim* in order to fit with the simulated modular robotic platform.

After having modeled the communication system of the *Blinky Blocks* in *VisibleSim* (Naz, 2017), we measured the execution time of the ABC-CenterV1 algorithm (Naz et al., 2015; Naz, 2017) – an algorithm for electing an approximate-center module in modular robots – on hardware *Blinky Blocks* and in simulations. Table 1.1 shows that the simulated execution time (average and standard-deviation) on *VisibleSim* closely match the execution time obtained experimentally on hardware *Blinky Blocks*, for small and larger configurations, and for sparse (e.g., lines), less-sparse (e.g., squares), compact (e.g., cubes) and mixed-density configurations with compact components linked by a critical path (e.g., the dumbbell).

We have also modeled the *Blinky Blocks* hardware clocks in *VisibleSim* and evaluated the synchronization precision of the Modular Robot Time Protocol (MRTP) (Naz et al., 2016b) – a protocol for providing global time synchronization across a modular robotic system –

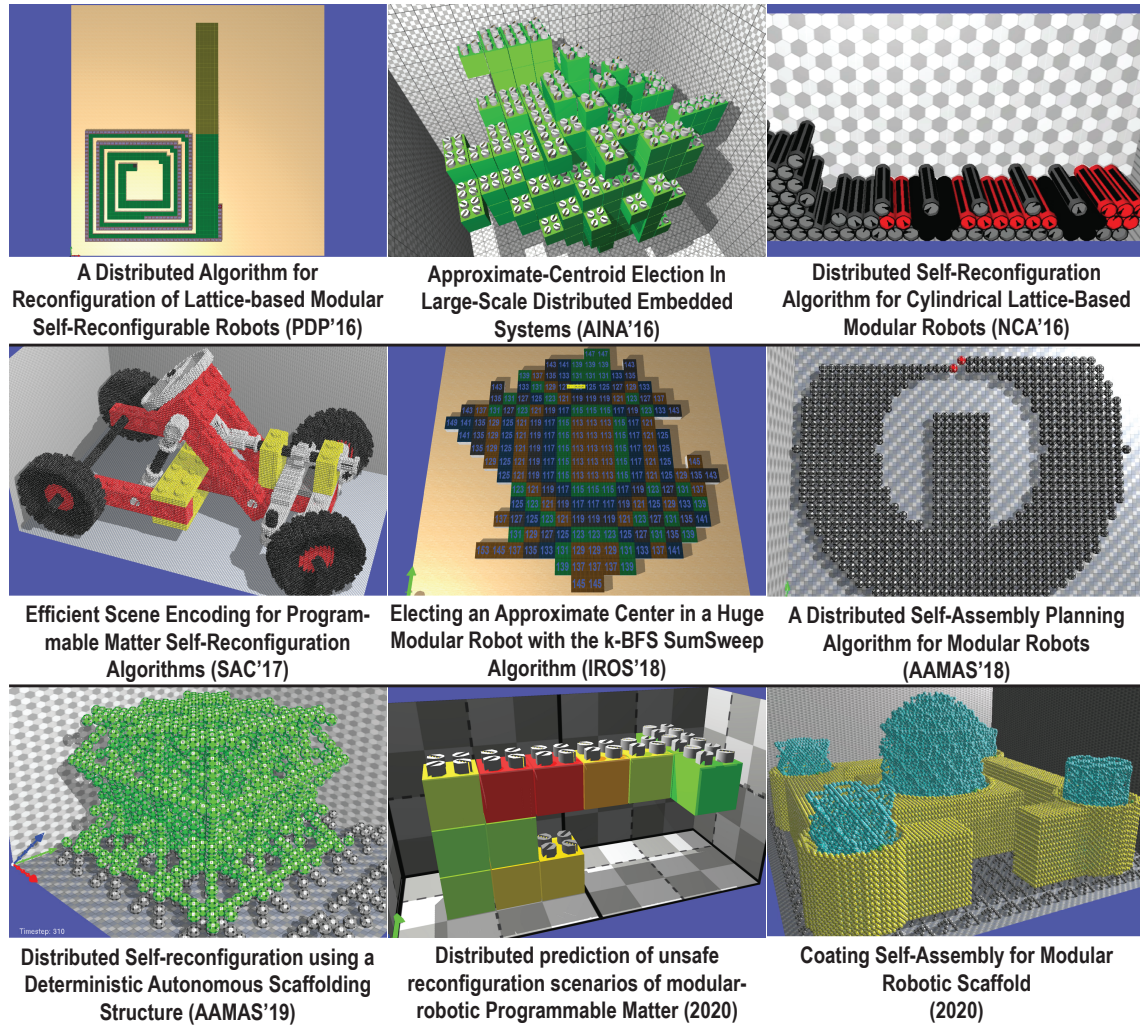


Figure 1.12: Select results from previous work using *VisibleSim* across several module types and tasks.

both with hardware modules and simulations. Experiments were conducted on a doubled L-shaped system composed of 10 *Blinky Blocks* over an hour, with a synchronization period of 5 seconds. Synchronization error distribution looks Gaussian both in simulation and hardware experiment results (Naz, 2017). In the hardware *Blinky Blocks* system (resp. in *VisibleSim*), MRTP has an average precision of 0.06 ms (resp. -0.11 ms) and a standard-deviation of 1.62 ms (resp. 1.40 ms).

Results obtained using *VisibleSim* show a very high fidelity to the hardware results, which indicates that *VisibleSim* is able to perform an accurate timing simulation of the algorithms.

Scalability In order to demonstrate the scalability of the *VisibleSim* simulation framework, we designed a stress test experiment, which consists in simulating a sort of *Brownian motion* of as many modules as possible, within a square grid. The underlying *Block-*

Shape	Size (module)	Diameter (hop)	Average execution time \pm standard deviation (ms)		Absolute error of the average execution time – simulator versus hardware – (ms) (relative error)
			Hardware	Simulator	
Line	5	4	234 \pm 1	244 \pm 3	10 (4.27%)
	10	9	545 \pm 5	544 \pm 5	1 (0.18%)
	50	49	2873 \pm 23	2885 \pm 17	12 (0.42%)
Square	9	4	598 \pm 45	588 \pm 14	10 (1.67%)
	25	8	1117 \pm 30	1119 \pm 27	2 (0.18%)
	49	12	1684 \pm 48	1686 \pm 44	2 (0.12%)
Cube	27	6	1229 \pm 56	1214 \pm 31	15 (1.22%)
	64	9	1927 \pm 51	1941 \pm 33	14 (0.73%)
Dumbbell	59	15	1262 \pm 56	1252 \pm 57	10 (0.79%)

Table 1.1: Average execution time of ABC-CenterV1 on hardware Blinky Blocks and in simulations. Statistics on the execution time were computed over 25 runs for every configuration.

Code program is quite straightforward:

- At the start, a single leader module *activates* and sends an *activation* message to all its neighbor.
- Upon reception of an *activation* message, modules turn into the *activated* state.
- *Activated* modules then alternate between a 0.5 s wait, and a random motion lasting 1 s.
- When a motion ends, the moving module sends an *activation* message to its new neighbors, if any, before starting the next wait/move cycle.
- The simulation ends when all modules are in the *activated* state.

This simple distributed program will thus propagate agitation across an entire modular robotic system, generating a massive number of messages, motions, and wait events in the process. The aim is therefore to stress the *VisibleSim* scheduler as much as possible and show that a graphical simulation is still possible with a massive robotic ensemble.

We run the program with a large set of square configurations⁴ and for each of them we compute the number of messages and the number of displacements that are necessary to activate every modules. For each size of configuration, the initial set of modules is made by growing a tree of modules from a regular list of seeds, ending when branches reach a cell that is already filled.

Table 1.2 shows the number of modules in the configuration for each square configuration. Figure 1.13 shows the number of messages and displacements as a function of the number of robots in the configuration. As shown in the Figure, we are able to simulate up to 30 million robots communicating and moving through the grid, which is to the best

⁴see <https://youtu.be/c5TDelf83Tg> to see the simulation in action on up to 500 000 modules

Square area width	Number of robots
100	4 778
500	144 008
1 000	574 560
1 500	1 292 280
2 000	2 300 696
3 000	5 166 374
4 200	10 122 052
5 200	15 504 808
6 500	20 644 666
7 500	32 212 645

Table 1.2: Number of robots for each grid size of stress test experiment.

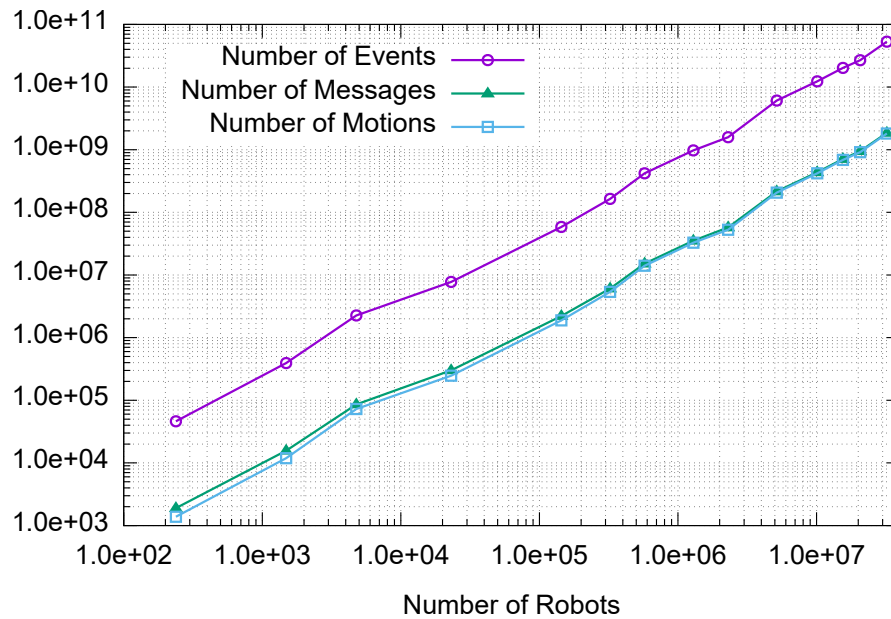


Figure 1.13: Number of motions and messages simulated during the stress test experiment.

of our knowledge a new record in the field of modular robotics simulation. The first experiments, dealing with up to 3 million robots have been made on a laptop (with 32 GB of RAM), and all subsequent simulations have been made on a server with 3 TB of RAM.

1.3.2.4/ DISCUSSION AND FUTURE WORK

In this section, we have introduced *VisibleSim*, a C++ framework for simulating large-scale lattice-based modular robotic ensembles. It differs from other modular robot simulators in its philosophy as a behavior-focused simulator, and its corresponding discrete-event-based style of scheduling. Various modular robotic designs supported by *VisibleSim* have been introduced, along with how to add new architectures by instantiating the OOP sim-

ulator framework, and implementing user applications. We have shown that it doubles as a powerful visualization software for effectively communicating research results, and that the simulator is flexible and easy to customize. Finally, we have outlined the versatility, reliability, and scalability of *VisibleSim*, by showing diverse usages of the software in published research, outlining the accuracy of simulations, and performing graphical simulations with more than a million individual modules. We therefore argue that *VisibleSim* can benefit any present or future research on the algorithmic foundation of modular robotic systems, especially since it is freely available open source software. *VisibleSim* is an ongoing project and there are a number of features that are currently under investigation, detailed below. In its current implementation, all the scheduling tasks are performed on a single thread. While it guarantees an accurate simulation, this also limits the scalability of the software. We are thus enabling multi-thread scheduling for the simulator, which raises a number of challenges for the preservation of the integrity of the simulation flow. Moreover, with distributed algorithms being notoriously difficult to develop and debug, we are seeking to implement DPRSim-style debugging (Rister et al., 2007), and further develop simulation replay to provide critical support to application development.

STATE OF THE ART OF SELF-RECONFIGURATION IN 3D LATTICES

Contents

2.1	Classification of Self-Reconfiguration Approaches	46
2.1.1	Bottom-Up Approach	48
2.1.2	Top-Down Approach	51
2.1.3	Theoretical Approach	59
2.2	Analysis of 3D Lattice Self-Reconfiguration Algorithms	62
2.2.1	Planning Under Mechanical Constraints	64
2.2.2	Free-Space Requirements and Obstacles	64
2.2.3	Collision and Deadlock Prevention Mechanisms	65
2.2.4	Goal-Shape Representation	66
2.2.5	Solution Methods	67
2.2.6	Surface Movements vs. Internal Movements	67
2.2.7	Motion Parallelism and Convergence	68
2.2.8	On the Complexity of Self-Reconfiguration	69
2.2.9	Simulation Environments	69
2.2.10	Evaluation Methods	70
2.2.11	Validation Methods and Analyses	71
2.3	Discussion on Programmable Matter	72
2.3.1	Self-Reconfiguration Criteria	72
2.3.2	Relevance of Existing Works	73
2.3.3	Perspectives	74

We have discussed in the introduction the self-reconfiguration problem, and why it is such a hard problem, no matter the architecture of the modular robot under study. Previous surveys in this field have focused mainly on the hardware problem. One exception is a survey by [Ahmadzadeh et al. \(2015\)](#), which broadly focused on the software challenges of these systems, giving an extensive review of existing methods and algorithms for reconfiguration planning, locomotion control, and synchronization. Though many details were given on the topic of self-reconfiguration and current abstraction and solution methods, we contend that a more in-depth review and comparison of existing methods on this particular topic ought to be proposed. This chapter, which is based on the survey article ([Thalamy et al., 2019b](#)), aims to deliver a detailed account of the current research on modular robotic self-reconfiguration for shape formation, especially in its three-dimensional lattice-based variant, as well as more theoretical works. We proceed by outlining the various high-level methodologies present in the literature, then dive down into the specifics of the algorithms and their underlying models, before focusing on their application to programmable matter. This translates into the following organization: First, we identify three different approaches that researchers in the field have adopted to tackle the self-reconfiguration problem. For each approach, we outline its inherent characteristics and constraints, and briefly mention the main models and works that it encompasses. Then, in Section 2.2 and based on Figure 2.6, we provide an in-depth summary and comparison of the self-reconfiguration algorithms from the previous section. Finally, we reframe the self-reconfiguration problem within the context of programmable matter, discuss properties particularly relevant for this application, and point out promising opportunities for future research.

2.1/ CLASSIFICATION OF SELF-RECONFIGURATION APPROACHES

Historically, researchers in the field of modular self-reconfigurable robotics have initially focused their efforts on the hardware problem of building metamorphic robots; then, research interest in the generic control of classes of these systems gradually emerged and numerous software frameworks were proposed; more recently, researchers started showing interest in what could be considered as a theoretical kind of metamorphic system, in the form of *Self-Organizing Particle Systems (SOPS)*.

In fact, from these three classes we can derive three approaches to self-reconfiguration algorithm design, that we will thereafter refer to as *Bottom-Up*, *Top-Down*, and *Theoretical*, as shown on Figure 2.1. These approaches differ by how they relate to their target execution platform (*Is the target platform designed to accommodate the algorithm, or is it the other way around?*), and, therefore, by the nature of the constraints that make up the model used by the algorithm. Accordingly, the *Theoretical* approach deals with

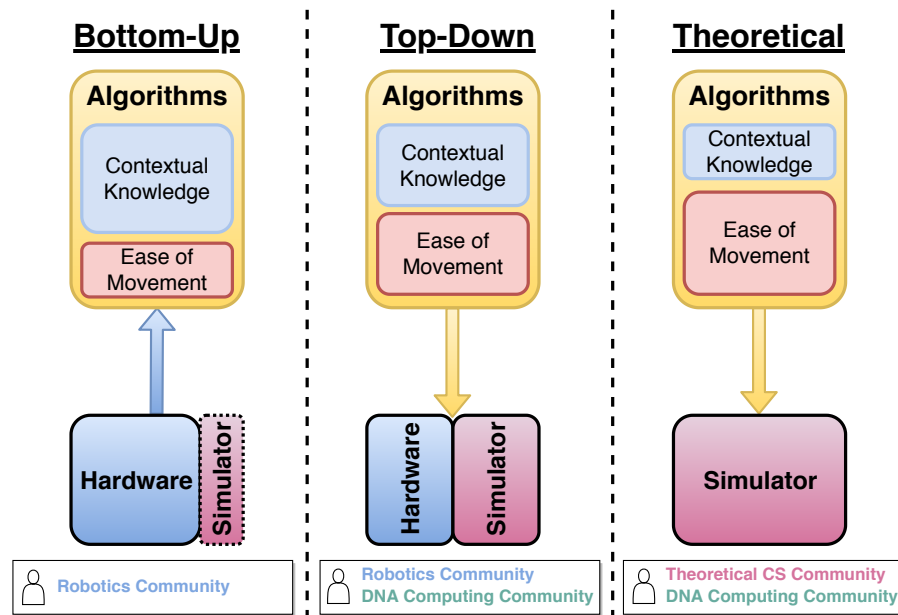


Figure 2.1: The three approaches to designing self-reconfiguration methods and their characteristics.

self-reconfiguration in the abstract, where the complexity and capabilities of the underlying model are often reduced to their minimum. *Bottom-Up* expresses the fact that the hardware systems were designed originally and algorithmic solutions for these specific systems have been subsequently proposed; the control software is hence inherently constrained by the particularities of the specific target platforms and lacks generic features in most cases. Conversely, *Top-Down* expresses the inverse relation, in which algorithms tend to be more generic, using models of the robots in which their specificities are abstracted and that are thus applicable to wide varieties of MSR. This approach will receive most of our interest, as comparison between generic algorithms is more enlightening. The essential difference with the Theoretical approach is that Top-Down works are generally based on more complex and powerful models for which they attempt to solve specific problems, while Theoretical works would instead attempt to solve problems as efficiently as possible by reducing the power of the model to its minimum (e.g., constant memory, no identification of modules, no synchronization, etc.).

Furthermore, as made visible on Figure 2.1, various research communities are involved in the different approaches:

- Roboticists have the exclusivity of the Bottom-Up approach because of their hardware expertise.
- Top-Down works, while essentially also produced by the robotics community, also counts works from the DNA computing and molecular programming community.

- Lastly, the Theoretical approach is followed both by the DNA computing and molecular programming community and then theoretical computer science community—mostly through the lens of combinatorial geometry and distributed computing.

In this section, we will successively explore the three aforementioned approaches to self-reconfiguration, discussing their leading fundamental models or MSR (summarized on Figure 2.2 below), and introducing the corresponding solutions that have been proposed.

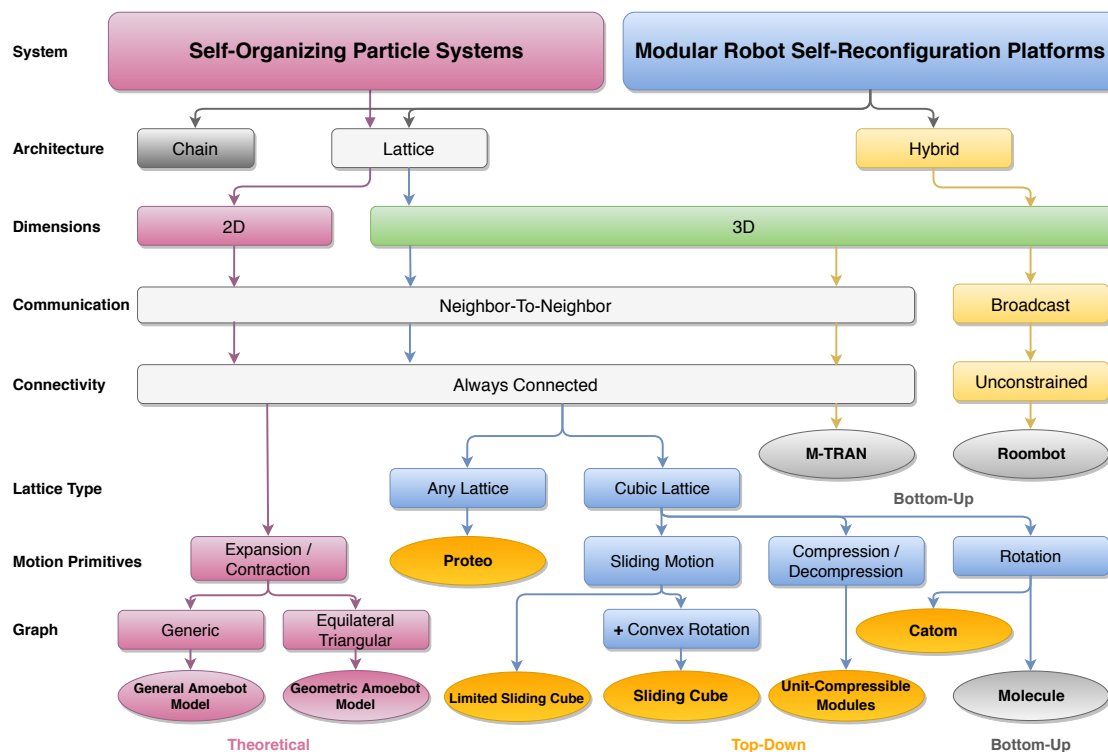


Figure 2.2: Overview of common self-reconfiguration models and select hardware systems.

2.1.1/ BOTTOM-UP APPROACH

Overview As we have just learned, the Bottom-Up approach translates into an initial focus on the modular robotic hardware. Researchers represented in this approach rather unsurprisingly tend to belong to the research community of roboticists. They have come up with numerous module designs, from Unit-Compressible Modules (UCM) like Telecube (Vassilvitskii et al., 2002) and Crystalline (Butler et al., 2003), to hybrids like M-TRAN (Fitch et al., 2013) and Roombot (Spröwitz et al., 2010), the Fracta self-reconfigurable structure (Yoshida et al., 1998), as well as bipartite systems like the Robotic Molecule (Kotay et al., 2000) and I-Cubes (Ünsal et al., 2001a,b). Many more designs can be

found in the literature, but these are the ones that are used in the algorithms concerned by this analysis.



Figure 2.3: A snake-like formation of Roombot modular robots. (Courtesy of Prof Auke Jan Ijspeert, Biorobotics Laboratory, École Polytechnique Fédérale de Lausanne)

This method credibly results in the most difficult self-reconfiguration planning, due to the intricacy of the geometry of hardware modules or their motion capabilities. These systems usually have strong non-holonomic motion constraints, complicating the reconfiguration process as a result. Motion constraints can either be local: induced by the geometry of the modules and by blocking constraints; or they can be global: like the *connectivity constraint* which states that the entire system's graph has to remain connected at all times. Several techniques for achieving holonomy at the cost of the granularity of the system have been devised, through the use of module aggregates with higher holonomy (*meta-modules*) (Ünsal et al., 2001a) or by having the system organized into a porous structure (Kotay et al., 2000) through which modules can flow unconstrained (*a scaffold*). While the kinematics are usually more complex in the *Bottom-Up* approach, modules are likely to assume a wider knowledge of their environment. These environmental facts come from sensor information about their orientation, position in the system, neighborhood, etc. Generally, as in the old saying *knowledge is power*, extensive environmental knowledge in individual modules allows for more straightforward algorithmic solutions, as learning the necessary facts could otherwise require a massive amount of communication.

Hardware-Specific self-reconfiguration Methods In this subsection, we review the hardware specific self-reconfiguration methods proposed in the literature for modular robots residing in 3D lattice environments.

Kotay et al. (2000) proposed a centralized solution for their bipartite Molecule robot based on a hierarchical planner consisting of three levels. Task-level planning stands as the highest level of planning and selects a configuration that suits the task at hand. It then

uses configuration planning to decide on a motion plan for Molecules to transform the initial configuration into the goal one. On the lower level, trajectory planning is used by the configuration planner to move individual modules to their goal position. They also introduced the aforementioned concept of *scaffolding*, to ensure that Molecules would converge into the goal configuration, though it made the granularity of their system extremely high as 54 modules constituted a single scaffold tile.

A similar approach was proposed by Ünsal et al. (2001a) for the I-Cubes bipartite system, consisting of three-degree-of-freedom links used for communication and actuation, and passive cubes for the modules. They used a centralized two-level planner in which the high-level planner decides on the position of modules in the goal configuration by using the low-level planner to search for a feasible plan of individual link motions, that would move the module to the desired location. Several iterative improvements were later made on this approach, by introducing *meta-modules* to simplify planning and therefore adding another layer of planning on top of the existing two, at the meta-module level. Another work with I-Cubes used a centralized divide and conquer approach, where the problem of planning the motion of a module from a position to another was divided into a sequence of local subproblems. They also used a two-level hierarchical planner in this work, where (1) solutions to subproblems were searched on the low-level planner while (2) the high-level planner was concerned with the actual motion of the module, combining solutions from the lower level (Ünsal et al., 2001b).

Although these early works using centralized planning laid the groundwork for much of the field and introduced valuable problem simplification techniques, they are inherently lacking the robustness, scalability, and autonomy that is so critical in self-reconfiguration. Therefore, and as we will see below, researchers eventually turned to decentralized self-reconfiguration method, and thus also had to face the challenges of distributed algorithm design.

Yoshida et al. (1998) proposed a distributed algorithm based on local information for 3D reconfigurable structures with star-shaped modules. They put forward a description of the goal shape using connection types, as previously used in some of their works on 2D hardware. Their approach used local rules with added randomness, in the form of stochastic relaxation based on simulated annealing.

Spröwitz et al. (2010) designed the Roombot hybrid modular robot, which relaxes some of the strong constraints imposed on MSR that greatly complicated planning. Roombots can communicate with other modules through broadcast instead of the traditional neighbor-to-neighbor communication, and does not require the robot to remain connected at all times (which is a major constraint of nearly all other systems), though it requires the presence of a structured ground surface with passive connectors. They proposed a decentralized self-reconfiguration algorithm using meta-modules made of two stacked Roombots,

which guarantees that individual modules can always move. Their approach relies on the locomotion of disconnected structures of Roombot meta-modules that converge into the desired configuration thanks to the attraction of a force field and a predetermined assembly order.

Finally, a number of self-reconfiguration methods for unit-compressible modules—square or cubic (in 3D) modules that can contract and expand on each of their sides— have been proposed. However, MSR made of unit-compressible modules can both be considered specific hardware (e.g., Crystalline in 2D and Telecube in 3D) and a class thereof. We decided to consider the later and cover self-reconfiguration algorithms for these systems under the *Top-Down* approach in Section 2.1.2, as they could potentially be used generically on any future hardware with a similar actuation mechanism.

2.1.2/ TOP-DOWN APPROACH

Algorithmic Conventions and Metrics We will introduce in this section a systematic convention that will be used to compare the further introduced algorithms.

Number of Modules Self-reconfiguration performance is usually evaluated relative to the number of modules in the configuration. Let n be this number.

Resolution We can also introduce a resolution parameter k , for which n is proportional to k^d , with $d = 2$ or $d = 3$ depending on the d -dimensional space. The resolution expresses the size of the modules (or *meta-modules*, in relation to the size of the shape. Therefore a low-resolution configuration would mean that each pixel (or voxel) of the goal shape corresponds to a meta-module made of many individual modules, while in a *full-resolution* configuration each individual module corresponds to a single pixel (or voxel).

Complexity The complexity of reconfiguration algorithms is generally expressed as a number of reconfiguration steps and number of motions. The number of reconfiguration steps is expressed in numbers of time steps. These time steps can be either synchronous, in which case a single time step commonly corresponds to the time required by the rotation of a single module (to which we will refer to as *reconfiguration time* throughout this survey), and where modules are often assumed to perform synchronously; or they can be asynchronous, in which case the duration of a single time step is defined by the authors. On the other hand, the number of motions represents the total number of motions performed by all modules during the entire reconfiguration. Both these complexities are expressed relative to the number of modules in the system n .

Furthermore, the complexity of the total number of messages exchanged during reconfiguration is another notable indicator of how efficient an algorithm performs at the network level, also expressed relative to n .

Additional complexities that could be worth investigating are the *real reconfiguration time*, in seconds, which would require the actuation time of modules to be known, or the *number of CPU operations* (global or per module depending on the underlying architecture).

Architecture Finally, there are number of different architectures that can be used for modular robotic systems. The first main distinction to be made is between centralized systems, in which all computation is performed on a single module of the system or on an external computer, and distributed systems, where the computation is performed distributedly across all modules in the system. If a distributed architecture is in use, the system can either be synchronous or asynchronous, depending on whether the modules rely on a global synchronization of the system to perform their tasks. Furthermore, communication between modules can be either local, in which case modules can only communicate with their immediate neighbor, or global, where any module can communicate with any other module, through unicast or broadcast communications. Finally, memory access can also be local to the module or their immediate neighbors or global to the whole system.

Most of the works that will receive the focus of this chapter assume distributed systems using local communications and memory accesses.

Overview With versatility being a major concern of researchers when designing self-reconfiguration algorithms, the *Top-Down* approach has a crucial role: creating shape formation methods that are not tied to a specific hardware implementation, and that can be applied to various MSR in a generic fashion. Moreover, a software-first approach where algorithms can help point out interesting requirements to include in the hardware platforms. These algorithms usually operate on models with particular kinematic capabilities and constraints that represent classes of robots, as can be observed on the map of usual models and select hardware systems on Figure 2.2.

Due to geometrical and mechanical attributes of robots being more generic, motion planning for these models tends to be slightly less demanding than for the specific hardware platforms found in the *Bottom-Up* approach. Conversely, these models have weaker assumptions about the environmental knowledge of the modules on average, and computation therefore tends to be heavier.

Generic Algorithms A number of self-reconfiguration methods that can be found in the literature are truly independent of any particular hardware implementation whatsoever.

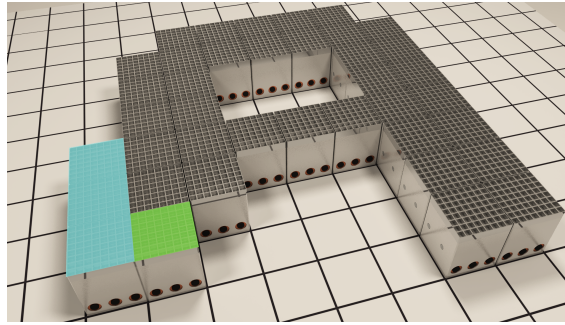


Figure 2.4: A sample configuration of modules from the Sliding-Cube model performing reconfiguration into a 2D A shape. (From the Smart Blocks project (Piranda et al., 2013))

These are the most generic algorithms, either at the level of modular robots in general, or for a particular class thereof as in the following work.

Dewey et al. (2008) designed a system of meta-modules for lattice-based modular robots named *Pixel*, that could considerably simplify reconfiguration planning in massive modular robots. Their main idea is to divide the reconfiguration problem into a planning task and a resource allocation task. The role of the former is to decide what meta-module positions in the goal configuration have to be filled next, and the one of the latter to decide where the meta-modules filling that position should be picked from. They achieve holonomy on their meta-modules by allowing them to be in two states: a *filled* state and an *empty* state. Modules are able to internally flow from a meta-module in the *filled* state to a meta-module in the *empty* state, hence performing a swap, and moving through the structure in predetermined manner—though the fundamental problem of local planning for module flow is not addressed in their paper. The difficulty is thus shifted from actual reconfiguration planning to creating meta-modules that have the desired holonomic features, and designing local rules for internal module flow between meta-modules. Finally, they show that their planner is complete and demonstrates a reconfiguration time that scales linearly with the diameter of the system.

Another approach to fully generic algorithms (for any architecture) can be found in (Fitch et al., 2013), in which the authors use a two-level hierarchical framework where the planning problem is formulated as a distributed *Markov Decision Process* (MDP). An MDP is defined by a 4-tuple $\langle S, A, T, R \rangle$, where: S is the set of *states*, represented by open positions to be filled by modules—which is relative to the number of faces of the modules; A is the set of *actions*, represented by the disconnection of a connector from a neighbor module and the reconnection to another, potentially using a different connector; T is a stochastic or deterministic *transition function* that decides on the next action to perform; R is the expected *reward*, set to -1 as a way to minimize the number of moves. The authors solve this MDP using a distributed implementation of dynamic programming using message passing. The MDP operates on the higher level of the planner, determining for

each mobile module (i.e., that **can** move) on which other module and connector it should attach during the next time step. Then the low-level planner computes the sequence of individual module motions that the moving module should follow in order to disconnect from its current neighbor and reconnect at its new anchor point. Modules search through the structure to ensure that they are not an articulation point of the system's graph to decide whether or not they are mobile—so as to satisfy the connectivity constraint—and lock a portion of it during their motion if mobile. As several modules can lock the same portion of the structure, they can also move in parallel, hence quickening the reconfiguration process. In this scenario, the complexity lies in designing an efficient kinematic planner to act as the transition function T .

Unit-Compressible Modules Butler et al. (2003) generalized their PacMan self-reconfiguration algorithm for 2D unit-compressible modules to 3D systems. An advantage of UCM is that they are able to travel through the volume of the structure, hence potentially benefiting from a higher number of parallel movements, and a shorter distance to their target compared to surface moving modules. In this work, the authors use a technique called *virtual relocation* to move modules from one end of the configuration to the other, swapping their identity with the modules compressing and decompressing along the path to their target position. PacMan is based on a two-stage distributed planning algorithm, wherein: (1) modules locally compute the difference between the current shape and the goal shape in order to decide on which modules should move; (2) a distributed search (depth-first search or deepening iterative search) for a mobile module is performed from the target position, dropping *pellets* along the way to mark the path to be followed by the selected module. A specific actuation protocol is then followed by the modules to make their way through the path without causing deadlocks or disconnections. An interesting aspect of this work is its high parallelism and efficiency, though it requires all modules to have a unique label.

In a similar work, Vassilvitskii et al. (2002) also used an algorithm in two phases, this time to reconfigure systems of Telecube unit-compressible modules. Their method had not only local decision-making, but also some degree of parallelism and completeness. Distributed planning was performed on cubic meta-modules made of eight Telecube modules, with: (1) a path planning phase inspired by the original *PacMan* algorithm for 2D unit-compressible modules with an exponential deepening search to find a mobile module as close to the goal as possible; (2) an execution phase where meta-modules would translate their motion plan into a sequence of individual meta-module motion primitives and execute it. The whole reconfiguration could be performed in worst-case $O(n^2)$ time.

The Proteo Model Introduced by Yim et al. (2001), the *Proteo* model includes constraints on the configuration space of the modules and their movements. Several of the recent works on 3D self-reconfiguration are using models that are different variants of *Proteo*. Metamorphic systems from the *Proteo* class share the following properties: (1) **homogeneity**: all modules share the same electro-mechanical and physical structure—as opposed to heterogeneity, where modules constituting a single MSR can be of various types; (2) **connectivity**: the system must remain connected at all times; (3) **mobility**: each module has motion capabilities; (4) **locality**: only neighbor-to-neighbor communication is allowed and each module is embedded with a local processor. Furthermore, *Proteo* modules only reside in lattice environments, where movements are only allowed from one cell to an adjacent open one, and with the help of a support module acting as a pivot—for rotation or sliding motion. These individual movements are treated as discrete steps. Though it is assumed that connectors between modules are strong enough to support all possible movements and configuration—hence ignoring mechanical structural constraints—a moving module cannot carry another with it. Additionally, another noteworthy aspect of the *Proteo* model is how modules deal with motion constraints. When a motion occurs, it must not result in a collision, nor split the robot into two disconnected structures. While these constraints are characteristic of self-reconfiguration models, it is assumed that *Proteo* modules can proactively sense both local and global violations (through embedded sensors) related to: (1) the movement of their motion pivot which would forbid their subsequent motion; (2) a motion that would result in a collision or deadlock; (3) a motion that would result in a violation of the connectivity constraint. As we will later discuss, these are quite strong assumptions, as preventing collisions, deadlocks, and preserving the connectivity of the systems through communication or coordination greatly hinder motion planning.

In the same paper, Yim et al. proposed a distributed self-reconfiguration algorithm for their class of modules based on local information and a coordination mechanism that they name *goal-ordering*. Two methods for attracting modules to goal positions are put forward. In both of them, the mechanism of *goal ordering* ensures that modules avoid overcrowding around a single goal position by allowing them to reserve one if they satisfy a set of constraints, and implements some coordination mechanisms to help nearby modules get into position. In the *distance-based* method, modules are attracted to the closest unfilled goal position using Euclidean distance, whereas the *heat-based* method uses a heat flow technique with accessible unfilled positions acting as heat sources, and modules not yet in position acting as sinks. Modules climb the gradient by moving towards positions with higher temperature. Furthermore, in order to prevent modules from getting trapped, randomness is used as a temperature tiebreaker and for adding noise to the goal ordering process—by regularly picking the second-best open position. A combination of the two gradient methods is shown to provide the best results, where the

algorithm defaults to the distance-based method and switches to heat-based when stuck. Experiments show that reconfiguration time scales roughly linearly with the number of modules, albeit never converging into the goal shape in some cases, generally because of overcrowding issues.

The Sliding-Cube Model The *Sliding-Cube* model has a lot in common with *Proteo*, from which it differs mainly by an absence of a homogeneity constraint and less powerful embedded sensors. This model has been extensively studied within the context of self-reconfiguration due to its simple kinematics. Modules under the *Sliding-Cube* model can be attached to up to six neighbors using connectors on each of their faces. They are capable of performing sliding motions on the surface of neighbor modules, as well as convex rotations along their edges. In contrast with the *Proteo* model, *Sliding-Cube* modules are assumed to be fitted with sensors that can only sense local information (mutual exclusion and blocking issues), they cannot decide on the violation of the connectivity constraint through the same means. *Sliding-Cube* algorithms can be implemented on varied hardware systems, potentially leveraging meta-modules to achieve cube-like structures with the proper kinematics. Some of the existing compatible hardware systems include UCM such as Telecube (Vassilvitskii et al., 2002) and Crystal (Butler et al., 2003), Molecule (Kotay et al., 2000), hexagonal lattice systems such as Fracta (Yoshida et al., 1998), and the M-TRAN hybrid MSR (Fitch et al., 2013).

This model was first introduced by Fitch et al. (2003), in which they also proposed the *MeltSortGrow* algorithm, that can reconfigure an heterogeneous MSR in an out-of-place manner, through sequential module motions. The algorithm consists of three phases: (1) in the *Melt* phase, the initial configuration is disassembled into a line, used as an intermediate configuration to simplify planning; (2) during the *Sort* phase, the line is sorted according to the type of the heterogeneous modules and their position in the goal configuration. The line is folded in two so as to avoid breaking the connectivity constraint; (3) with the final *Growth* phase, the goal configuration is sequentially assembled from the sorted line configuration, by repeatedly moving the module at the tail of the line into its goal position. Both a centralized and decentralized version of the algorithm were proposed. In the decentralized version, distributed planning is used to find a path for mobile modules to and from the intermediate configuration. A surprising finding they made is that reconfiguration planning for heterogeneous modular robots is not asymptotically harder than homogeneous reconfiguration as previously thought, as they achieved $O(n^2)$ and $O(n^3)$ reconfiguration time for the centralized and decentralized versions, respectively. Clearly, the main problem with this approach—aside from the sequential motion of modules—is the amount of free space that it requires. Therefore, the authors decided to investigate the effect of free space restriction on the self-reconfiguration of heterogeneous *Sliding-Cube* modules. In (Fitch et al., 2007), they introduced the *TunnelSort* algorithm for solving self-

reconfiguration problems among obstacles in $O(n^2)$ time. Modules navigate on the interior of the structure through tunneling and on a one-module thick crust on its surface. This assumption on the presence of a free space crust is abandoned in (Fitch et al., 2005), in the *ConstrainedTunnelSort* algorithm, hence enabling reconfiguration in environments consisting of arbitrary obstacles, forming what they refer to as a *bounding region* around the system. Both algorithms consist of a homogeneous phase, in which modules organize into the desired goal shape independently of their module type, and a module swapping phase, wherein individual modules in the goal configuration are swapped through tunneling in order to obtain the correct type specifications of the goal shape. While *ConstrainedTunnelSort* is not complete, its homogeneous phase shows $O(n^2)$ reconfiguration time and moves and $O(n^4)$ time and moves ($\Theta(n^2)$ in practice for common cases) in the second phase.

Moreover, Zhu et al. (2017) combined Cellular Automata (CA) and Linden-mayer-systems (L-Systems) in order to efficiently reconfigure a *Sliding-Cube* modular robot into a class of goal shapes that can be described by branching structures, in a robust, distributed, and highly parallel manner. The CA rules are used to control the movements of the modules and enable the growth of the goal reconfiguration, from the turtle interpretation of the L-system. The desired structure is grown from an initial seed module, and a new seed module is added at each branching in the growing structure and can initiate the growth of a substructure in parallel. Their approach demonstrates a linear increase in reconfiguration time with the number of modules. As with many other algorithms experimenting with unconventional shape representation techniques, a major drawback of this approach is the difficulty of designing the L-system rules that correctly describe the desired configuration. It is nonetheless a clear example of a self-reconfiguration algorithm that is highly specialized and efficient for a specific class of goal configurations.

Additionally, Støy used unspecified modules with kinematics similar to those of the *Sliding-Cube* in (Støy, 2006) and (Støy et al., 2007). His first approach was to use a two-step approach based on a CA, scaffolding, and attraction gradients. It consists of an off-line preliminary step, wherein CA rules are generated from a Computer Aided Design (CAD) model or mathematical description of the goal shape. This description is first made porous so as to build a scaffold to ease reconfiguration. The self-reconfiguration starts from a seed module controlled by the CA, that attracts wandering modules (i.e., not in goal shape) by the means of attraction gradients for them to descend. Collisions are avoided thanks to the scaffolding structure and a local distributed algorithm is used for connectivity-checking. Their algorithm is convergent and reconfiguration time grows linearly in the number of modules. The main drawback of this method is that it is not systematic: a different cellular automaton needs to be generated for every goal configuration. To circumvent this problem, Støy et al. (2007) proposed to replace the CA-based shape description method by a volume approximation of the goal shape using a set of overlap-

ping bricks of different sizes, while still forming a scaffolding structure. The resolution of the volume approximation is adjustable, with higher resolutions requiring more modules. Local rules are used to replace the CA from their previous algorithm, and need only to be created once, as they are not tied to any particular reconfiguration problem. They show that different kinds of attraction gradients can be used depending on whether one is seeking to minimize time and number of messages or the number of individual module motions.

The Limited Sliding-Cube Model Kawano investigated a version of the *Sliding-Cube* model with increased kinematic constraints where convex rotations are not allowed. He demonstrated both homogeneous and heterogeneous self-reconfiguration of these systems, using algorithms based on local rules and meta-modules that guarantee the preservation of the connectivity and the existence of a mobile module in the structure. In (Kawano, 2015), the author showed that self-reconfiguration using homogeneous *Limited Sliding-Cube* modules could be performed in quadratic time using meta-modules and tunneling motions, even in environments with obstacles. This approach benefits from a high degree of motion parallelism. It is later extended in (Kawano, 2017) for heterogeneous reconfiguration. In (Kawano, 2016), a different approach is proposed for heterogeneous modules, using a compression and decompression mechanisms with virtual walls, which condenses the initial shape at the start of the reconfiguration in order to perform modules swaps through tunneling (for ensuring the proper placement of heterogeneous modules), and expands it into the goal shape at the end. However, this method seems to rely on the assumption that modules have enough force to carry or lift an arbitrary number of modules on one of their faces. Moreover, although the author uses 2×2 meta-modules during part of the reconfiguration to ease permutations by providing motion pivots to sliding modules, the goal shape is not described at the meta-module scale; therefore the granularity of the system is not increased—thus achieving what the author refers to as a *full-resolution* algorithm.

General Cubic-Lattice-Based Works Lengiewicz et al. (2019) tackled reconfiguration by having modules flow through a porous structure with moving boundaries, incorporating interesting aspects from the scaffolding method in (Støy, 2006) and (Støy et al., 2007), as well as the multistage reconfiguration with parallel movements between boundaries from the *PacMan* algorithm (Butler et al., 2003). Their method uses maximum-flow searches to reconfigure massive ensembles of cubic modules on a cubic lattice through a scaffold formed by porous 7-module cubic meta-modules. Their approach decomposes the reconfiguration problem into two partly disjoint subproblems: (1) trajectory planning from the current to the goal configuration (i.e., deciding how the boundaries of the current shape should evolve in order to reach the goal configuration); (2) finding an optimal flow

of modules between the boundaries of the current shape and through its volume. Their algorithm is remarkably efficient as the number of module movements is proportional to the resolution of the robot, that is to say $O(\sqrt[3]{n})$ movements. There are a few caveats in their method, however, that would require additional coordination measures for which they propose several solutions that could be investigated: (1) the existence of virtual modules acting as heat sinks and able to communicate while not physically present; (2) no connectivity preservation mechanism—that they propose to solve using a connection gradient; (3) the reliance on a global streamline planner, for which they propose an asynchronous and distributed version in the same work, which is quite efficient and based on local memory and local communication assumptions only. This work could possibly be extended to any other hardware systems capable of performing internal movements through a scaffold arrangement of these systems.

2.1.3/ THEORETICAL APPROACH

The third approach to the self-reconfiguration problem is led by the theoretical computer science community. They are interested in distributed shape formation and programmable matter in their most fundamental form. Their central question is this: Once most of the constraints regarding the hardware have been stripped out, what can be said about the self-reconfiguration problem, and what sensor information and contextual knowledge is truly indispensable to shape formation?

Predictably, and as displayed on Figure 2.1, this approach operates solely at the software level, developing algorithms to be experimented on using simulation, without concern for the hardware. They also differ from other approaches by the nature of the assumptions that form the basis of their research. Here, researchers are less concerned with the kinematics of the computational units (referred to as *particles*), but are rather interested in the basic abilities of the particles in relation to their own state knowledge. The chief model that has been considered on this particular form of metamorphic systems, named *Self-Organizing Particle Systems* (SOPS), is the *Amoebot* model (Derakhshandeh et al., 2015b; Daymude et al., 2019), inspired by the biological behavior of Amoebae. In its most abstract version, the particles reside on a graph, which represents all the possible positions that a connected set of particles may assume. However, it is the infinite equilateral triangular graph that is generally used in algorithms for practical purpose, where particles are arranged on a 2D triangular lattice. Besides, some important properties of the model must be highlighted. For instance, all particles have constant memory size, modest computational power and do not store any identifier. Furthermore, particles do not have access to global information and all their decision-making happens at the local level, in a synchronous fashion. Particle motions are constrained in a more traditional manner, however, with individual units unable to carry another because of their limited physical

strength. More importantly, all particles in the system are required to form a single connected component at all times—which is equivalent to the connectivity constraint. This motivates the choice of movement capability of the particles, where they perform motion through expansion and contraction primitives, effectively occupying either one or two adjacent graph positions at all times. More complex forms of motions are also introduced, such as *handovers*, where a particle contracts out of a node while another expands into it simultaneously.

While it is true that other theoretical models of Programmable Matter exist besides Self-Organizing Particle Systems as the *Amoebot* model, especially a number of works emerging from the DNA computing and molecular programming communities, we believe that the lack of real reconfigurability of these models implies that they are too far removed from modular robotic systems to be pertinent to this survey. This is because these models are either passive systems that cannot autonomously decide on their motion but are instead preset to reach a desired configuration such as for instance *DNA Tile Assembly* models (Doty, 2012; Patitz, 2014), *Population Protocols* (Angluin et al., 2006), or in the case of *Hybrid Programmable Matter* systems of active robots acting on passive tiles (Gmyr et al., 2019). We may nonetheless note that the *NuBot* active model has been used to perform simple shape formation through self-assembly by Woods et al. (2013), which makes it more pertinent than the aforementioned works to the topic of this survey, and can be of interest to the reader. Furthermore, programmable matter can take the form of programmable self-folding molecules, able to transform into any shape by folding, though these systems resemble more chain-type modular robots than lattice-based modular robots. This has been even realized in hardware (Knaian et al., 2012), therefore shifting this particular work towards the Top-Down class.

Shape Formation in Self-Organizing Particle Systems In this section we will exclusively focus on works concerning the geometric version of the Amoebot model, due to its similarities with lattice-based modular robots. A number of distributed and local algorithms have been proposed for the purpose of shape formation, which will be mentioned below. For simplicity, early works on the problem of shape formation in SOPS have focused on having particles self-organize into primitive shapes. It was first demonstrated by Derakhshandeh et al. (2015b) with the example of a line. The particles had to self-organize to form a line on the triangular grid, with an additional condition: all the particles constituting the line have to be in a contracted state. Their approach assumes the existence of a seed particle (for which the authors also propose a leader-election algorithm in the same paper) that defines the starting point of the line. Other particles organize themselves into a *spanning forest*—essentially a spanning set of disjoint trees. The root of each tree represents a leader that will rotate around the goal shape in a predetermined direction, while followed by all particles in the tree, in a snake-like manner. Trees of par-

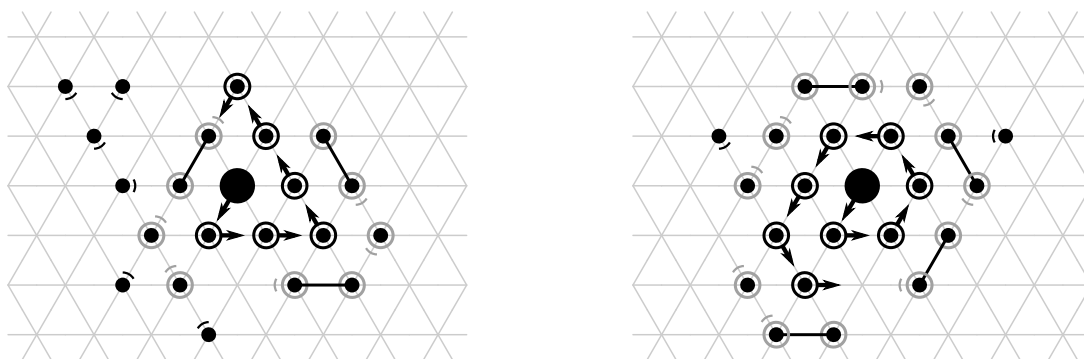


Figure 2.5: Shape formation of a triangle (left) and of a hexagon (right) on a 2D triangular lattice with the Amoebot model. (Courtesy of Prof Andrea Richa, Self-organizing Particle Systems Lab, Arizona State University)

ticles hence rotate around the growing line until they reach one of the ends on each side of the seed, and add themselves to it one at a time. This process is guided by local rules determining the next valid position and be filled, and can reconfigure any connected set of particles into a line with worst-case $O(n)$ rounds (where a round involves a single motion from all particles) and $O(n^2)$ moves (contractions and expansions). The *spanning forest* component used for implementing *leader-follower* motions is one of the fundamental components of all the shape formation works in SOPS mentioned below.

Based on this work, a general framework for shape formation in SOPS was later proposed in (Derakhshandeh et al., 2015a), wherein the authors demonstrate shape formation into scale adjustable triangular and hexagonal structures from any connected set of particles in $O(n^2)$ moves. This algorithm also relies on a leader-follower approach, with particles rotating around the shape being formed (initially only made of the leader particle, or *seed*) in a *snake formation*, until they reach the next position to be filled in the growing shape.

This is later extended to support the formation of any shape, with a general SOPS shape formation framework in (Derakhshandeh et al., 2016). This framework relies on a few assumptions, however: (1) the particles initially form a connected set arranged in a not-necessarily-complete triangle; (2) the goal shape can be described as a constant number of equilateral triangles, whose scale depends on the number of particles in the system; (3) particles know their orientation and move in a clockwise manner; (4) particles can use randomization; (5) particles motions are scheduled in a sequential manner. Under these assumptions, this shape formation algorithm can form any shape using only local information and in a distributed fashion with parallel movements, and in worst-case $O(\sqrt{n})$ rounds. The algorithm first reconfigures the particle system into a line formed by equilateral triangles, each containing a *triangle coordinator* particle, that can direct expansion and contraction motions of the entire triangle, in a way reminiscent of meta-module motions in MSR. The goal shape is then built by a series of triangles motions in a specific assembly order computed using a set of rules.

Since then, another approach to a general SOPS shape formation framework with an equilateral triangle approximation has been proposed by [Di Luna et al. \(2018b\)](#), that relaxes some of the assumptions made in the previous algorithm by [Derakhshandeh et al.](#), albeit with an $O(n^2)$ moves and rounds complexity. An interesting finding is that pre-determining the orientation of movement is not a necessary condition. The authors demonstrate that their algorithm is complete if randomization is allowed, without requiring a specific initial arrangement of particles. Their proposed shape formation method consists of a sequence of seven phases, including spanning forest construction, agreement on the direction of movement, intermediate line formation, and the final goal shape construction phase.

Finally, several problems derived from shape formation have been investigated. Firstly, the problem of forming a shape that achieves the maximum compression for a given set of particles, solved using a stochastic approach based on Markov chains in ([Cannon et al., 2016](#)). Then, the problem of shape recovery, in which particles are assembled in an arbitrary shape and some defective particles in the system must be discarded while maintaining the current shape. It was demonstrated in ([Di Luna et al., 2018a](#)), with a line recovery technique resulting in a scaled-down version of the line after defective particles have been removed.

2.2/ ANALYSIS OF 3D LATTICE SELF-RECONFIGURATION ALGORITHMS

In this section, we examine how the works mentioned in the previous section compare against each other on a variety of aspects, in order to provide a clearer overview of successful methods and their inherent compromises. This discussion is based on Figure 2.6 (which depicts a comparison of modular self-reconfiguration and particle self-organization methods in a tree-style manner), but goes one step further as some characteristics of the works had to be left out of the diagram for the sake of clarity. Furthermore, even though they appear in Figure 2.6, works previously classified as part of the *Bottom-Up* and *Theoretical* approaches are purposely left out of the discussion, due to the impediment to comparison caused by their lack of genericity and the different nature of their underlying system, respectively.

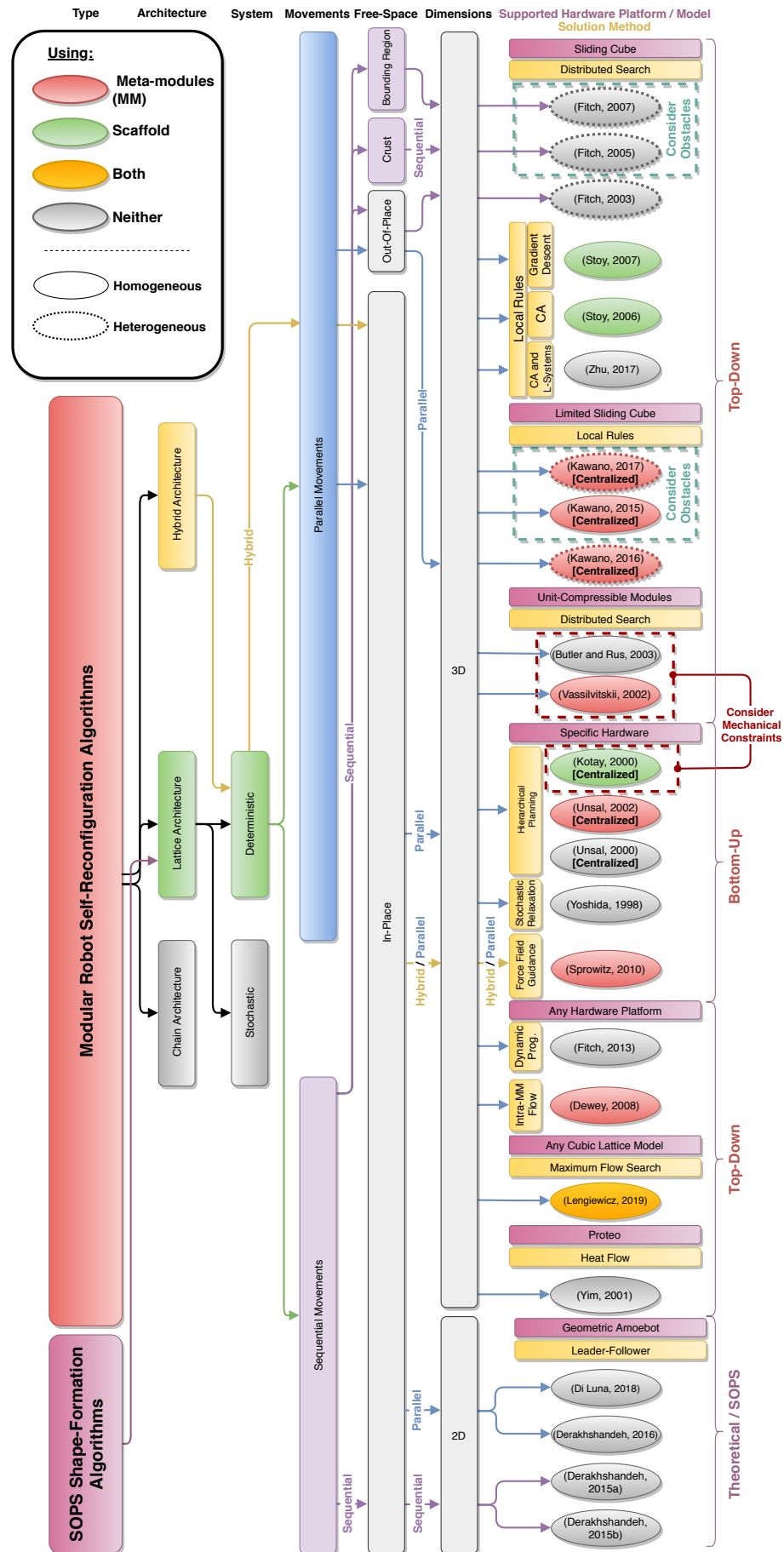


Figure 2.6: Overview of modular robotic and particle system self-reconfiguration methods.

2.2.1/ PLANNING UNDER MECHANICAL CONSTRAINTS

Most of the self-reconfiguration algorithms presented here do not truly take into account physical constraints in their planning, such as those unavoidably imposed by gravity in 3D systems. It is, however a salient requirement of self-reconfiguration algorithms is they are to be realized in actual hardware systems and on a large scale in the future. Some researchers have recently shown interest in this problem. In (Holobut et al., 2017), the authors mention two types of mechanical failures that can occur in a MSR: (1) *loss of stability* due to a shift in the center of mass of the system, which might be caused by the movement of modules; (2) *structural failure* induced by the breaking of a bond between modules due to an excessive load imposed on a connector. The configuration stability problem was mentioned in (Butler et al., 2003), in which the authors state that stability can be insured under a few conditions, and for reconfiguration on a small class of shapes that they call *stem cells*, thanks to UCM traveling through the volume of the structure. Besides, failures due to overstressed connectors are investigated in (Holobut et al., 2017), where the authors present a distributed procedure for predicting if the next reconfiguration step will cause a structural failure. Ideally, procedures of this sort could be added to the planning process of self-reconfiguration algorithms so as to further constrain possible motions to mechanically safe ones exclusively.

2.2.2/ FREE-SPACE REQUIREMENTS AND OBSTACLES

Most algorithms do not explicitly factor in the amount of free space required by the reconfiguration. *Free space* requirement is defined in (Fitch, 2004) as “the total amount of space occupied by intermediate configurations during shape-changing”. Within that context, the most desirable space-related property for a given algorithm is that it can perform *in-place* reconfiguration, which means that it requires no more space than the union of the initial and goal configurations. As can be observed in Figure 2.6, it concerns most algorithms, though a slight degree of tolerance is granted in some cases. It follows naturally that algorithms requiring an arbitrary amount of free space performs *out-of-place* reconfiguration, which is evidently prohibitive to most applications. Some instances of out-of-place reconfigurations are: (Fitch et al., 2003), where a line of length n is used as an intermediate configuration, hence requiring a massive amount of space in a given plane, and (Kawano, 2016) in which the compression and decompression mechanisms inevitably use more space than is desirable. Fitch et al. further investigated self-reconfiguration under free space constraints: (1) in (Fitch et al., 2005), wherein their algorithm only assumes a one-module thick *crust* around the intermediate configurations—which might also help in environments with obstacles; (2) in (Fitch et al., 2007) where the reconfiguration is constrained by an arbitrary shape (a *bounding region*) that blocks the

motion of modules. The latter is analogous to performing reconfiguration among obstacles, a problem which was also studied in (Kawano, 2015) and (Kawano, 2017). The difficulty of motion planning in environments with obstacles naturally comes from the restriction it imposes on module movements, even preventing reconfiguration altogether in some cases.

2.2.3/ COLLISION AND DEADLOCK PREVENTION MECHANISMS

The path planning of modules constitutes an extremely complex problem to solve by itself, but it becomes incredibly more tedious when motion parallelism is considered. Indeed, when considering concurrent module motions, two kinds of additional kinetic constraints appear: movement blocking and deadlock. The former relates to the presence of one module preventing another to move, either while moving or simply by having a disadvantageous position. (This is especially a problem in rotating modules.) The latter can happen when two modules attempt to concurrently enter the same lattice position, and either results in a collision or requires additional coordination measures to be solved. Researchers have come up with several ways to tackle this collision avoidance problem.

A first solution is to use what we name *kinematics simplification methods*. One of these methods is *scaffolding*, which consists in having modules in the configuration organize into a porous structure through which modules can flow to their destination without blocking, at the cost of a higher granularity. An additional consequence of using a scaffolding structure is that hollow, solid, and concave shapes become no harder to build than regular shapes, by suppressing local minima issues thanks to the free passage of modules across the whole system.

Another method aggregates modules into logical units named *meta-modules*. If carefully designed, meta-modules can have holonomic properties that greatly simplify planning. Self-reconfiguration frameworks using meta-modules usually perform motion planning at the meta-module level, and use local rules to realize the transitions between meta-module states at the level of individual modules. Depending on their function, using meta-modules can have a negative impact on granularity, as shape description is also done at the meta-module level: while one voxel normally one voxel equals one module, then one voxel equals one meta-module when they are in use. It is common to have cubic $2 \times 2 \times 2$ meta-modules in 3D algorithms, but they can also be much larger as envisioned in (Dewey et al., 2008). As can be seen on Figure 2.6, the use of scaffolding and meta-modules in self-reconfiguration methods is now commonplace.

A complementary approach is to have modules rely on sensor information—thus assuming the presence of potentially very powerful sensors on modules—or communication to avoid collisions. Modules can either adopt a *proactive* collision solving mechanism, in

which they detect movements that will result in collisions and abort or avoid planning them at all; or a *reactive* mechanism, in which the modules wait for a collision to occur and decide *a posteriori* the way forward. A *proactive* stance involving sensors is often assumed, as in the *Proteo* and *Sliding-Cube* self-reconfiguration algorithms (Yim et al., 2001). Proactive detection through communication is never used in practice, because this process could be extremely costly in terms of time and messages exchanged as querying every module to ensure it is not blocking to the current motion would necessarily result in a complete flooding of the configuration graph on each verification. Reactive deadlock resolution is also avoided, as collisions could potentially put at risk the regularity of the lattice and jeopardize the entire self-reconfiguration.

In hardware models based on rotation-only primitives and surface motion, such as the quasi-spherical *Catom3D* (Piranda et al., 2018), the problem of detecting potentially blocking modules becomes even more essential. Indeed, if sensor detection is not available, ensuring the safe rotation of a module must imply a full traversal of the network to detect potential collisions, which is time- and message-prohibitive. This is an instance of a hardware requirement for relaxing collision avoidance computations, as discussed in Section 2.3.2.

2.2.4/ GOAL-SHAPE REPRESENTATION

Researchers have come up with ingenious ways to represent the goal configuration over the years. In most cases algorithms assume this representation to be globally known by modules, therefore motivating research on efficient shape description techniques so as to avoid overloading their limited memory. In lattice systems, one of the simplest representations of a goal shape is using a grid, which comes at a high memory cost as the size of the representation scales with the number of modules. A shared representation where the description is disseminated across the modules constituting the system could also be considered, but it depends on a number of challenging problems related to data dissemination and retrieval in distributed systems that would need to be solved first (Bourgeois et al., 2016).

Fitch et al. (2013) investigated representing the goal shape as a volume (which they termed *bounding box*) that modules have to fill in order for the reconfiguration to complete. This technique works nicely for convex shapes but requires additional assembly rules for other shapes.

Some works have used description of goal shapes with variable resolutions, where lower resolutions require fewer modules and are thus faster, whereas higher resolution provide a higher level of detail at the cost of longer reconfiguration times and an increased size of the robot. This has been investigated by Støy et al. (2007), through a volume approxi-

mation of a CAD description of the goal shape using overlapping bricks of various sizes. A resolution parameter can be supplied that alters the positioning of the bricks in order to reflect the desired resolution. This approach has the advantage of not having the size of the description increase with the number of modules, but rather with the complexity of the goal shape. It contrasts with a previous work by Støy (2006), where the shape description was embedded into the rules, and whose number would grow linearly in the number of modules. Lengiewicz et al. (2019) also used a variable resolution of the goal shape in their max-flow algorithm, though using a grid representation at the meta-module level.

Though it has not yet been used in a self-reconfiguration algorithm per se, a promising vectorial method for compact shape representation was introduced by Tucci et al. (2017), inspired by a common technique in image synthesis. It uses *Constructive Solid Geometry* (CSG) to describe an object as a tree of primitive geometrical objects, transformations, and set operations, thus having the size of the representation scale with the complexity of the shape and allowing for adjustable resolution at a negligible cost.

Goal shape representation can also be absolutely central to the algorithm, such as in the self-reconfiguration algorithm for branching structures by Zhu et al. (2017), which relies on a recursive description based on L-Systems, defining the goal shape with a rewriting system and formal grammar. A strong advantage of this approach is the compactness of the description, which comes at the expense of a lack of generality of the algorithm, which is narrowly specialized in self-reconfiguration for branching structures.

2.2.5/ SOLUTION METHODS

Solution methods have already been detailed on a case-by-case basis in Section 2.1 for every algorithm covered in this chapter. Nevertheless, it is worth noting that on a higher level, researchers have so far largely relied on three categories of approaches in order to have modules move into position in the goal shape: (1) searching through the configuration for a mobile module or reachable open position while building a motion path; (2) attracting wandering modules to open goal positions through gradient-like techniques; (3) emergent methods based on local rules and CA. An extensive survey of abstraction and solution methods in the context of modular robotics control was offered by Ahmadzadeh et al. (2015) and covers this topic in detail.

2.2.6/ SURFACE MOVEMENTS VS. INTERNAL MOVEMENTS

Two paradigms for module movements exist: surface movements and internal movements—i.e., through the volume of the object. Sometimes a combination of both is used as demonstrated in some *Sliding-Cube* algorithms. Internal movements currently

exist in three flavors which are: (1) compression / decompression with UCM, (2) tunneling, and (3) motion through a scaffold. According to [Rus et al. \(2001\)](#), this second mode of movement is advantageous compared to surface relocation because self-reconfiguration through the volume of the robots generally requires $O(n)$ fewer moves than by surface motions. The literature seems to evidence that internal motions allow for higher degrees of parallelism, at least using scaffolding or unit compression, as it eases the avoidance of motion blocking and collisions. The difficulty then becomes trajectory planning through the volume of the object and avoiding internal collisions. It remains, however, an open question whether metamorphic systems involving a very large number of modules and using internal movements of any sort could be physically realized in practice, as it might turn out to be impracticable to maintain a perfect module alignment for tunneling in massive ensembles, or build modules with sufficient elasticity and connector strength to safely allow motions through a scaffold.

2.2.7/ MOTION PARALLELISM AND CONVERGENCE

As discussed earlier in this chapter, sequential motions are highly prohibitive in medium-to-massive self-reconfiguring ensembles, as they tend to dramatically increase the duration of the reconfiguration process. Yet, it greatly simplifies planning and reduces uncertainty by making deadlocks and collisions virtually impossible. Convergence into a goal shape under these conditions therefore only depends on whether or not a satisfiable assembly order supports the process. We believe that there is an inescapable trade-off to be made between achieving a great degree of motion parallelism and being able to guarantee the convergence of the system into the goal shape. Previous results seem to indicate that uncertainty due to collisions between modules, deadlocks, and local minima increases with the number of modules moving concurrently. Nonetheless, few works were able to truly quantify that effect in relation to their methods, as in [\(Yim et al., 2001\)](#) where a brief discussion on the convergence of the algorithm was provided. In the rest of the literature, the likelihood of convergence of the proposed methods remains unclear. While existing algorithm with parallel motions show a high variance in the number of modules that are able to move concurrently, depending on the method that is being used, reconfiguration works based on scaffolding techniques specifically tailored for massively parallel internal motions show great promise in allowing the parallel and collision-free motion of modules, as [Lengiewicz et al. \(2019\)](#) and [\(Thalamy et al., 2020\)](#) have shown.

The only sequential modular robot self-reconfiguration algorithms covered here were proposed by Fitch et al. in the context of heterogeneous SR. While MeltSortGrow ([Fitch et al., 2003](#)) is truly sequential, *TunnelSort* ([Fitch et al., 2007](#)) and *ConstrainedTunnelSort* ([Fitch et al., 2005](#)) both could easily have modules move in parallel during their tunneling phase.

2.2.8/ ON THE COMPLEXITY OF SELF-RECONFIGURATION

It has been noted in Section 2.1.2 that common metrics for self-reconfiguration algorithms are *reconfiguration time*, usually expressed in number of time steps, and *number of moves*, which counts the total number of individual module motions required to complete the reconfiguration. However, by itself the *number of moves* is not sufficient to get a good sense of the performance of a given algorithm, as it does not convey enough information about the degree of parallelism that it achieves, which is a critical parameter for most reconfigurations on massive MSR. Therefore, reconfiguration time is plausibly the most important metric of self-reconfiguration, as it directly communicates the level of parallelism of the algorithm.

As pointed out in the introduction, researchers are not yet able to find the optimal number of moves or time for a given reconfiguration problem, but as shown in the previous section some algorithms have been able to achieve $O(n^2)$ number of moves both for homogeneous (Kawano, 2015; Vassilvitskii et al., 2002) and heterogeneous (Kawano, 2016, 2017; Fitch et al., 2003) modular robots, and even $O(n)$ moves (Støy, 2006; Lengiewicz et al., 2019), potentially at the cost of a lack of a convergence guarantee as in (Yim et al., 2001). Furthermore, Michail et al. (2017) formally demonstrated that the 2D self-reconfiguration of systems with modules that could only perform rotations was much harder than with modules capable of both rotation and translation. This is evidently also the case for 3D systems.

An additional metric that provides information on the energetic cost of a reconfiguration along with the number of moves is the *number of messages* exchanged during the reconfiguration. It is seemingly of lesser importance than the number of moves as the energetic and time cost of sending a message is nearly negligible compared to the cost of actuation for movements. However intensive communication is often used as a way to simplify the planning process and prevent physical collision and other undesirable events from occurring, which is likely to still cause a massive energetic overhead due to the sheer volume of transmitting messages. Furthermore, because of the immense size of such distributed systems in conjunction with the limited memory size of modules, excessive communications could quickly overload the message queues of the modules and therefore have a dramatic effect on reconfiguration as a consequence of the subsequent message losses (Naz et al., 2018).

2.2.9/ SIMULATION ENVIRONMENTS

Regarding the simulation environment generally used in the field to experimentally evaluate algorithms, we can distinguish three main trends: (i) Authors failing to specify their means of simulation; (ii) Authors developing simulators from scratch for their particular

hardware model; (iii) Authors experimenting through more general-purpose simulators. One exception is [Lengiewicz et al. \(2019\)](#), who performed a numerical simulation through *Wolfram Mathematica*. The second group concerns the majority of aforementioned algorithms, more specifically where Yim, Støy, Rus, or Fitch are one of the co-authors, with simulation environment developed in *Java 3D* as the main underlying technology. Though it might be noted that *SRSim* ([Fitch et al., 2003](#)), developed by Robert Fitch has been used for a variety of self-reconfiguration and locomotion algorithms using the *Sliding-Cube* model and in one case ([Fitch et al., 2013](#)) a model of the M-Tran hardware. Halfway between hardware specificity and general-purpose is *DPRSim*, used in ([Dewey et al., 2008](#)). It includes both a physical and graphical engine suited for millions of modules. Finally, there are a number of general-purpose simulators that have been effectively used in the context of self-reconfiguration even though they have not been used in any of the works represented here. These simulators are *ARGoS* ([Pinciroli et al., 2012](#)), which specializes in swarm robotics and supports a large number of hardware platforms, and *VisibleSim* ([Dhoutaut et al., 2013](#); [Piranda, 2016](#)), a discrete-time step simulator for modular robotic ensembles that has been used extensively to demonstrate 2D self-reconfiguration and other distributed algorithms on a variety of hardware models.

Please refer to Section 1.3.1 of the previous chapter for a more thorough discussion on MSR simulation environments.

2.2.10/ EVALUATION METHODS

In the context of experimental evaluation of algorithms, the disparity does not stop at the simulation environments. Validation methods and self-reconfiguration cases vary greatly across the presented algorithms, therefore making any attempt at comparison cumbersome.

There are two sets of self-reconfiguration instances that are commonly used to evaluate algorithms. The first serves more as a proof-of-concept and consists in the reconfiguration of an initial shape into a chair or table shape ([Støy, 2006](#); [Spröwitz et al., 2010](#); [Vassilvitskii et al., 2002](#); [Fitch et al., 2003](#); [Kawano, 2015, 2017](#)), while the other is a set of reconfiguration problems introduced by [Yim et al. \(2001\)](#) and used in ([Støy, 2006](#)). It involves three reconfiguration cases from a plane into a disk, solid ball, hollow ball, or cup. The initial plane represents a *maximal constraint-free connected overlap* with the goal configuration, providing a maximal overlap between initial and goal shape without blocking constraints on exterior modules. These reconfigurations can be performed at various numbers of modules and cover all possible classes of goal shapes—i.e., convex, concave, solid, and hollow shapes. Finally, works on heterogeneous reconfiguration by Fitch et al. were evaluated with heterogeneous volumes made of a gradient of module types

Work	Proved Correctness	Proved Completeness	Analysed Complexity
(Butler and Rus, 2003)	✓	(p)	✗
(Dewey et al, 2008)	✓	✗	✗
(Fitch et al, 2003)	✓	✓	✓
(Fitch et al, 2005)	✗	(p)	✓
(Fitch et al, 2007)	✓	✗	✓
(Kawano, 2015)	✓	✓	✓
(Kawano, 2016)	✓	✓	✓
(Kawano, 2017)	✓	✓	✓
(Vassilvitskii et al, 2002)	✓	✓	✓

Table 2.1: Summary of complexity analyses and proofs provided in *Top-Down* works. Missing works did not provide any item. (p) means that completeness was only *partially* proven, for a limited class of reconfigurations.

that they would reverse under various environmental constraints (Fitch et al., 2003, 2005, 2007). As discussed previously on the topic of complexity (Section 2.2.8), researchers have been mainly interested in quantifying the number of individual module movements required by the aforementioned reconfigurations, as well as total reconfiguration time; alas, very few works mentioned the number of messages.

In terms of the scale of experiments, most works used hundreds to dozens of hundreds modules. Few works have demonstrated simulations involving thousands to millions of modules. Nonetheless, Dewey et al. (2008) did so by assembling a trumpet made from 1 million meta-modules and a complex building consisting of 10 million meta-modules, from an initial cuboid.

2.2.11/ VALIDATION METHODS AND ANALYSES

Researchers also resort to formal analysis as a way to validate their algorithms, and for further demonstrating the particular capabilities of their methods—under their various assumptions. Ideally, reconfiguration algorithms should be able to demonstrate both *correctness* and *completeness*. The former implies that it will produce a motion plan that reconfigures any initial configuration into any goal configuration (*total correctness*), and is guaranteed to terminate (*total completeness*), while the latter means that any well-formed reconfiguration problem can be performed. We will also use the term *partial completeness* to discuss algorithms which are provably complete only for a limited class of reconfigurations. Table 2.1 summarizes the formal analyses provided by *Top-Down*. It should be noted that most algorithms that could demonstrate completeness were able to do so because they mostly rely on sequential motions at some point during reconfiguration, or

only proved *partial completeness*. Indeed, it is worth noting once again that parallel motions lead to an increased entropy in the reconfiguration process, hence preventing an easy access to provable properties of the algorithms. This is an additional argument for the more thorough experimental evaluation methods that we discuss in next section, as they may be the only way to accurately assess the performance of highly parallel and distributed reconfiguration algorithms.

2.3/ DISCUSSION ON PROGRAMMABLE MATTER

In this section we further discuss the state of the art of self-reconfiguration methods in MSR, but from the vantage point of our vision of programmable matter. We analyze how previous research efforts compare against the specific requirements of MSR-based programmable matter, and what perspectives can be envisioned for the future of the field. In our vision, programmable matter is made of potentially millions of very restricted sub-millimeter micro-electro-mechanical-systems (MEMS) modules, with limited embedded computation used for communication and manipulating actuators, themselves supporting adherence and locomotion (Bourgeois et al., 2016).

2.3.1/ SELF-RECONFIGURATION CRITERIA

There are a number of essential properties that we argue self-reconfiguration algorithms for programmable matter should have:

- **Lattice-Based:** An organization of modules in a lattice appears to be the most advantageous solution, as its highly regular structure allows for an easier positioning of modules thanks to its discretization of the space, compared to a chain arrangement. Yet, maintaining a regular lattice structure on MSR consisting of up to millions of modules might turn out problematic in practice due to repeated imperceptible misalignments and irregularities in the geometry of the modules, seemingly inconsequential problems which would be dramatically magnified by the size of the system. Alternative architectures such as irregular lattices might therefore need to be devised and studied.
- **Distributed:** As self-reconfiguration is a computationally intense process and programmable matter is required to be autonomous, software solutions will need to be distributed in order to be independent from any controlling external entity and mitigate the computational load on individual modules. Furthermore, as previously pointed out, centralized methods do not offer any robustness to failure, a critical aspect of programmable matter as explained below.

- **Homogeneity:** Since programmable matter is generally thought of as massive ensembles of mass-producible and interchangeable modular robotic units, it is evident that reconfiguration frameworks should operate on homogeneous MSR.
- **Motion Parallelism:** With scalability as the main concern of software methods for programmable matter, a high degree of parallelism is required. Algorithms should thus aim to maximize simultaneous module movements. Consequently, the primary performance metric for self-reconfiguration is arguably reconfiguration time, as discussed in the previous discussion on complexity.
- **Reliability:** It has already been noted that there is a compromise to be made between parallelism and ease of convergence into the goal reconfiguration. One can think of extreme examples with, on the one hand, large colonies of ants attempting to build a bridge with a substantial failure rate but with nearly all the agents involved acting concurrently, and on the other hand, slow and steady sequential Lego-like construction tasks where convergence is assured at the cost of a reduced building speed. Building objects made of programmable matter would nevertheless require a high degree of confidence in the success of the reconfiguration. A high fidelity and resolution are therefore important, potentially with a very slight tolerance for misplaced modules in some applications of the technology.
- **Robustness:** Robustness will have to be an essential property of self-reconfiguration algorithms for programmable matter, as faults are almost guaranteed to occur during the reconfiguration of systems comprising millions of individual units.

Besides, the network aspect of the underlying hardware on which is based our vision of programmable matter has to be carefully taken into account, as it has been shown that large lattice-based distributed systems relying exclusively on neighbor-to-neighbor communications are particularly at risk of latency and reliability issues. This is a result of the huge diameter of such systems coupled with a network high average distance, which together pose a serious design challenge to prospective algorithmic solutions (Naz et al., 2018).

2.3.2/ RELEVANCE OF EXISTING WORKS

When confronting these requirements to the works discussed in the earlier sections, it comes to light that existing self-reconfiguration methods have yet to satisfy them all. What seems to stand out from this analysis is that current algorithms are either relying too heavily on the characteristics of specific hardware systems (in the *Bottom-Up* approach), or are making impractical assumptions on the abilities of the underlying hardware, such

as unreasonably powerful sensors or over-simplistic motion primitives (in the *Top-Down* approach).

The relationship between hardware and software poses a number of issues as it is quite challenging to find a common ground between the two, since designing powerful hardware at the micro or nano scale is difficult, while, on the other hand, it is very difficult (if not impossible) to solve problems with hardware that is very primitive. We therefore find that the different communities discussed in this chapter each have a crucial role to play in solving this problem. The theory community is interested in solving problems with the minimum effective hardware, and thus informs other communities of the expected performance of a given task for different hardware capabilities. The robotics community is interested in producing capable hardware at a scale as small as possible, and informs other communities of what can be expected in terms of the capabilities of the models and their mode of motion. Finally, the researchers interested in software methods for these metamorphic systems can compose solutions according to this set of information.

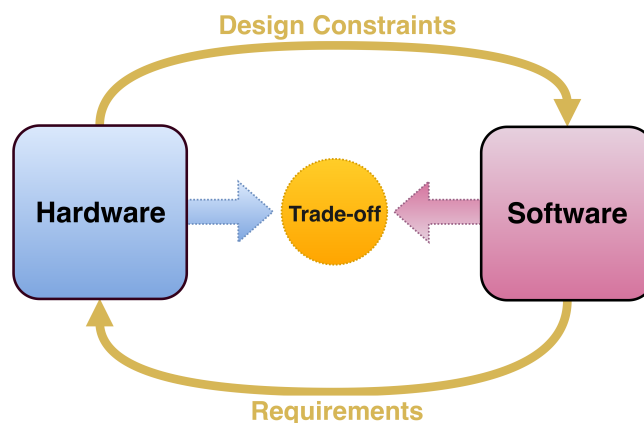


Figure 2.7: Interrelationship of hardware and software programmable matter components.

Thus, we believe that viable solutions to self-reconfiguration in the context of programmable matter will necessarily be the result of a compromise between *Bottom-Up* and *Top-Down* approaches (see Figure 2.7), in that they will need to factor in the design, production, and integration constraints imposed by the hardware, while also meeting the requirements requested by the software. These two mutually dependent classes of pre-requisites will therefore need to converge for practical solutions to emerge.

2.3.3/ PERSPECTIVES

We hint below at some open problems related to modular robot self-reconfiguration that will need to be tackled in the coming years in order to progress towards practical reconfiguration solutions.

Firstly, we suspect that there may be no one-size-fits-all self-reconfiguration solution, where a single method can offer both completeness and near-optimal performance. Perhaps the best approach to self-reconfiguration could involve having a large set of algorithms specifically tailored for a very particular class of reconfiguration problems that are autonomously selected depending on the characteristics of the current problem. If that is the case, designing software methods for classifying reconfiguration problems (based on common features between initial and goal configurations, or lack thereof) as well as for dynamically selecting the best method for solving this problem among a library of methods, will certainly be a challenging issue. This is similar to the concept of *hyper-heuristics*, that defines algorithms for autonomously selecting the most efficient *heuristic* for the current problem, usually using artificial intelligence approaches.

Also, while Yim et al. (2001) have proposed the *maximal constraint-free connected overlap* between initial and goal configurations, additional overlapping patterns could be designed in order to simplify the subsequent reconfiguration process, or optimize for some predetermined metric—e.g., motion parallelism for minimizing reconfiguration time, and number of movements or messages for minimizing the energetic impact.

It has also been pointed out that mechanical constraints—gravity and connector stress in particular—will have to be carefully considered in future reconfiguration methods before large metamorphic systems can be physically realized. These additional constraints would potentially limit the range of possible reconfigurations (depending on the underlying hardware), while also greatly reducing the size of the search space due to impractical intermediate configurations and unsafe module motions. It would be interesting to see how distributed mechanical computation methods such as in (Holobut et al., 2017) could be sensibly integrated into the reconfiguration process, and at what cost.

Then, algorithms with built-in robustness and fault recovery abilities should be further studied, as these are also essential properties required by practical applications of SR. Some approaches might require a pool of additional modules ready to be summoned into the reconfiguration process to replace faulty units (as well as procedures for discarding them), or emergent ways to approximate a desired configuration with a reduced number of resource modules.

Furthermore, though current self-reconfiguration methods already offer attractive performance bounds, solutions often rely on possibly impractical assumptions—e.g., the presence of powerful embedded sensors on modules for collision and deadlock avoidance, scaffold-less tunneling, disregard for low-level planning, or virtual modules communicating with physical modules in the vicinity of unfilled goal positions. Finding out if the current level of performance can be preserved when some of these assumptions are relaxed is essential.

Finally, we find that present experimental evaluations used in the literature are unsatis-

fyng for clearly reporting on the capabilities of the proposed methods (e.g., classes of shapes that can or cannot be formed, scalability, convergence reliability), which makes comparison between methods cumbersome and inaccurate. Therefore, we argue that there is a need for a standardized benchmark for self-reconfiguration algorithms with a common purpose, that covers a wide range of carefully chosen reconfiguration scenarios under varying constraints (and possibly random configurations). We can already identify a number of pitfalls to its design, as self-reconfiguration can be defined in various ways, depending on whether or not the positioning of the goal configuration is constrained, whether an initial overlap between configurations is required, etc. These initial assumptions have a considerable impact on reconfiguration and make for additional parameters to be taken into account.



CONTRIBUTION

INTRODUCTION TO ENGINEERING FASTER SELF-RECONFIGURATION

Contents

3.1 Scaffolding and Structural Engineering	86
3.2 A Dedicated Self-Reconfiguration Platform	88
3.3 Visual Aspect Preservation Through Coating	89

Throughout the first two chapters, we have identified the modular robotic model, constraints, and assumptions under which we consider self-reconfiguration (Chapter 1), and discussed the relevant state of the art of self-reconfiguration (Chapter 2).

Building upon this knowledge, we describe in this chapter how we propose to accelerate the process of self-reconfiguration and attenuate the complexity of module motion planning. This is done by slightly tweaking the parameters and setting of the self-reconfiguration, and by forcing an arrangement of the matter that structurally mitigates concurrency issues.

We will see that this strategy is not suitable for all kinds of context for self-reconfiguration. “*There ain’t no such thing as a free lunch*”, saloon owners and economists would tell you alike, and this is no exception. In this particular case, we are conceding to sacrifice ubiquity and mobility (of the system as a whole), to gain a boost in reconfiguration speeds. This may not be acceptable for many applications of metamorphic robots (such as search and rescue robots), but we believe that for the purpose of object representation and display, it is.

As we have seen in Section 1.2.1, the *3D Catom* model imposes highly restrictive constraints on the motion of the modules, with some that are not even local to the modules themselves. We have discussed the tremendously prohibitive messaging overhead required to clear the **remote blocking conundrum** or the **connectivity constraint** for any motion. While these motion constraint issues are exacerbated by the FCC structure of the lattice and the ensuing combinatorial explosion of a 12-neighbor grid, they are fairly standard among modular robotic models. There is, however, one particular technique that researchers in the field have used to remediate some of these issues and facilitate planning: **scaffolding** — which was introduced in Chapter 2. In this work, we build upon the idea of a scaffolding, and propose a geometry and construction method for an FCC lattice scaffolding, from which we derive very important self-reconfiguration benefits.

As a reminder, scaffolding consists in arranging the internal structure of the modular ensemble as a porous and highly regular scaffold, intentionally reducing the density of the matter so as to ease the internal motion of modules through it. This can be done by removing modules that do not contribute to the overall shape or to the structural strength of the configuration (Støy, 2004). Nonetheless, this idea had only ever been applied to modules in a Square Cubic (SC) lattice (a regular 3D grid), and with holes no larger than one module in size. The best example of such apparatus can be found in the works of Støy (Støy et al., 2004; Støy et al., 2007) (see Figure 3.1). The same scaffold geometry later inspired (Lengiewicz et al., 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.

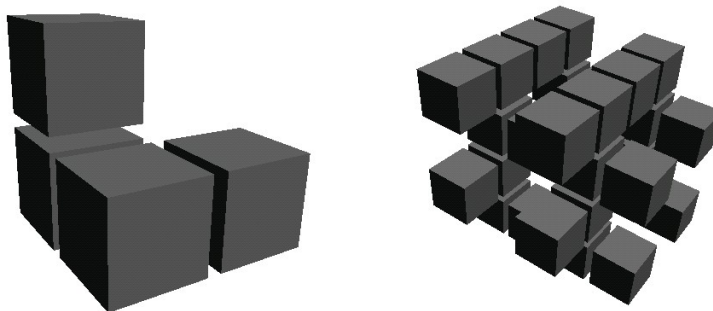


Figure 3.1: Scaffolding structure in a Square Cubic (SC) lattice as proposed by [Stoy et al. \(2004\)](#)

Scaffolding is a powerful tool for self-reconfiguration and presents numerous advantages, which are discussed below.

Motion Support The primary motivation for using a scaffolding structure is that it greatly relieves the burden of concurrent motion planning that has to be done by the programmer of the system, as we will see. But on a higher level, a scaffold also has an impact on the actual possible trajectories of module motions: By lowering the density of the configuration and leaving space inside the forming objects, modules can now flow through the object, while with a higher density only surface motions are possible. This means a different reconfiguration paradigm where not only more paths are available for module motions, but they are also shorter and more direct.

Furthermore, thanks to the regularity and thus to the predictability of the internal structure of the scaffold, the interior of the growing object can be segmented into deterministic and parallel (non-intersecting or at least seldom-intersecting) motion paths through which modules can flow without risk of concurrency issues. This process is named **pipelining**. This transforms the programming challenge from a motion coordination problem to the one of the resource allocation or traffic regulation across motion paths, with the coordination issues between modules now only local to specific locations where motion paths are susceptible to intersect. But this predictability can also directly benefit solving the **remote blocking conundrum**. By increasing the spacing between modules and reducing the number of neighborhood locations of a module that can be occupied, planning motions becomes easier with scaffolding. Careful readers might point out that reducing the number of potentially blocking positions still means having to check the presence of modules in several positions before a motion, which still generates an unreasonable communication overhead — no matter how reduced it is. We propose, however, to remove that communication phase altogether, by artificially reducing the set of motions that a

module can undertake to a number of predetermined motion *paths* between two scaffold positions, where all motions in the path cannot be blocked by a scaffold module — Section 3.1 will cover that further. Finally, scaffolding can also have a positive effect on the other dreadful global motion constraints: the **connectivity constraint**. Once again, this depends on the exact design of the scaffolding structure, which will be discussed later on, but with our scaffold design, every move that is not locally immobilized (by local motion constraints) cannot break the connectivity constraint either. One last thing remains unaddressed by the scaffold at this point is the mutual exclusion problem and motion coordination challenge. On the one hand, mutual exclusion over a particular pipeline or motion path is ensured thanks to a local traffic-light-style motion coordination protocol (Section 4.1.4). On the other hand, mutual exclusions at path intersections are handled by the reconfiguration algorithm itself.

As a summary, Table 3.1 shows the impact of scaffolding on the various motion constraints introduced in Section 1.2.1.

Motion Constraint	Aided by Scaffold
Pivot Constraint	✗
Bridging Constraint (Local)	✗
Bridging Constraint (Global)	✓
Remote Blocking Conundrum	✓
Motion Coordination Challenge	✓
Connectivity Constraint	✓

Table 3.1: Summary of motion constraints that are easier to satisfy in a scaffold setting. (See Section 1.2.1 on motion constraints.)

Reduced Matter Usage Another advantage of scaffolding is that it reduces the number of modules that constitute the target object. However, this also means that scaffold-based self-reconfigurations have also *less modules to displace*, which also contributes to a faster reconfiguration time. Its purpose is therefore twofold, both having a positive impact on reconfiguration time: reducing the amount of matter that must be displaced to perform the reconfiguration, and supporting module motions for an easier displacement and coordination of the remaining modules.

As there are no existing scaffolding models for the FCC lattice and our module geometry, we propose a novel scaffold model that suits our needs in Section 3.1

The Limits of Scaffolding There is, however, a major downside to scaffolding, and it is that it greatly alters the quality of the visual aspect of the represented objects. Indeed, on a structural level using scaffolding for self-reconfiguration can be seen much like audio compression methods: audio compression methods discard information that is not es-

sential to the integrity of the track for the sake of a reduced memory footprint, but it might impact audio quality; scaffolding saves modules and eases processing at the expense of visual quality, by only keeping modules that are mechanically/structurally essential.

Notwithstanding, we believe that this can be mitigated by **coating** the scaffolded object after the reconfiguration with a thin layer of modules, in order to achieve the benefits of scaffold usage at a lower cost to the external aspect of objects. Section 3.3 further explores this idea, and Figure 3.2 illustrates this concept of coating through a side-by-side comparison of regular, scaffold-based, and coated objects.

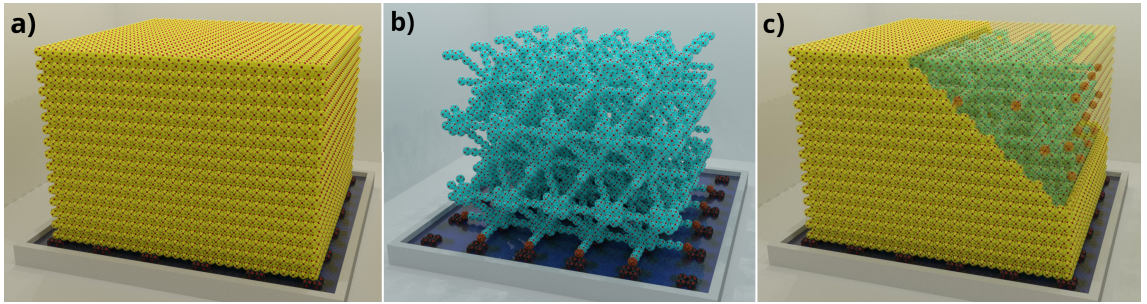


Figure 3.2: Side-by-side comparison of: **(a)** regular cube made of *3D Catoms*; **(b)** scaffold version of the object; **(c)** scaffold cube with added coating.

The Sandbox: Enabling Anisnumeric and Clustered Reconfiguration Virtually all self-reconfiguration problems that have been posed until now have assumed that the initial and the goal configuration had *the same number of modules*, which we will refer to as **isnumeric** reconfiguration. This makes perfect sense for modular robotic applications in unknown and unstructured environment such as exploration or search and rescue missions, or any other applications where versatility/ubiquity is important or where the integrity of the robotic unit has to be maintained. However, in other applications such as our object representation in our case, where self-reconfiguration could be confined to a *dedicated environment*, *anisnumeric* reconfiguration can be considered — reconfigurations where the number of modules in the initial configuration \mathcal{I} and goal configuration \mathcal{G} differ ($|\mathcal{I}| \neq |\mathcal{G}|$). **anisnumeric** reconfiguration thus covers two possible cases in the relation between the number of modules during reconfiguration: **hyponumeric** reconfigurations, where the goal configuration has fewer modules than the initial one ($|\mathcal{I}| > |\mathcal{G}|$), and **hypernumeric** reconfigurations, which is the opposite ($|\mathcal{I}| < |\mathcal{G}|$).

This constraint on the size of the configurations also relates to the connectivity constraints, as attracting new modules coming from outside the system requires some sort of wireless or global communication channel since they are not connected to the configuration. Accordingly, discarding modules from the initial configuration would mean that these modules cannot be summoned back into the reconfiguring robot later without external communication means. Some authors have thus relied on wireless communication

for that purpose, or structured (grid-like) environments supplying energy and used as a global communication bus between modules across the grid (Spröwitz et al., 2014).

In a nutshell, *hyponumeric* reconfiguration thus requires the extra modules can be discarded somewhere near the reconfiguration scene, and *hypernumeric* reconfiguration requires a reserve of modules from which additional modules can be attracted. Both require a means of communication between modules in the configuration and modules that are attracted or discarded.

Nonetheless, it has not yet been pointed out what makes *anisonumeric* reconfiguration attractive in the first place. On a basic level, the fact that modules cannot be discarded or (re)introduced into the configuration at any time means that all modules in the initial configuration will have to move to a goal position in the final configuration, potentially having to traverse the entire configuration (and potentially wait for its turn to do so). Instead, through *anisonumeric* reconfiguration, a module that is too distant to where it is needed might instead be discarded, while another module is introduced right next to the position that needs to be filled. Naturally, this can only be beneficial if modules can be discarded and introduced at various locations, and it has a cost in terms of actual module units and energy. But, as **speed** is our critical parameter, that is a very interesting property.

For these reasons, we develop in this work a framework for *anisonumeric* reconfiguration, in the form of a **dedicated self-reconfiguration platform**, which we name **Sandbox**.

The *Sandbox* consists in a reserve of modules that is located underneath the reconfiguration scene, and that is able to introduce modules at various locations regularly placed on the ground of the reconfiguration scene (cf. Section 3.2).

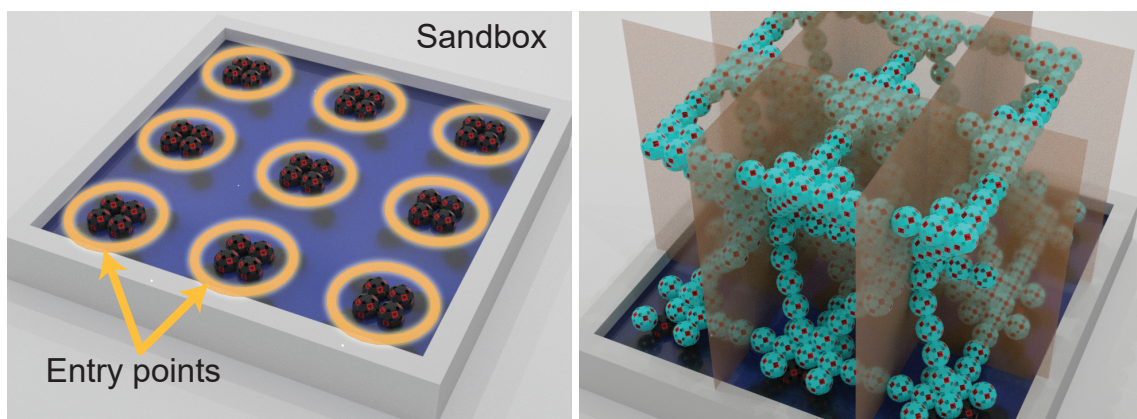


Figure 3.3: **(Left)** Overview of the sandbox, with the entry points used for supplying and discarding modules circled in orange. **(Right)** Scaffold of a cube of side 13 modules over the sandbox. Brown planes divide the object into several vertical areas. Scaffold modules from each area can be supplied exclusively through the sandbox entry points directly below them, enabling *pipelined* reconfiguration.

One major advantage of having these locations for introducing/discarding modules regu-

larly placed on the reconfiguration scene means that the resource allocation concerns of the self-reconfiguration can be segmented or *clustered* into areas of the goal configuration located around each of these *sandbox entry points* (cf. Figure 3.3). In other words, the initial and goal configurations can be discretized into areas that are located around (and above) the entry points, so that modules can self-organize in each area to transform this area from the initial configuration into the matching area from the goal configuration through displacement of configuration modules, feeding of modules of the sandbox, or the discarding of modules to the sandbox, depending on the local problem. Modules from one area almost never have to cross into a nearby area, thus easing coordination and further increasing the predictability of the self-reconfiguration from the point of view of the modules.

Things really become exciting when using scaffolding in conjunction with the sandbox, however, as combining the two together enables unprecedented reconfiguration ease and speeds thanks to a multi-level **pipelining**: pipelining at the level of the shape thanks to the sandbox, where each part of the shape can be constructed in parallel once high-level construction rules are observed; and pipelining at the shape areas, thanks to the dedicated motion paths offered by the scaffold.

Relationship to the Self-Reconfiguration Literature Our approach is somewhat conceptually similar to (Dewey et al., 2008), where modules are arranged into regular multi-module units (*metamodules* as discussed in Chapter 2), which can be in an empty state (only structural modules of the unit), or in 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.

Furthermore, previous scaffolding approaches mentioned in previous paragraphs 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 work. We are also, 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 et al., 2019). Because of this, 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 is 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), as discussed in Chapter 2.

A comprehensive account of the structure (when relevant) of each of the proposed components is provided in the following sections.

3.1/ SCAFFOLDING AND STRUCTURAL ENGINEERING

This section focuses on the anatomy and construction of the scaffolding structure introduced in the introduction.

As a reminder, we aim 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**.

3.1.0.1/ 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 \vec{x} and \vec{y} are classical orthogonal axes but where the vertical axis \vec{z} is skewed and defined as $\vec{z} = (\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, \frac{1}{2})$. 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 3.4a).
- Two horizontal branches placed orthogonally across the \vec{x} and \vec{y} axes named the **X** and **Y** branches (in red and green in Figure 3.4b, respectively).
- Four upward branches ascending at a 45° angle and placed orthogonally to each other: the **Z** branch along the \vec{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 3.4b.
- 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 3.4b).

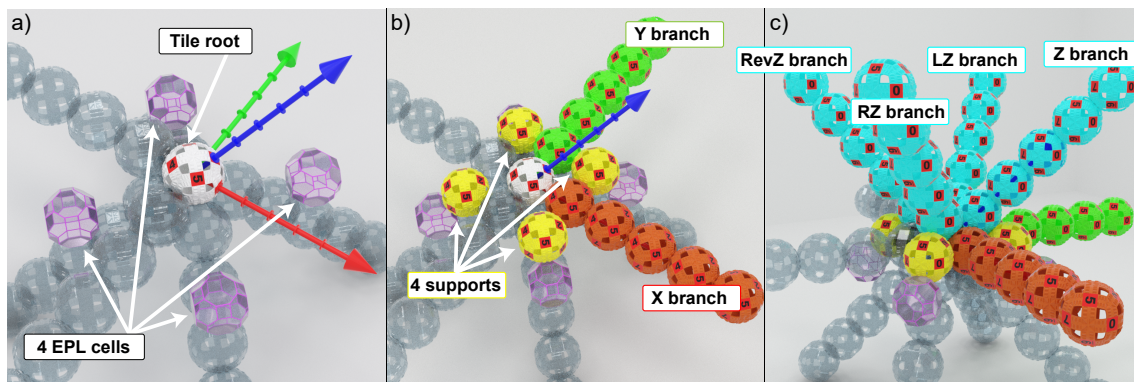


Figure 3.4: **Anatomy of a scaffold tile:** (a) **Tile root** and vertical entry point locations, ingoing branches from parent tiles in transparency; (b) **Supports** and outgoing horizontal branches; (c) Outgoing upward branches.

3.1.0.2/ 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 less dense shapes with fewer modules to place but also might result in a lower fidelity for the details of the shape. Throughout this manuscript, we will assume $b = 6$ as the length of the branches, as it is a very reasonable value mechanically.

When 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 (1 module + the **tile root**) and b modules long. Furthermore, **support** modules need only be present if the upward branch below it 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/ A DEDICATED SELF-RECONFIGURATION PLATFORM

This section briefly discussed the **sandbox** — our dedicated self-reconfiguration platform that supports manages the supply and withdrawal of modules to and from the reconfiguring ensemble. The nature and features have been discussed in the introduction. The design of such system will require considerable future research work, which unfortunately cannot fit into this thesis. Therefore, it is impossible to present the exact design and structure of the sandbox at this point. We have, however, imagined that the sandbox could be 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. Or have this sandbox scaffold connected to a mass of modules that can climb onto the scaffold and into the reconfiguration scene. For the purpose of this work, we presently assume that the top of the sandbox shares the same structure as the scaffold, and consists in fully grown scaffold tiles and branches meeting at the ground level, providing platforms for starting new tiles with 4 incoming branches (and thus feeding paths) to each platform (see Figure 4.1).

This work focuses on the coordinated construction of the scaffold of a shape from an ordered reserve of modules rather than on the transformation of a prebuilt shape into another, which will be further addressed. Therefore, our initial state is an empty reconfiguration scene, and all the modules taking part to the reconfiguration will have to be introduced to the growing shape through one of the ground tiles at the top of the sandbox. The reconfiguring modules will remain connected to the modules from the sandbox at all times (as per the *connectivity constraint*), thus sandbox modules can be attracted onto the reconfiguration scene thanks to messages propagated through the scaffold, in the same way that modules are attracted to various parts of scaffold in our reconfiguration algorithms presented in the next chapter.

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/ VISUAL ASPECT PRESERVATION THROUGH COATING

As stated in the introduction, we propose to compensate the negative impact of scaffolding on the visual aspect of objects by covering the surface of the porous objects formed by the scaffolding with a single layer of modules. We call this process **coating**. This scaffold and coating method can be seen as a special case of self-reconfiguration, that takes place among obstacles (the scaffold, constraining the motions and assembly of modules), and from a reserve of modules,

This section introduces the coating problem and the challenges it poses in a face-centered cubic (FCC) lattice. It shows how a coating can be designed in this context, before we provide a straightforward algorithmic solution in Chapter 5.

While we have found interesting solutions for efficiently constructing the interior of objects, we have yet to implement a coating algorithm that reaches the same level of parallelism as our scaffolding algorithms — the coating is hence the current limiting factor of our method, requiring further research. However, we are interested in this thesis in showing that even with a relatively inefficient coating method, using a coated scaffold may well be preferable to building the equivalent dense shape.

Given a prebuilt scaffold structure made of *3D Catom* modules in a 3D lattice environment and a description of it, coating consists in covering the surface of the shape with *3D Catom* modules such that the object appears solid while taking advantage of the mechanical stability provided by the scaffold itself.

Then, given the geometry of the *3D Catoms*, covering the surface of this scaffold using a single layer of modules would suffice to make the object appear solid. In that context, *solid* means that it would appear to be filled with matter instead of being hollow, hence providing high fidelity to the object that is being represented.

Finally, there are several ways that a coating can be devised for a given scaffold, which relates to the amount of contact between the modules of the surface of the scaffold and those of the coating layer. This relates to the mechanical stability of the object, as the scaffold provides an internal structure to the object that grants its mechanical stability. This can be represented on a spectrum, with a *loose coating* on one end, and a *tight coating* on the other. In that case, a tight coating means that the coating is made such that it fits to the scaffold as closely as possible, and thus provides the highest number of contact points between the surface of the scaffold and the coating layer, which yields to a maximal structural strength. A tight coating is, however, dramatically more difficult to

achieve than the alternatives (intractable even), as it is essentially a case of reconfiguration among obstacles, which greatly constrains the possible assembly order of the coating, as numerous deadlocks could be created by unreachable cells between the growing coating and the scaffold structure itself. On the other hand, a completely loose coating is always at a distance from the scaffolding surface and thus provides no contact points and structural benefits (indeed, the scaffold itself adds no value at all in such case), but greatly relaxes the constraints imposed upon the construction of the coating, as it can be done in isolation from the scaffold. In this work, we propose a middle ground between these two options, based on a loose coating, but with added contact points between the scaffold and the coating layer.

We assume that all modules hold a description of the target shape, and an additional simple lookup engine/function for evaluating whether a position is in the target shape. This description is lightweight and vectorized based on *Constructive Solid Geometry (CSG)*, as first introduced in the context of lattice-based modular robotics in Tucci et al. (2017). Consequently, all modules can deduce if a position is a coating position or a support position.

The definition of the scaffold and coating are thus both derived from a single CSG description of the target shape stored in the memory of modules. A position is considered to be inside the scaffold if its position verifies a set of geometrical rules determining if this position can be a scaffold component, and if that position is within the object described by the CSG, at least at a distance of two lattice cells from its border. Then, a module is in the coating if it is on the border of the CSG object, that is to say if it is inside the object but has a neighbor that is outside of it. This will result in a skeleton formed by the scaffold at the core of the object, surrounded by an empty envelope, and then the coating, thus leaving space between the two (see Figure 3.5.a).

The contact points (or *structural supports*, not to be confused with the *Support* modules in scaffold tiles) are closely linked to the scaffold itself and its b parameter, as the supports are modules resulting from lengthening the external horizontal branches of surface tiles by one module (see Figure 3.5.b, thus closing the gap between the coating and the scaffold at various points of horizontal layers every b modules in height).

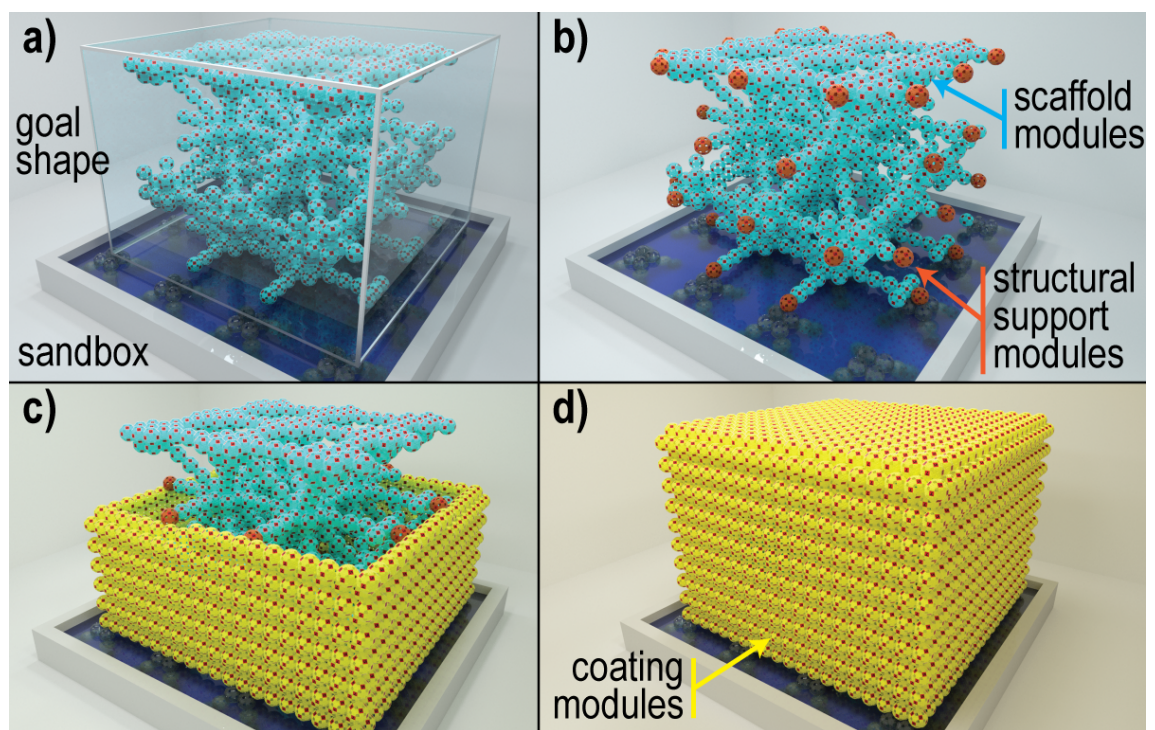


Figure 3.5: **(a)** Scaffold of a cube of size $20 \times 20 \times 20$ modules, with highlighted target volume; **(b)** scaffold with horizontal branches extended into structural supports; **(c)** snapshot of the coating phase; **(d)** fully assembled coating of a cube, with scaffold inside.

SANDBOX AND SCAFFOLD-BASED SELF-RECONFIGURATION ALGORITHMS

Contents

4.1	Fundamentals	95
4.1.1	Scaffold Construction Principles	95
4.1.2	High-level Planning: Tile Construction Scheduling	99
4.1.3	Low-level Planning: Module Navigation	105
4.1.4	Motion Coordination Algorithm	107
4.2	Building Simple Pyramids	109
4.2.1	Motivations	110
4.2.2	Assumptions	110
4.2.3	Self-Reconfiguration	111
4.2.4	Analysis	112
4.2.5	Simulations	118
4.3	Semi-Convex Generalization	121
4.3.1	Motivations and Challenges	122
4.3.2	Updated Model and Assumptions	123
4.3.3	Analyses	124
4.3.4	Simulations	127
4.4	Generalization	133
4.4.1	Motivations and Challenges	134
4.4.2	Updated Assumptions	134
4.4.3	Main Idea	135
4.5	Discussion	136

In the previous chapter, we proposed to solve the self-reconfiguration problem more efficiently by introducing 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 proposed 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 so as to preserve the external aspect of the object. Accordingly, while the object looks 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. We have also introduced the *sandbox*, which is an environment specifically engineered for self-reconfiguration and that enables the parallel addition and subtraction of modules to and from the reconfiguring ensemble during reconfiguration.

The objective of this chapter is to introduce our 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 4.1 starts by introducing the fundamentals of our method, introducing the construction principles of the scaffold and the various algorithmic primitives of our self-reconfiguration method.

Research initiatives typically follow an iterative process and this one is no exception as reaching our current state of advancement has required several iterative versions of our algorithm. The first two versions of our scaffold assembly algorithm have focused on constructing pyramid shapes exclusively, at various scales. Among those, the first version (Thalamy et al., 2019a) had an even simpler model, where all modules were assumed to perform synchronously and in which all modules motions had a similar duration. That way, the coordination of module motions was made a lot easier, and the entire construction of the target shape was deterministic. Then, the second version of our algorithm (Thalamy et al., 2019c) dropped the synchronicity and motion duration constraints, and used a custom distributed motion coordination protocol (see Section 4.1.4) for handling the uncertainty and asynchronicity of the models. Finally, our latest version (Thalamy et al., 2020) generalized results from the previous version to a large subclass of convex shapes.

The synchronized version of this algorithm has been intentionally left out of the document, as it adds no particular insight compared to the asynchronous version which is more realistic. Therefore, Section 4.2 presents the asynchronous and specialized (with regard to square pyramids) version of our algorithm, as a way to familiarize the reader with the concept and mechanisms at play.

We then dive, in Section 4.3, into the partial generalization of the scaffolding algorithm to a well-defined subclass of convex shapes, and demonstrate how it performs theoretically,

and in simulations on various reconfiguration cases.

While the full generalization of the algorithm has only been partially studied, in Section 4.4 we describe the conditions under which our work can be extended to any shape, and the solutions that have been investigated for that purpose.

All algorithms discussed in this chapter are fully decentralized and operate on robotic ensembles where micro-robots act like autonomous agents. As mentioned in previous chapters, all presented simulations were performed in *VisibleSim* (cf. Section 1.3.2).

4.1/ FUNDAMENTALS

4.1.1/ SCAFFOLD CONSTRUCTION PRINCIPLES

Tile Construction Ordering Due to the bridging constraint and the other motion constraints imposed on the modules, a scaffold tile cannot be built in any order¹. There are several rules that must be respected to limit the number of possible intersecting paths and avoid deadlocks during the construction of a tile:

1. The first component of the tile that is placed will always be the **tile root**. This is 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 subsequent order depends on the location of the tile within the scaffold, the **support** modules and X_1/Y_1 modules must be attracted if that particular tile requires them. This has to do with the fact that all these modules attracted to these initial components are at risk of crossing each other's paths during construction. When all of these are in place, the rest of the construction of the tile can proceed entirely in parallel. This phase therefore takes additional coordination measures.
3. Finally, horizontal branches must be built before all vertical branches are grown in order to prioritize the horizontal growth of the shape, for mechanical purpose.

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 4.1.

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 tiles to be built (named **seed tiles**) will be on a corner of the base of the object, and the

¹See youtu.be/DjLwsrzA0MI?t=0 for an example of tile construction, in the case of a full tile.

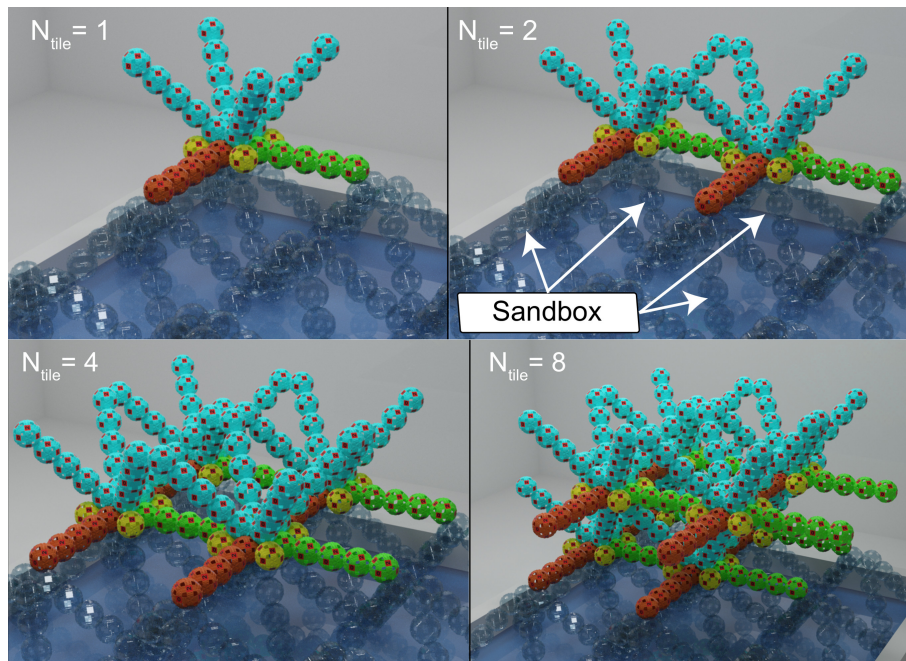


Figure 4.1: **Anatomy of the entire scaffold:** Breakdown of a sample scaffold consisting of an arrangement of 8 tiles with all branches grown, directly over the sandbox (branches from sandbox tiles in transparency).

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 \vec{x} and \vec{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 — hence the other is a **child tile** of the *parent*. A tile usually has more than 1 parent and up to 6 (one for each outgoing branch) if all ingoing branches are grown. Parent tiles are responsible for the growth 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 (see Figure 4.2).

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 4.3, 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

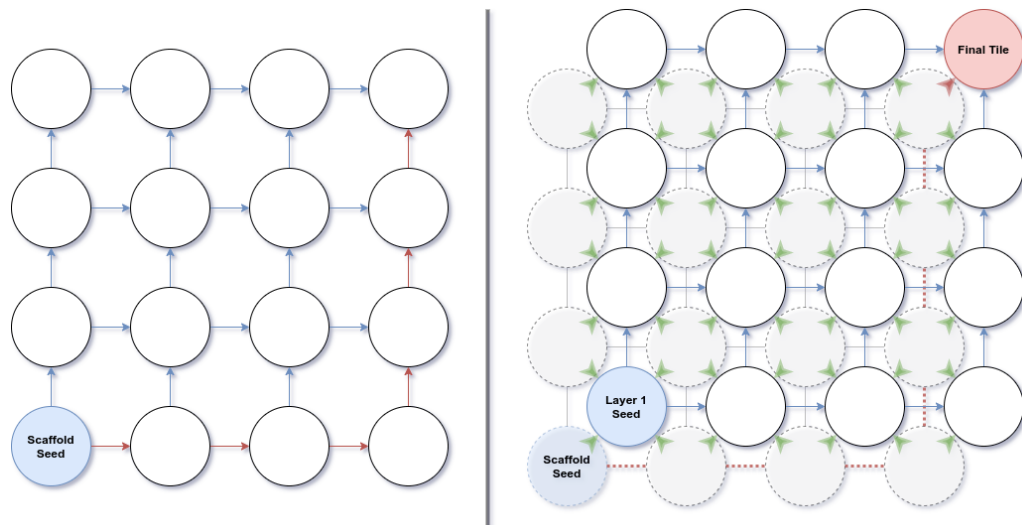


Figure 4.2: Diagram of the **construction polytree** of a $4 \times 4 \times 2$ -tile cube. **(Left)** Bottom tile layer; **(Right)** Top tile layer, with green arrows the edges between the bottom and above layer. Red edges highlight a possible **critical path**.

higher portion of the tile.

Any module entering a tile will do so from one of the four EPL, that is to say, 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 strictly within a particular tile during its construction. What motivates this mode of operation is that it severely limits 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.

Note that for the generalization of this construction methods to all morphologies of scaffolds, a topic that Section 4.4 will touch on, it is possible that tiles are fed through horizontal branches, but this can only happen if they have no ingoing vertical branches because of concavities in the shape of the scaffold.

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

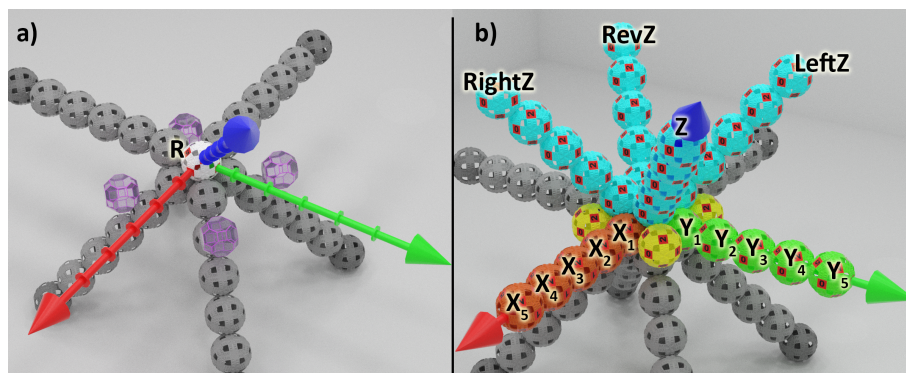


Figure 4.3: Reminder of the anatomy of a scaffold tile. **Entry Point Locations (EPL)** in pink on the left image.

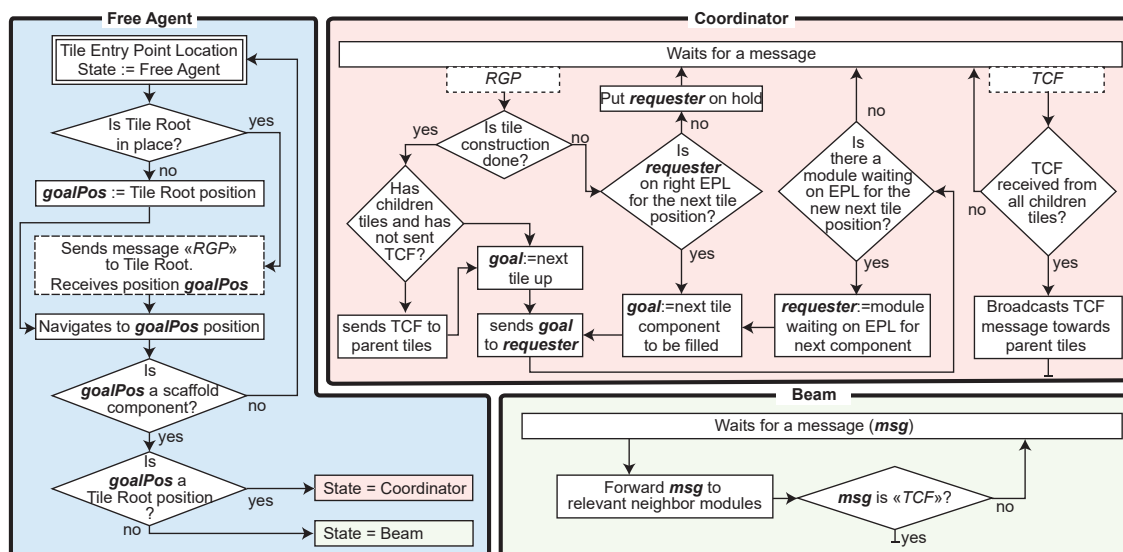


Figure 4.4: **Simplified view of the behavior of each module state and transitions between them.**

be synthesized over the course of this section. Figure 4.4 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 the modules do not just appear from nowhere, but are already within the system as *Idle* modules.
- **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 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.

4.1.2/ HIGH-LEVEL PLANNING: TILE CONSTRUCTION SCHEDULING

Our proposed self-reconfiguration planning process² operates at two levels. 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. This can also be seen in Figure 4.2 as the dependencies between nodes of the construction poly-tree are always from left to right and from the bottom up. 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.

The growth of the scaffold will, therefore, start from either a single ground position (a single 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,

²Please refer to the following video for a walk-through of a reconfiguration into a cube: youtu.be/DjLwsrzA0MI?t=36.

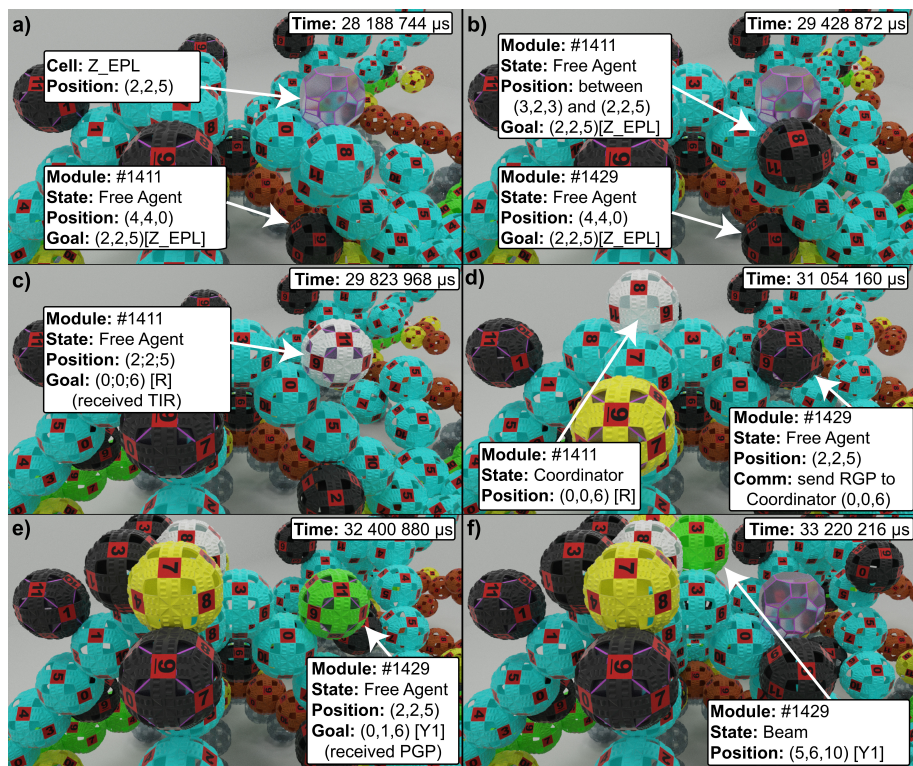


Figure 4.5: **Simulation snapshots of the *Free Agent* goal assignment process.** Two *Free Agents* (#1411 and #1429 drawn in black) climb up to the *Z_EPL* cell and get assigned their position in the future tile: *Tile root* for #1411 through a TIR message as the tile was missing its *Coordinator* (in 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.

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.

Figure 4.6 shows another example of the resulting polytree of a shape. This shape has a concave ground layer and requires two seed tiles that can begin the construction in parallel until the construction reaches tiles that are dependent on parent tiles from both construction processes, at which point both construction have to synchronize. This is done seamlessly as the construction of the *merge tile* can only start once all dependencies (presence of ingoing branches from parent tiles) are met.

The construction of a tile begins when a *Free Agent* module arrives at one of the EPL of the future tile (see Figure 4.5a, b & c), and claims the empty *tile root* position (Figure 4.5d). 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 4.5d & e) in order to build the various branches and *tile supports* it needs (see Figure 4.5f). By default, when a module ar-

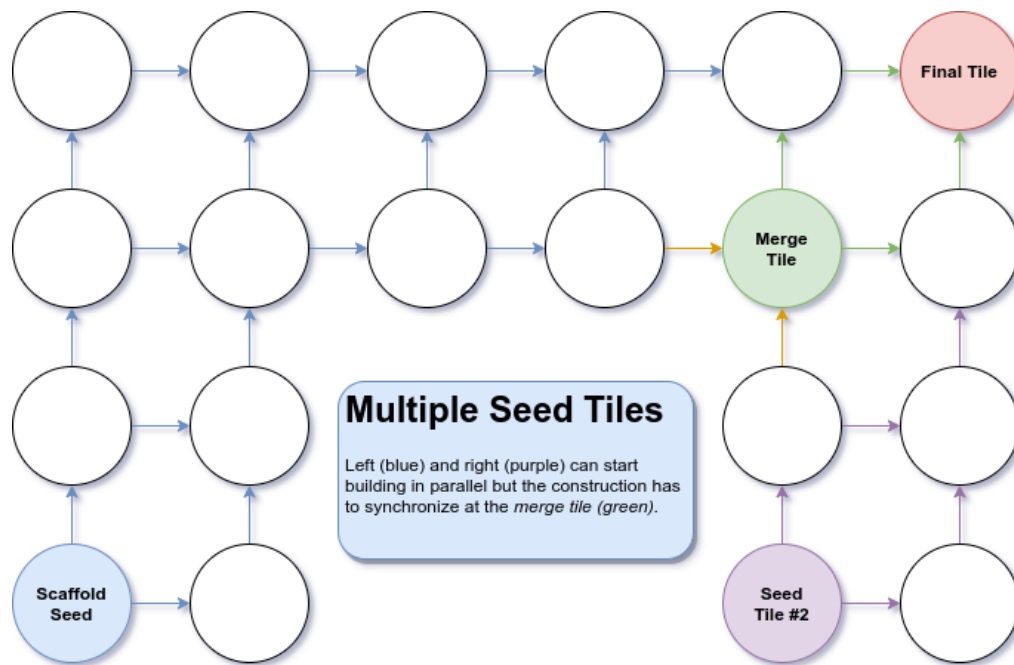


Figure 4.6: Diagram of the **construction polytree** of the ground layer of a shape with concavities at the ground level, requiring two seed tiles. Blue edges show dependencies from the *scaffold seed*; purple edges from the second seed tile; green edges from the merge tile once both processes are synchronized, and orange edges highlight the two edges that cause the synchronization. Either the blue process reaches the merge tile first and it has to wait for the purple process, or the other way around.

rives 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 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) [\emptyset]: 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) [\emptyset]: In some cases, arriving modules might send RGP to the tile before the *tile root* has taken its position. In such 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) [\emptyset]: 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 bringing the module that will claim that location. More precisely, every branch or *support* has a preferred feeding EPL by default: the EPL directly 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 Il. 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 Il. 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 priority in the construction order of tiles is respected, by enforcing synchronization points between concurrently growing portions of the goal shape.

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 4.4 also summarizes the high-level reconfiguration process.

Influence of Target Shape Placement and Orientation As we have seen throughout this section, our algorithm is not symmetrical with regard to the shape. In other words, we enforce a direction to the growth of the scaffold and have a criterion based on coordinates to determine the seed tiles, the set of seed tiles and thus the construction process will differ depending on the orientation of the target shape with regard to the sandbox.

Furthermore, as the sandbox consists in discrete entry points for supplying modules from the reserve, the construction process will also be influenced by the placement of the target shape on the scaffold. While we chose in the following experiments to align the front-left

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;

```

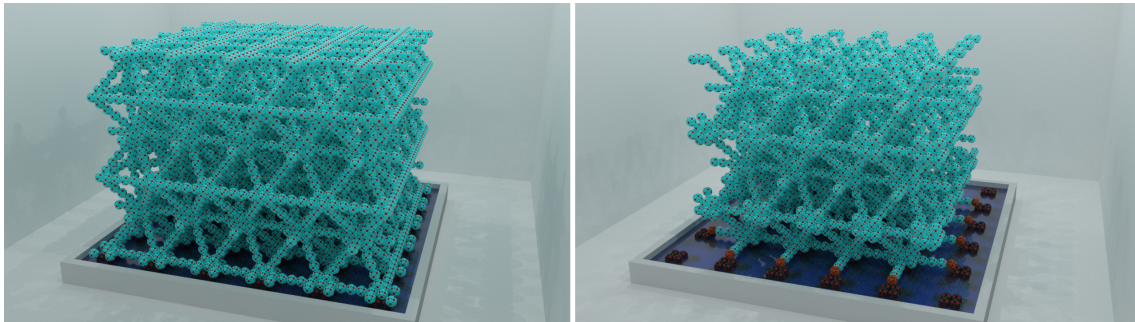


Figure 4.7: Visual comparison between: **(a)** a scaffold cube with its front-left corner aligned on an entry point, and **(b)** the same cube with its center aligned on an entry point. (This cube is actually smaller by one module in each direction to accommodate the coating.) The centering difference is most noticeable at the ground level.

corner of the target shape (the scaffold seed) with a sandbox entry point, other experiments on coating might assume a sandbox-centered target shape. The resulting scaffold is, therefore, different depending on the centering of the target objects on the sandbox (see Figure 4.7). Surely, this has an impact on reconfiguration time that might be interesting to quantify. Nonetheless, these goal shape orientation and position parameters do not influence the order of the performance of our method.

4.1.3/ LOW-LEVEL PLANNING: MODULE NAVIGATION

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 planning 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 $\langle \text{neighborhood}_{\text{bin}}, EPL, \text{destination}, \text{step} \rangle$ to a displacement vector \overrightarrow{disp} , where each element corresponds to:

- $\text{neighborhood}_{\text{bin}}$: 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 II. 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.

4.1.4/ 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. (2016a). 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 perform a motion towards it and turns into the *orange-light* state and defers 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) [\emptyset]: 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.

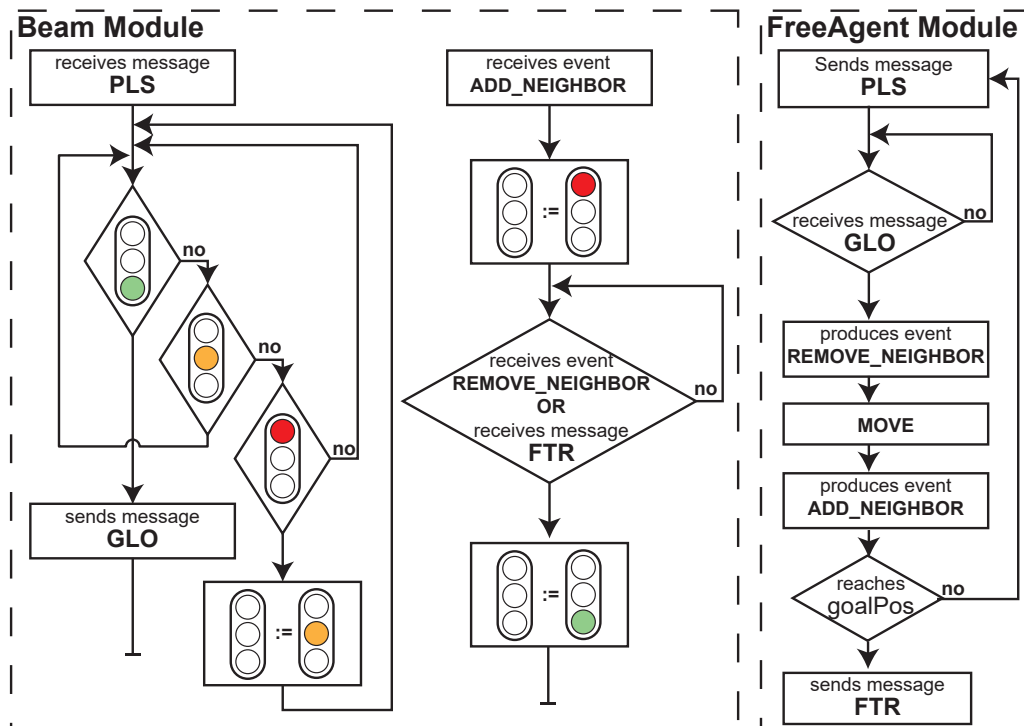


Figure 4.8: **Light state transition diagram.** The two *Beam* routines are executed concurrently on pivot modules.

The transition between these states is not only assured via messaging, as modules also monitor their interfaces to react to any *connection* or *disconnection* event and update their state accordingly. Thus, if a *Beam* module 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 the *green-light* state (see Algorithm 3, ll. 6–7). These mechanisms are summarized in Figure 4.8 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.

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     |   |   |   sendMessage(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();

```

4.2/ 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 below.

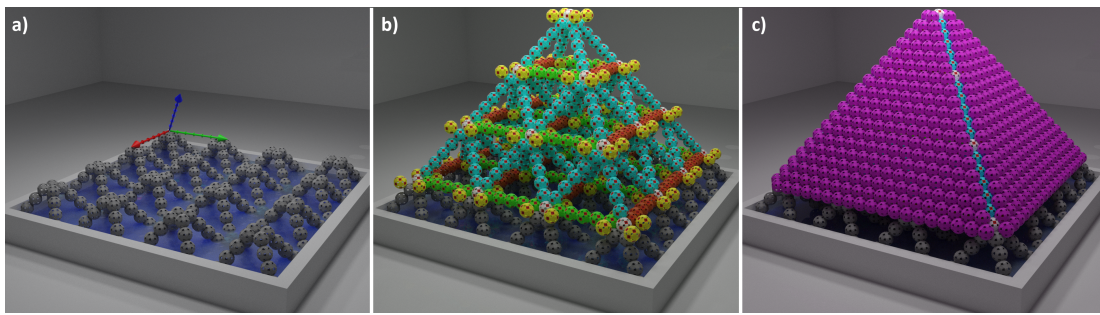


Figure 4.9: Construction of a 3D model of a pyramid using scaffolding ($b = 6$). **(a)** Support structure; **(b)** Scaffold of the 4-pyramid; **(c)** Envisioned coated 4-pyramid, after removal of support modules (differs from the actual implemented coating method).

4.2.1/ 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.

4.2.2/ 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.

4.2.3/ SELF-RECONFIGURATION

The self-reconfiguration proceeds exactly as explained in Section 4.1, 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.

Surplus Modules Management There are three possible variants of the algorithm that can be used. In the first one, named *Continuous Flow Algorithm*, 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. As an analogy, in this particular variant the sandbox is can be thought of as an open tap of modules that only stops once the target shape is filled. The number 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 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 main version that will appear in the following experiments and generalizations, as it is the

most efficient both in terms of times or number of modules compared to the other two variants. Returning to our tap analogy, this variant also corresponds to an open tap, but where the flow is interrupted once enough modules have been injected into the reconfiguration scene.

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.

There is a third variant, named *request-based feeding* that was present in the first synchronous version of our method (Thalamy et al., 2019a) mentioned earlier, which also only used the right number of modules, but that would only attract them from the sandbox on request by the coordinator of the tile that needed them. In this scenario, whenever a tile root would arrive at a new tile that must be constructed, it would compute its requirements by matching the different ingoing branches it has to the needs of its outgoing branches, deducing for each ingoing branch how many modules are needed and when (which was only made possible by the synchronization assumption). By contrast, the sandbox does not act as an always open tap, but rather a tap that gets turned on only on request and for a predetermined duration. These messages would be sent down each of the ingoing vertical branches and then be relayed by the tile coordinators to the tiles below the requester tile, until reaching the sandbox tiles directly under the requester tile, which are then responsible for calling modules from the sandbox at the right time and supplying them to the requester. This corresponds to an additional message type named `INITIATE_FEEDING (IF)`, found only in this variant:

MESSAGE_NAME (ACRONYM) [DATA]

INITIATE_FEEDING: (IF) [{*RequestedTileComponents*}] : Sent by a freshly arrived *Coordinator* down all of its incident vertical branches to express its resource requirements to 4 lower-level *Coordinators* connected to the sandbox—i.e., how many modules it needs for building its tile.

In this variant, a tile would therefore have to wait some time for the requested modules to arrive before its construction can proceed, and it is thus slower than the other two variants with a continuous feeding of modules. This waiting time is proportional to the height of the tile in the scaffold, which will be reflected in the reconfiguration time presented in the next section.

4.2.4/ 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).

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 manuscript, 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 $N_{\text{tiles}}(i)$ denote the number of tiles at tile layer i , with tile layer 0 as the base of the object. We have:

$$N_{\text{tiles}}(i) = (h - i)^2 \quad (4.1)$$

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

$$N_{\text{tiles}} = \sum_{i=0}^{h-1} N_{\text{tiles}}(i) = h^3 - 2h^2 + h \quad (4.2)$$

Then let $N_{\text{modules}}(i)$ 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:

$$\begin{aligned} N_{\text{modules}}(i) = & (h - i) [(h - i - 1)b + 1 + (h - i - 1)(b - 1)] \\ & + 4(b - 1)(h - i - 1)^2 + 4(h - i)^2 \end{aligned} \quad (4.3)$$

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

$$\begin{aligned} N_{\text{modules}} &= \sum_{i=1}^h N_{\text{modules}}(i) \\ &= (2b - \frac{1}{3})h^3 + (\frac{9}{2} - 2b)h^2 + \frac{5}{6}h \end{aligned} \quad (4.4)$$

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

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

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.

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 \vec{x} axis border of the base between $(0, 0, 0)$ and $((h - 1)b, 0, 0)$ positions, followed by the opposite \vec{y} axis border of the base between $((h - 1)b, 0, 0)$ and $((h - 1)b, (h - 1)b, 0)$ positions, and up the backward edge to the top tile of the pyramid between $((h - 1)b, (h - 1)b, 0)$ and $(0, 0, (h - 1)b)$ positions; or along the \vec{y} axis border first, and then the opposite \vec{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 along 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 one in 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 time steps 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:

$$T_{\text{tile}} = 16(b - 1) \quad (4.5)$$

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

$$T = \sum_{i=1}^{(h-1)} 16b - 16 = 16(b - 1)(h - 1) \quad (4.6)$$

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.*

Proof. Using Equation 4.4, 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 4.6, 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. \square

Request-based Feeding Complexity Similarly, we find that for the *request-based feeding* variant, the time to build a given tile can be expressed as:

$$T_{\text{tile}}^{\text{rb}}(i) = [24 + 6b + 2b(i - 1)] \times ts = [24 + 4b + 2b \times i] \times ts \quad (4.7)$$

Theorem 3. *The reconfiguration time of the reconfiguration of the h -pyramid is $O(N^{\frac{2}{3}})$ for the request-based feeding variant.*

Proof. Using Equation 4.7, we can express the time required to construct the h^{th} level of the h -pyramid in the *request-based feeding* variant in number of motion times as:

$$T^{\text{rb}} = \sum_{i=1}^h 24 + 4b + 2b \times i = 24h + b(5h + h^2) \quad (4.8)$$

We conclude that the reconfiguration time is $O(h^2)$ time steps. Using Equation 4.4, 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 4.8, we deduce bounds for the motion time T^{rb} :

$$6\left(\frac{N}{q}\right)^{\frac{2}{3}} + 54\left(\frac{N}{q}\right)^{\frac{1}{3}} < T^{\text{rb}} < 6\left(\frac{N}{p}\right)^{\frac{2}{3}} + 54\left(\frac{N}{p}\right)^{\frac{1}{3}}$$

We conclude that the reconfiguration time is $O(N^{\frac{2}{3}})$ and $O(h^2)$ time steps in the case of the *request-based feeding* variant. \square

This $O(h^2)$ reconfiguration confirms our previous intuition that the reconfiguration time would be impacted by a factor of h as the waiting time for requested modules is a factor of the height of the requesting tile and thus a factor of h .

Message Complexity An analysis of message complexity is presented below, for the messages pertaining to the high-level construction process exclusively (see message list in Section 4.1.2) — motion coordination messages are thus excluded.

Theorem 4. *The complexity of the number of messages $N_{\text{messages}}^{\text{high}}$ sent to schedule the construction of a N modules pyramid is $O(N_{\text{modules}}^{\frac{4}{3}})$.*

Proof. Each module sends 4 kinds of messages during a reconfiguration: $N_{\text{messages}}^{\text{high}} = N_{\text{RGP}} + N_{\text{PGP}} + N_{\text{TIR}} + N_{\text{IF}} + N_{\text{IBR}} + N_{\text{CR}} + N_{\text{TCF}}$. Where *RGP*, *PGP*, *TIR*, *IF*, *IBR*, *TCF*

and CR messages denote the messages detailed in Section 4.1.2. In the worst case, $N_{TIR} = c \times \sum_{i=1}^h i^2 = O(N_{\text{tiles}})$, c is a small constant. This is because TIR is sent at least once per tile, to at most the 4 ingoing ascending branches to that tile, and to a distance of at most 3 hops (from the tile's coordinator to the EPL).

Similarly, and in the worst case, $N_{IBR} = k \times \sum_{i=1}^h i^2 = O(N_{\text{tiles}})$, k is another small constant. IBR is sent a maximum of 6×6 times per tile (once from each ingoing branch to every other ingoing branch), and to a maximum hop distance of 2.

N_{CR} is also sent a maximum of 4 times per tile (once to each ingoing vertical branch) by the coordinator, to a maximum distance of 3 hops to reach all 4 EPLs. We have, therefore, $N_{CR} = O(N_{\text{tiles}})$.

The number m of IF messages sent by a module depends on the level i of its docking tile: $m = 4b \times (h - i)$. Then,

$$N_{IF} = \sum_{i=1}^h i^2 (4b \times (h - i)) = \frac{4b}{12} (h - 1) h^2 (h + 1)$$

As for N_{TCF} , which propagates a single message from the leaves of the construction polytree to its roots, through each tile's parents, which are up to 6, we have in the worst case $N_{TCF} = 6 \times O(N_{\text{tiles}})$.

Furthermore, messages RGP and PGP are sent $u = 3$ or $u = 4$ times every time a *Free Agent* module enters a tile, except if it will become root, therefore using Equation 4.3,

$$N_{RGP} = N_{PGP} = u \times \sum_{i=1}^h ((N_{\text{modules}}(i) - i^2) \times i)$$

As N_{IF} , N_{RGP} , and N_{PGP} are $O(h^4)$ and $N_{IBR} = N_{CR} = N_{TIR} = N_{TCF} = O(h^3)$, we can deduce as in the previous proofs that $N_{\text{messages}}^{\text{high}}$ is $O(h^4)$ and $O(N_{\text{modules}}^{\frac{4}{3}})$, regardless of the variant being used (which only adds or removes N_{TIR} from $N_{\text{messages}}^{\text{high}}$). \square

Though we do not provide a thorough mathematical analysis of the messaging aspect of the low-level planning process (see Section 4.1.4), we provide the following insights on the topic:

- The number of messages $N_{\text{messages}}^{\text{low}}$ from the low-level process is proportional to the number of motions N_{motions} that occurred during the reconfiguration, as it is only the intention of motion from a module or the end of a pre-approved motion that generates low-level planning message traffic.
- *PLS* and *GLO* are only sent locally within a distance reachable by the module seeking to move or the pivot granting its approval, therefore it is sent only a constant

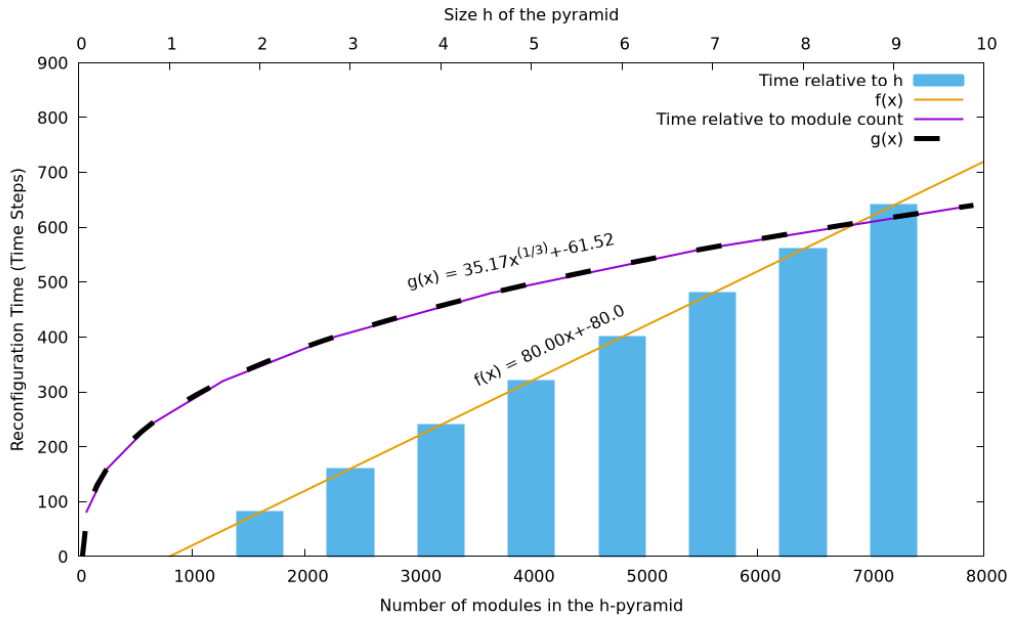


Figure 4.10: Reconfiguration time relative to tile count and module count for increasing sizes of h -pyramid

number of times per motion.

- FTR is only sent at most once per module motion to the *light pivot* of the mobile module.
- Therefore, $N_{\text{messages}}^{\text{low}}$ is likely of the form $N_{\text{messages}}^{\text{low}} = N_{\text{motions}} \times (N_{\text{GLO}}^{\text{motion}} + N_{\text{PLS}}^{\text{motion}} + N_{\text{FTR}}^{\text{motion}}) = N_{\text{motions}} \times c = O(N_{\text{motions}})$, where N_{motion} concern a single motion and c is a constant.

4.2.5/ SIMULATIONS

Reconfiguration Time 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 4.10 shows the simulation results, to which we have added a plot of the fit of both curves, confirming that reconfiguration time is indeed linear in $O(h)$ and $O(N^{\frac{1}{3}})$ for the h -pyramid.

Variants Comparison and Random Motion Duration Experiments We study the following aspects of our work below:

- Compare the *continuous flow* (with surplus) asynchronous and *request-based feeding* synchronous variants of our self-reconfiguration algorithms with varying pyramid scaffold sizes. We focus on studying the impact of the *continuous flow* algorithm in terms of excess modules count and total reconfiguration time. We measure the

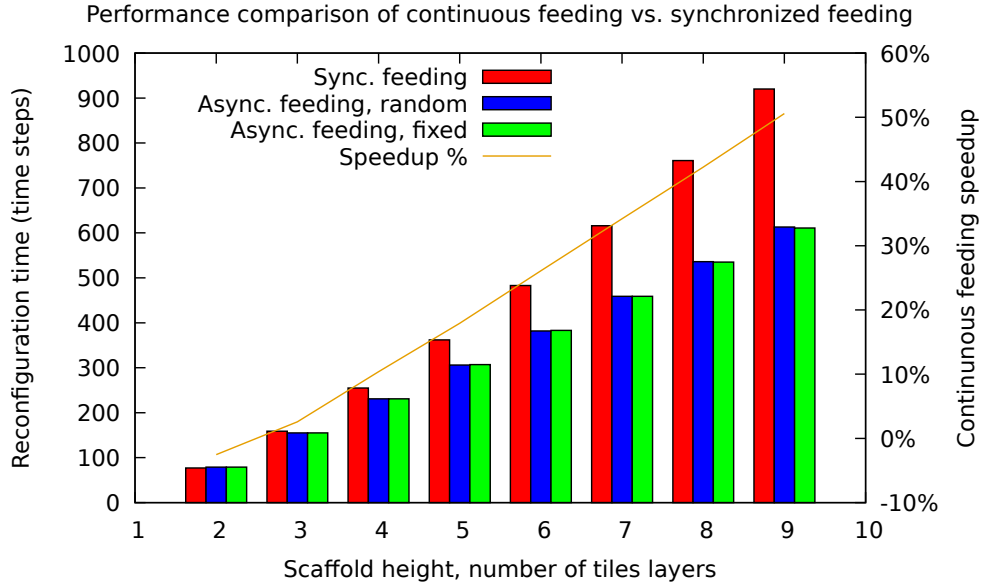


Figure 4.11: *Request-based* (*sync.* in the legend) *feeding* vs. *continuous* (*asynchronous* in the legend) *feeding* variants comparison, and variable motion duration results.

reconfiguration speedup and modules usage as performance indicators compared to the *request-based feeding* 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 4.11, 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: *request-based feeding*, and two variants of *continuous flow*, one with fixed movement time, one with pseudo-random varying movement time. We make several observations from the results: first, *continuous flow* performs faster than *request-based 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 footnote³). Second, both variants of *continuous flow* perform almost identically, which shows that our motion coordination algorithm allows modules to synchronize with their predecessors in a very efficient manner. Indeed, if one module performs a faster motion, it will have to wait until it can continue its movement for its predecessor to free the path. On the other hand, if its motion is slower, it will be able

³Side-by-side comparison video of *request-based feeding* and async: <https://youtu.be/XpG20m7waJk>

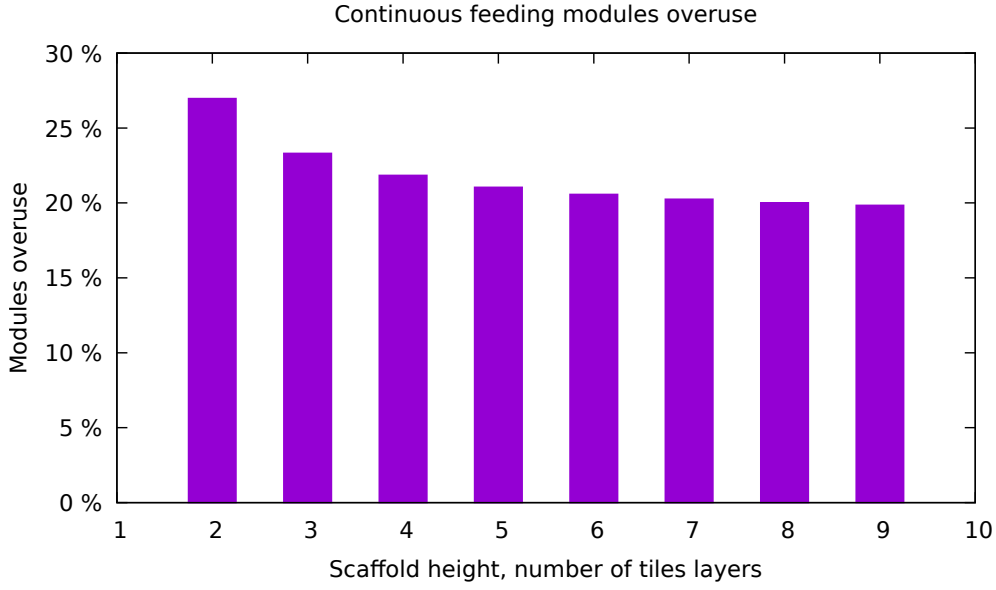


Figure 4.12: Modules overuse by the *continuous flow* variant.

to move to the next position without having to wait for its predecessor to leave (as it will have already freed the next position along the path).

Figure 4.12 shows the percentage of module overuse due to a continuous feeding, this can be compared to *request-based feeding*, which has no overuse. These modules are not lost since they can be sent back to the sandbox, or used for further operations. We have also seen that this excess can be avoided altogether by computing requirements directly at the sandbox and *turning off the tap* once enough modules have entered from an entry point in the *no-surplus* variant.

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 5. *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:

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

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

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

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

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

As the construction time gain increases and as the modules overuse remains stable as the size of the construction increases, we conclude that our algorithms with a continuous feeding of modules from the sandbox (regardless of surplus), *continuous flow* and its *no-surplus* variant, scale better than *request-based feeding*. It is a key property when dealing with programmable matter, since we aim at building shapes based on micro-robots which will require an enormous number of robots.

Remarks on Stochastic Motion Duration Results Figure 4.11 and the previous section showed that variance among modules in the duration of their motion had little to no effect on the reconfiguration time. While this is true in most cases as the different speeds among modules tend to cancel each other out, we find that this particular study suffers from a lack of accuracy and exhaustiveness. Indeed, given the role-based nature of our algorithm it can easily be seen that certain modules, the (future) coordinators, would have a much more serious time impact on the reconfiguration if their motion was delayed. This is in principle only serious if these modules are modules along the critical path of the object, in which case a late arrival of a tile root would halt the rest of the construction until that module would arrive, leading to an overall delay that is not recoverable. We feel that this previous experiment failed to represent that fact, and that more thorough evaluations with more data on slowed down critical path modules would be needed to reflect the true impact on motion delays on self-reconfiguration. This study nonetheless succeeds in validating our motion coordination algorithm as an effective motion synchronization method for avoiding reconfiguration issues in an uncertain and unpredictable reconfiguration setting.

4.3/ 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.

Semi-Convex Shape Definition

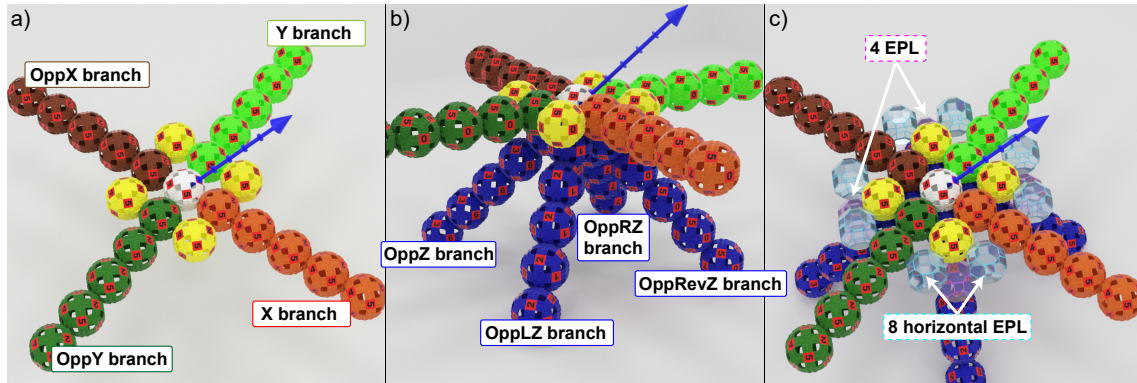


Figure 4.13: **Extended anatomy of a scaffold tile:** (a) Opposing outgoing horizontal branches **OppX** and **OppY**; (b) Opposing outgoing vertical branches, downward; (c) Addition of 8 horizontal entry point locations (in transparent blue) for horizontal feeding, along with the 4 standard vertical EPLs (in transparent pink).

4.3.1/ 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 $l_y(i)$ 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 \vec{x} or \vec{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 subparts 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.

4.3.2/ UPDATED MODEL AND ASSUMPTIONS

Specifying Shapes In order to perform self-reconfiguration with a full knowledge of the goal shape, modules only need to know 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) corresponds to 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 a 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.

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 ($-\vec{x}$) or to their front ($-\vec{y}$), 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 \vec{x} axis (thus, following axis $(-1, 0, 0)$), and **OppY**, in reverse along the \vec{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 4.13a).

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

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.

4.3.3/ 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.

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 4.14, $l = (4 - 1) \times 6 + 3 = 21$.

Theorem 6. *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 this 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 ($N_{\text{even}}^{\text{top}}$ if h is even or $N_{\text{odd}}^{\text{top}}$ otherwise); and *support* modules from the border of the bottom tile layer (N_0). Then we get:

$$N = N_0 + \left\lfloor \frac{h-1}{2} \right\rfloor N_{\text{odd}} + \left\lfloor \frac{h}{2} \right\rfloor N_{\text{even}} + (h \bmod 2) N_{\text{odd}}^{\text{top}} + ((h+1) \bmod 2) N_{\text{even}}^{\text{top}} \quad (4.9)$$

For example, in Figure 4.14: $h = 4$, $e = 3$ and $b = 6$ then

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

We express the several components used in Equation 4.9 depending on h , b and e :

$$\begin{aligned} N_0 &= 8h - 4 \\ N_{\text{even}} &= (6b - 1)h^2 + (e - 10b + 9)h + 4b - 4 \\ N_{\text{odd}} &= (6b - 1)h^2 + 2(5e - 5b - 4)h + 4b - 6e + 6 \\ N_{\text{even}}^{\text{top}} &= N_{\text{odd}} + 4(e - b)(h - 1)^2 - 2(e - 1)(4h - 3) \\ N_{\text{odd}}^{\text{top}} &= N_{\text{even}} + 4(e - b)(h - 1)^2 \end{aligned} \quad (4.10)$$

The length (in number of modules) of the critical construction path l_c of the cube (drawn in red in Figure 4.14) is obtained by a depth first traversal of one of the critical sub-trees, yielding the expression:

$$l_c = \begin{cases} 4(h-1)b + \frac{b}{2} + e & \text{if } h \text{ is even} \\ 4(h-1)b + e & \text{if } h \text{ is odd} \end{cases} \quad (4.11)$$

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 4.2.4, we obtain $O(h)$ as the height of the tree and thus an $O(N^{\frac{1}{3}})$ reconfiguration time for cubic shapes. \square

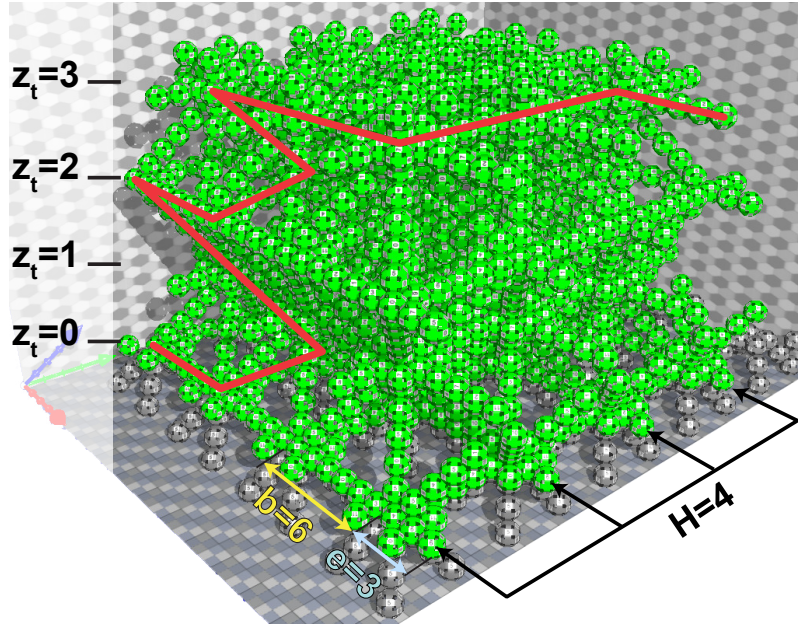


Figure 4.14: Example of a cube of length $l = (4 - 1) \times 6 + 3 = 21$. The critical path l_c of length 78 modules is drawn in red.

Generalization to *Semi-convex* Shapes

Theorem 7. *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:

$$l_x = (h_x - 1) \times b + e_x$$

$$l_y = (h_y - 1) \times b + e_y$$

$$l_z = (h_z - 1) \times b + e_z$$

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 $O(N^{\frac{1}{3}})$. \square

4.3.4/ SIMULATIONS

In this section, we provide various indicators to evaluate the performance of our algorithms, obtained from simulations on the *VisibleSim* simulator (see Section 1.3.2). First, we study a set of canonical shapes. Second, we provide a study on a larger and composite use case.

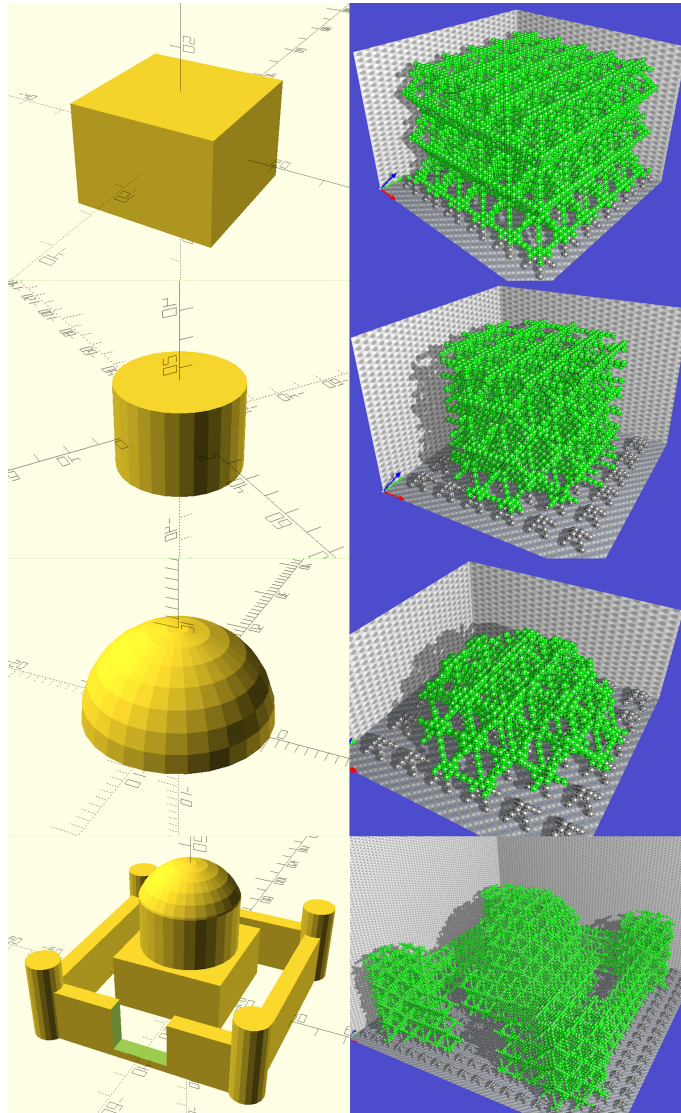


Figure 4.15: **Overview of the shapes under study:** (Left) OpenSCAD preview of the CSG model of the goal shape; (Right) Scaffold interpretation built by our algorithm. For canonical shapes, dimensions are set to $d = 6$ tiles.

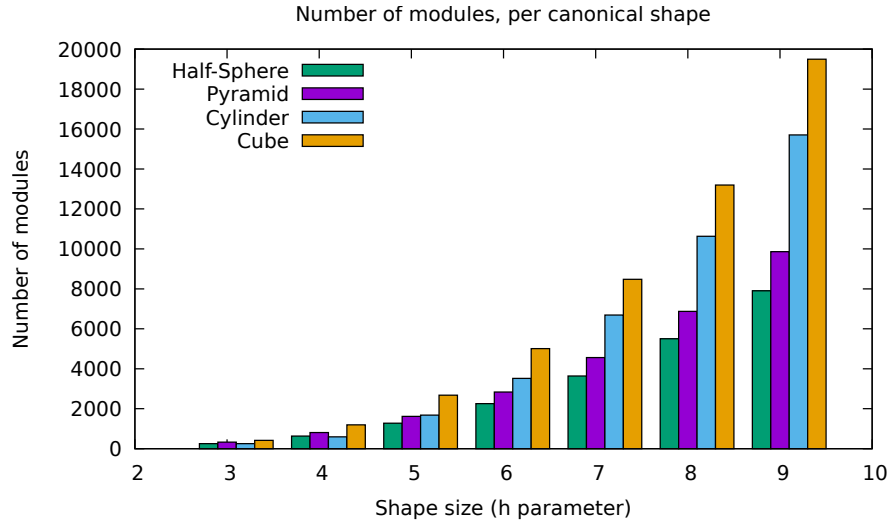


Figure 4.16: Number of modules in canonical shapes, with varying sizes.

Comparison between canonical shapes In the following pages, we compare the reconfiguration times with global and relative indicators⁴. We compare canonical shapes—i.e., pyramid, cube, cylinder, and half-sphere—with sizes ranging from $d = 3$ to $d = 9$ tiles wide. Figure 4.16 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 module 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:

- `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 4.17 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.

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

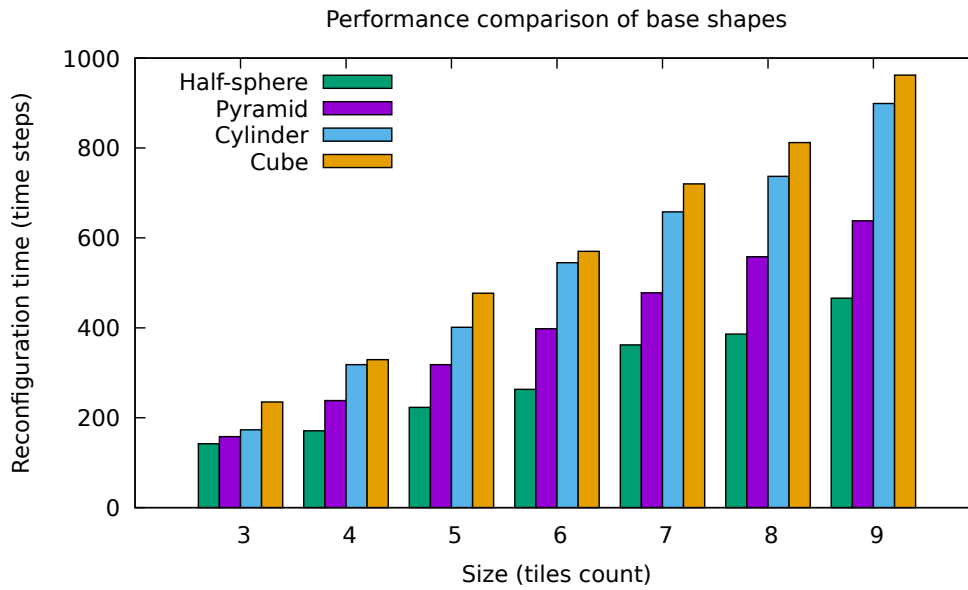


Figure 4.17: Global reconfiguration time, with varying sizes.

- The time increase is linear with the d parameter.

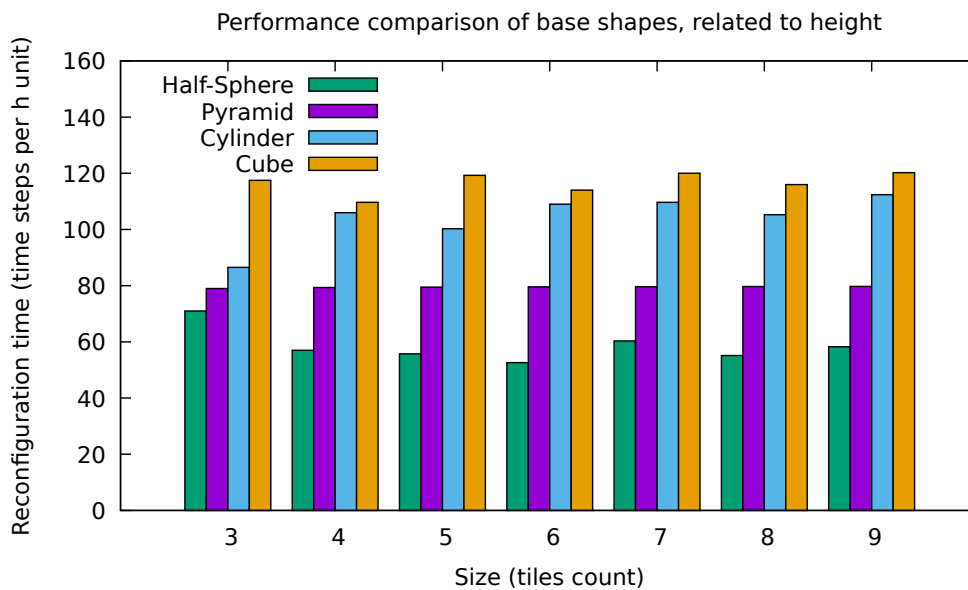


Figure 4.18: Reconfiguration speed in time steps per height level.

Figure 4.18 shows more clearly that the reconfiguration time is stable according to the size of the target shape.

Figure 4.19 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

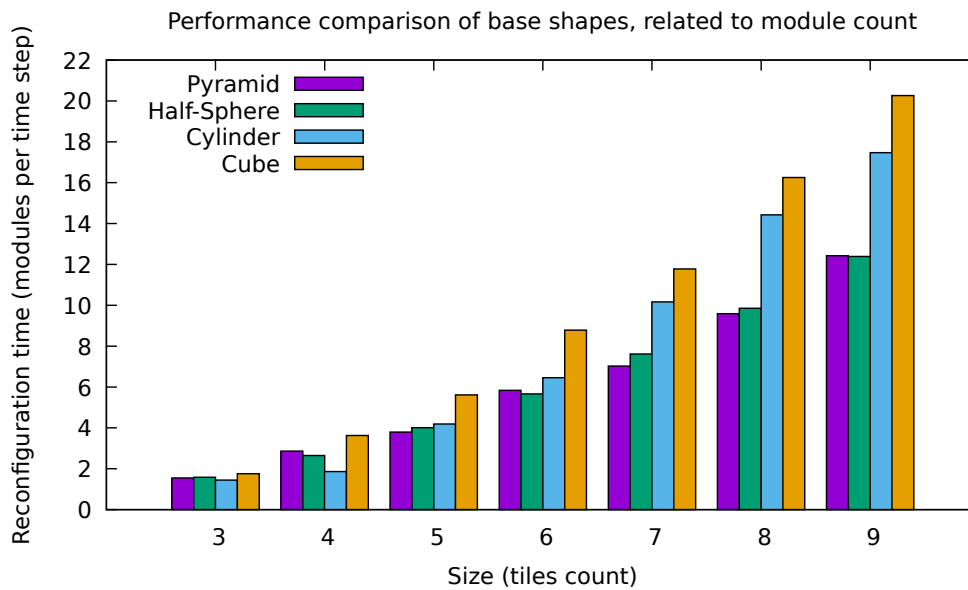


Figure 4.19: Reconfiguration speed in modules per time step.

also increases. It is easily explained by the ability of the algorithm to move more modules in parallel with a wider base.

- 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 4.20 shows the convergence rate of our 4 canonical 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 allows us to see the differences in terms of modules required to build a given shape as well as the corresponding 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.

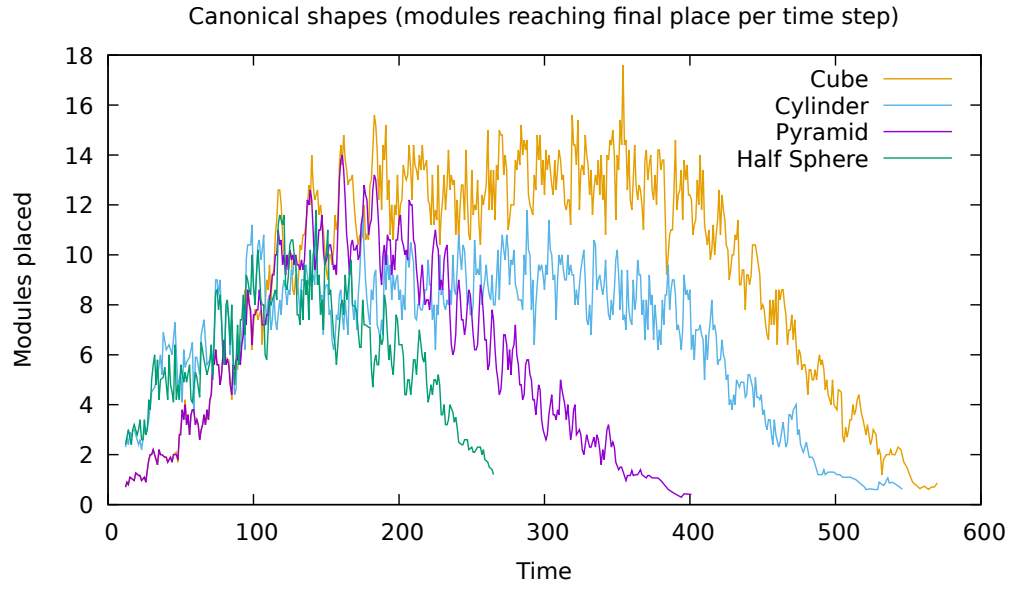


Figure 4.20: Instant module placement for canonical shapes with $d = 6$.

- We also observe a 3 steps 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.

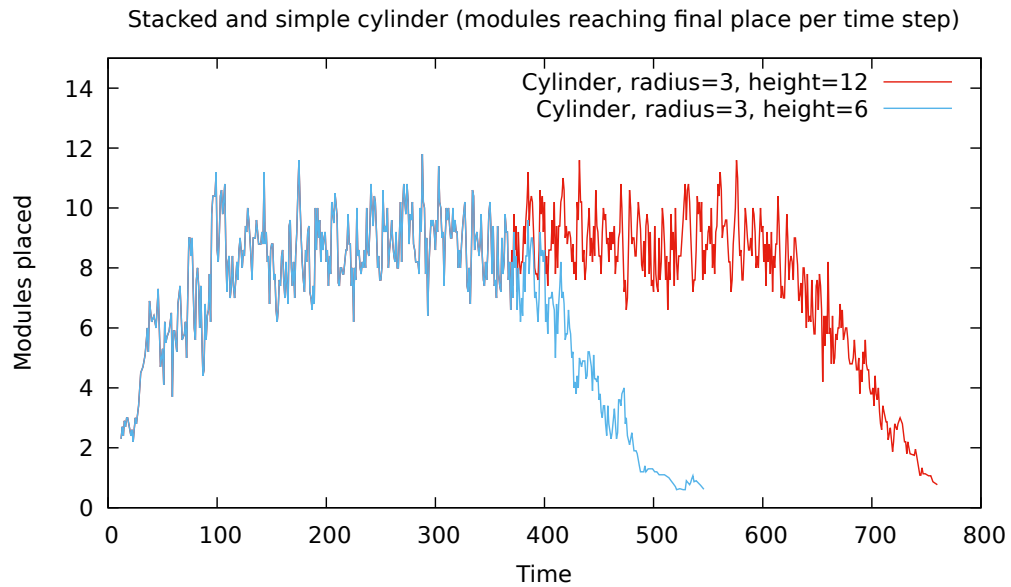


Figure 4.21: Instant modules placement: comparing simple and stacked cylinder. Both curves overlap until $t = 380$ time steps.

Figure 4.21 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 short one.

In the previous experiments, we have studied various metrics to quantify the performance of our reconfiguration method on canonical shapes. We have 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 4.3.3. We also show that some shapes (i.e., cubes and cylinders) are inherently more parallel than the 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 provides the optimal throughput to all the tiles of the shape, since module flows are never divided from the sandbox to their destination.

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. The CSG description of this shape is shown in Listing 4.1 below:

Listing 4.1: CSG description of the sandcastle composite shape.

```
union() {
  difference() {
    translate([0,-40,7]) cube([80,6,20]);
    translate([0,-40,15]) cube([20,8,30]);
  } // Front wall with door

  // Other walls
  translate([0,40,7]) cube([80,6,20]);
  translate([-40,0,7]) cube([6,80,20]);
  translate([40,0,7]) cube([6,80,20]);

  // Corner towers
  translate([-37,-37,12]) cylinder(30, 12);
  translate([-37,37,12]) cylinder(30, 12);
  translate([37,-37,12]) cylinder(30, 12);
  translate([37,37,12]) cylinder(30, 12);

  // Ground Base
  cube([80,80,6]);

  // Central Castle
  translate([0,0,12.5]) cube([40,40,20]);
```

```

    translate([0,0,32]) cylinder(20, 20);
    difference() {
        translate([0,0,35]) sphere(21);
        translate([0,0,33]) cylinder(18, 38);
    }
}

```

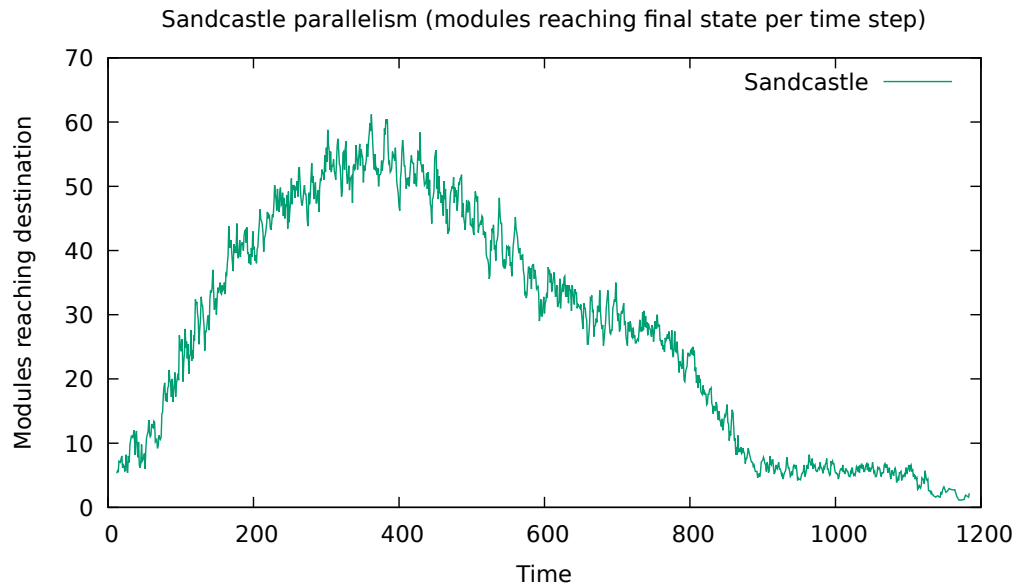


Figure 4.22: Sandcastle reconfiguration speed, modules in place per time step.

Figure 4.22 shows the reconfiguration speed as the instantaneous number of modules placed at each time step. We observe the same global behavior as 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 (5000 modules), the overall time is limited: only 1200 time steps for the sandcastle against nearly 600 time steps for the cube. That’s a $3\times$ speedup per module in 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. These shapes are independent and do not need to synchronize.

4.4/ 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 4.3.4—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.

4.4.1/ 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 4.3 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.

4.4.2/ 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 4.13c.

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 number 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 (Holobut et al., 2017).

4.4.3/ 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.

4.5/ DISCUSSION

Through the various formal analyses from Sections 4.2.4 and 4.3.3 and simulation results presented in Sections 4.2.5 and 4.3.4, 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 to. Furthermore, we have hinted at what could be done to transcend this limitation in Section 4.4.

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 subparts 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 Chapter 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 et al., 2007; Lengiewicz et al., 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 sublinear 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.

A SIMPLE COATING ASSEMBLY ALGORITHM

Contents

5.1 Coating Self-Assembly	141
5.1.1 High-Level Assembly Strategy (Bottom-Up Layering)	141
5.1.2 Standard Layer Assembly Strategy (The Tucci Algorithm)	142
5.1.3 Support Layer Assembly Strategy (Border Completion)	143
5.2 Results	146
5.2.1 Preservation of message complexity	146
5.2.2 Simulations	147
5.2.3 Complexity	149
5.3 Discussion	150
5.3.1 Limits of the Current Method	150
5.3.2 Discarded Coating Strategies	152
5.3.3 Module Dispatch From the Sandbox	153
5.3.4 Towards More Efficient Coating Methods	153

We have shown in Chapters 3 and 4 that unprecedented self-reconfiguration speeds could be achieved thanks to two propositions we made. First, engineering the reconfiguration environment such that the reconfiguration takes place over a reserve of modules, or *sandbox*, through which modules can be supplied from the ground of the reconfiguration scene or discarded from it. Second, by engineering the goal shape itself and building a porous skeleton, or *scaffold*, of the shape instead of the compact target object, and with a regular and predictable internal structure, the construction requires fewer modules and it is much easier to coordinate the flow of modules for maximum parallelism and efficiency.

Nonetheless, the scaffolding technique has one major drawback as explained in Chapter 3, which is that the external aspect of the built object is not preserved (as its surface is porous too), so the fidelity of the constructed object to the supplied model is lessened. For that reason, we proposed to cover the surface of the porous object using a single layer of modules, through a process called *coating*, so as to recreate the correct external aspect of the object and complement scaffolding. This scaffold and coating method can be seen as a special case of shape self-reconfiguration from a reserve of modules.

This chapter provides a straightforward algorithmic solution to the coating problem introduced in Section 3.3, showing that even with a relatively inefficient coating method, using a coated scaffold may be preferable to building a dense shape.

This work relates most closely to the literature on robotic self-assembly, whose aim is to produce a correct and deadlock-free assembly plan for constructing a shape, either made from passive materials brought by swarm robotic units (Werfel et al., 2014; Deng et al., 2019), or from modular robotic units themselves as in our case (Tucci et al., 2018; Pescher et al., 2020; White et al., 2005).

In the second case, self-assembly approaches usually rely on a set of construction rules that provide a feasible and deadlock-free assembly plan to construct a shape. This assembly plan is sometimes pre-computed prior to the construction itself, in a centralized manner. It can then be followed by the robotic units taking part in the construction of the target shape. This is usually done through a virtual disassembly of the target shape, as is the case with the compiler for the TERMES swarm systems (Werfel et al., 2014; Deng et al., 2019), which has been recently applied generically to modular robotic self-assembly in (Pescher et al., 2020). In other instances, such as in (White et al., 2005), the exact order of the construction of the shape is pre-defined by a human operator.

Lastly, assembly decisions can be made by the distributed robotic units on the fly, by relying on a set of local neighborhood rules that describe the order and constraints under which a module of the shape must attract neighbors and by resolving global constraints through communication (Tucci et al., 2018) (Thalamy et al., 2019c,a). Our present work belongs to the latter category.

Finally, the problem of coating a 2D shape has been studied theoretically in the context of *self-organizing particle systems (SOPS)*, more specifically under the *Amoebot* model. In this theoretical approach, it is shown in (Derakhshandeh et al., 2017; Daymude et al., 2018) that the coating of a 2D object can be done using only local information in linear time with high probability. To the best of our knowledge, our work is the first work on the coating of a modular robotic structure by other modular robotic units in 3D.

5.1/ COATING SELF-ASSEMBLY

This section describes how the coating is assembled in such a way that no deadlock can occur, without regard to the actual flow of the modules from the sandbox to their target location. Our assembly method can be decomposed into three different components:

- A high-level rule set that describes how the coating must be assembled, essentially building the horizontal layers constituting it one at a time, from bottom to top.
- Two different strategies for assembling a coating layer, which determine the assembly order of the coating within that layer:
 - ◊ a 2D assembly algorithm that can assemble a wide and complex border but that does not support the presence of obstacles.
 - ◊ a border completion algorithm, that is less parallel but can assemble borders containing obstacles, such as the *structural supports* mentioned in Section 3.3.

The strategy to be used for a given layer thus depends on whether or not structural supports, potentially causing obstacles, have been introduced for that layer. We will use the term *attract*, to refer to the action of a module that advertises that a position next to itself is ready to be filled, causing another module to come and claim it, thus filling that position.

5.1.1/ HIGH-LEVEL ASSEMBLY STRATEGY (BOTTOM-UP LAYERING)

The manner in which the coating is assembled at the scale of the object can be summed up as **Bottom-Up Layering**, which means that the coating is assembled one layer at a time, from the base of the object to its top. This may be sub-optimal, but thanks to the space between the scaffold and the coating itself, this relaxes the constraints imposed on the construction of a given layer, turning it into a simpler 2D problem. Coating layer n thus has to wait for coating layer $n - 1$ to have finished building before starting its construction, with $n > 0$.

The assembly of a given layer always starts with the attraction of a module to a single position of that layer, and proceeds through the recursive attraction of neighbors by attracted modules, according to the rules detailed in the next subsections. The need for a single source and direction of growth of the shape is a consequence of the motion constraints of *3D Catoms* introduced in Subsection 1.2.1. Let $seed_n$ the first module that must be attracted for horizontal layer n of the coating. Please note, however, that for layer made of multiple disjoint parts as it might happen in some shapes, these parts are built independently, from different seeds. This module will be attracted by a module from the previous layer, $attractor_n$, determined according to a method inspired by the Tucci Algorithm (Tucci et al., 2018):

Any module can test if it is an attractor module for the next plane by checking if it has a top neighbor position that is part of the border of the coating, and if that is the position with minimum y position and maximum x position across the border that has a bottom neighbor that is in layer $n - 1$. This can all be done through a virtual border following and an exploration of the next coating layer, as modules all hold the CSG description of the shape in memory. $seed_0$ is simply determined by the coordinate criterion as it has no matching attractor. A simple messaging is used to reach a consensus on when the construction of the current layer is over, and for notifying the next attractor (or attractors in the case of a splitting of the shape) that the construction of the next layer start.

From there on, two different methods are used to assemble a given layer, depending on whether this layer needs to attract support modules or not. Now, these methods need to lead to a correct solution systematically, no matter what the morphology of the coating layer is like. Indeed, while the simplest coating layers are simply a one-module thick border around a section of the object, when the surface of the object has a steep slope, and which can be solved trivially by simply adding modules one at a time following the border, more complex cases exist. These cases include cases with thicker borders when the surface of the object has a gentler slope (as the lattice forces an approximation of the shape, much like the approximation of lines using Bresenham's algorithm in computer graphics (Bresenham, 1965)), when the layer is a fully horizontal plateau, or any combination of those cases. Systematically finding an assembly plan is hence not trivial in all cases.

5.1.2/ STANDARD LAYER ASSEMBLY STRATEGY (THE TUCCI ALGORITHM)

In the case where the layer does not need to attract support modules (when $z \bmod b \neq 0$, with b the branch length parameter of the scaffold), the assembly problem is a standard 2D assembly problem, without obstacles. Luckily, the assembly problem without obstacle has been previously solved in the exact context of FCC lattice modular robots using the

Tucci Algorithm (Tucci et al., 2018), hence we can rely on this method for this type of layers. The 2D version of this algorithm is briefly explained in the rest of this section.

As stated in the previous subsection, every layer starts with the attraction of a first module to the *seed* position. Then, each module M_i attracts neighbor modules to the cells among its 4 horizontal neighbor locations that are inside the target shape G (or coating in our case), according to local rules that enforce a diagonal growth direction of the plane and without deadlocks. The rules are built in such a way that :

- Modules having a local neighborhood in which the addition of a neighbor in a direction does not risk to cause another position to become unreachable, attract a module to that neighbor position right away.
- Otherwise, if an attraction might block a nearby position, they communicate with their neighbors to synchronize and ensure that the attraction is only performed when potential blocked positions are filled.
- If a target position might be a merge point between two parts of a plane growing concurrently around an internal hole, the neighbor seeking to attract a module to that position will send a probe message that will follow the border of the hole and will return when all the other positions of the border have been filled.

Please refer to (Tucci et al., 2018) for more information on assembly rules and experiments showing that this method achieves a very high convergence rate into the goal shape, with only a number of messages linear in the number of modules in the shape.

5.1.3/ SUPPORT LAYER ASSEMBLY STRATEGY (BORDER COMPLETION)

For all layers that need to attract support modules ($z \bmod b = 0$), first, all the structural supports are attracted by their neighbor modules from the previous layer, and then a different assembly algorithm is applied, taking into account the supports as obstacles.

Structural Supports and Segments Attraction Therefore, when the consensus on the completion of the $n - 1$ layer has been reached, all the modules check whether they have a support position as a top neighbor on the next plane, and attract another module to that position if that's the case. However, as mentioned, the structural support modules now create obstacles to the assembly process, which because of the *bridging constraint* of 3D Catoms can cause neighbor coating positions to become unreachable for any growth direction other than one originating from the support itself (see Figure 5.1.b). This is not always the case, and module can cause no potentially blocked cells (see Figure 5.1.a), as this depends on the location of the support module itself.

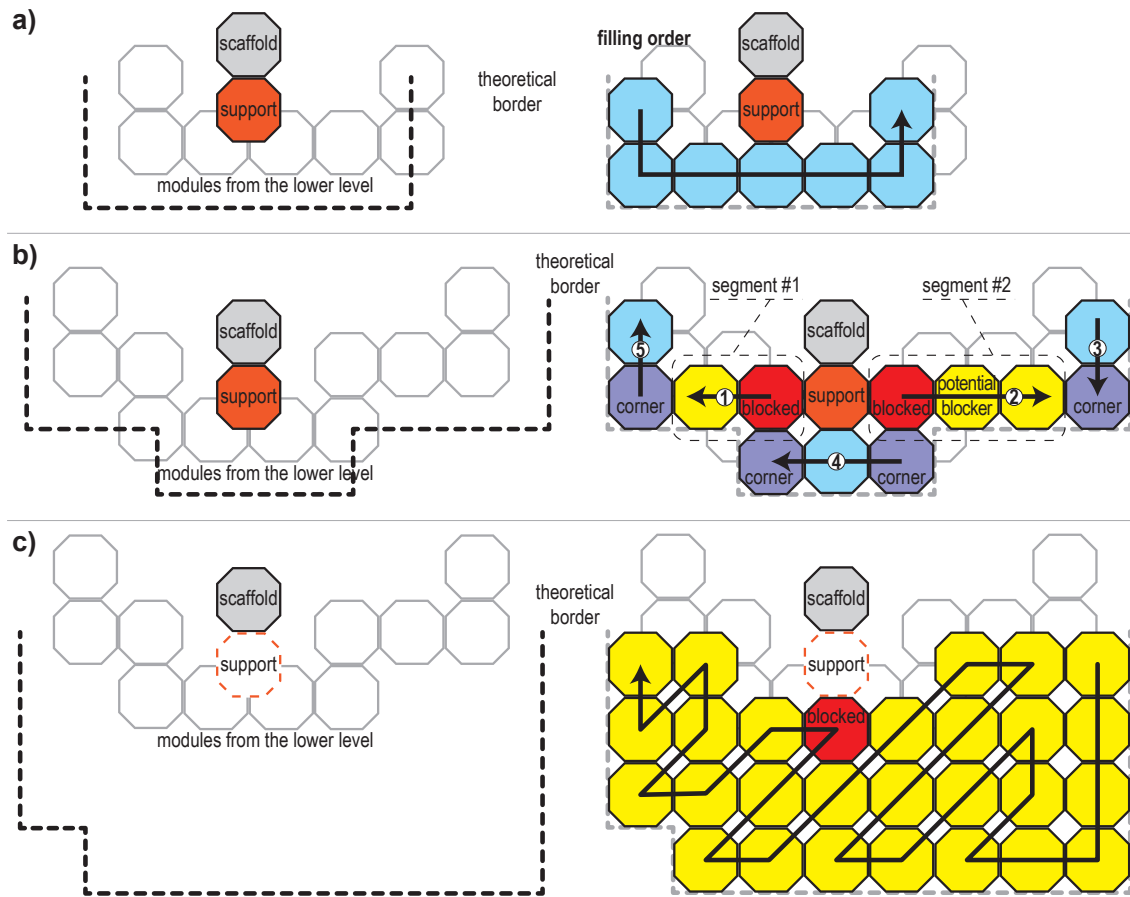


Figure 5.1: **(a)** Structural support producing no blocked positions; **(b)** support producing blocked positions and corresponding support segments; **(c)** support position that cannot be filled in the current implementation.

Now, let us focus on the case where the support is attracted and might cause some positions to become blocked (Please refer to Figure 5.1.b throughout this paragraph). In that case, the support module (in Orange) would cause its East and West neighbor positions to become unreachable, blocked by the yellow modules, if the direction of growth of the border has any source other than the support. This is not the case of its South neighbor (blue, bottom), as the position opposite from the support is not part of the coating. In that case, the only way to avoid a deadlock is by actually growing this part of the coating from the support module. Fortunately, this does not require the whole coating layer to be grown from the support, which would cause intractable synchronization issues, but simply a number of segments, originating from the support, followed by the blocked position, and consisting of all modules in that direction until the next corner module (in purple, not part of segments). While it is possible that two support modules are adjacent to modules of the same segment, this segment will only be created by one of the two structural supports, as it is impossible that a coating border contains two structural supports that create blocked positions without a corner between the two, except on wide borders.

Indeed, in extreme cases, in which a support is adjacent to a coating section that is larger than one module thick, the insertion of the support would force the construction of the whole plane from the support. This can be problematic if one or more other structural supports are adjacent to the same coating section, causing multiple starting points of this coating layer that are very hard to synchronize. In such case, where a support is adjacent to a border thicker than one module, its insertion is omitted (see Figure 5.1.c).

Segments are grown by recursively attracting a module to the next position of the segment and sending it a message when it arrives instructing it to continue the growth of the segment, until the next position is a corner. When that happens, an acknowledgment is returned to the support growing that segment, so that it can be known when it has finished growing its segments.

Segment Detection Once all segments from all structural supports have been grown, the coating layer will be partially filled by all segments positions. As Tucci's algorithm does not support obstacles (otherwise simply defining support positions as part of the coating would have sufficed as a general solution to our coating problem), another method of assembling the remaining modules is required.

This algorithm can only start once all the segments are in place, so the first step is to detect when that is the case. For that purpose, the *attractor* module of the previous layer will send a *NextPlaneSupportReadyRequest* message across the external border of the coating, in a single direction. This message contains a single bit of data, which indicates whether a segment has been detected along the border above. For all modules that do not have a support neighbor in the layer above, the message will be directly forwarded to the next module along the border. For all the others, one of three things can happen:

1. Their support neighbor has not yet arrived: In that case, the module will wait for the support to arrive, and then forward the request to it.
2. The support is present and has not finished growing its segments: The module forwards the request to it and the support holds its response until its segments are complete.
3. The support is present and has finished growing its segments or does not have any: The module forwards the request to it and the support responds with *True* if it has grown a segment, *False* otherwise.

Once a module receives a response from a support, it sets the segment detection bit to $sd = sd \text{ or } sd_{\text{rcvd}}$ so that once set to *True* it cannot be unset later along the path. The message is then forwarded with the newly set data bit to the next module along the path. This ends when the message reaches the *attractor* module again, from the other side of

the border. Since its propagation cannot proceed as long as all the segments of the structural supports along the path are not complete, the purpose of this message propagation is twofold: detecting the presence of segments in the layer above; (2) detecting when all structural supports have been attracted and have finished constructing their segments.

Border Completion Algorithm At that point, either no segment has been detected and the *attractor* initiates the Tucci algorithm as before, since there are no obstacles, or, segments have been detected and it performs a border completion algorithm. It is quite straightforward: if the *seed* module of the next layer is not part of a segment, it first attracts it. Then, or if the *seed* is part of a segment, it sends it a *BorderCompletion* message. The border completion message is then propagated along the coating border in a single direction (or two, but this requires a synchronization corner somewhere along). When a module receives it, either the next coating position along the border is already filled, and it forwards the *BorderCompletion* message there, or first attracts a module to that position and then forwards it. The layer is over when the message returns to the *seed*.

When all layers of the coating have finished building, the coating algorithm terminates.

5.2/ RESULTS

5.2.1/ PRESERVATION OF MESSAGE COMPLEXITY

In this section, we show that the number of messages used by our coating assembly method is linear in the number of modules in the coating. It was shown in (Tucci et al., 2018) that the number of messages to assemble a shape using the Tucci algorithm was linear in the number of modules in the shape. We thus aim to demonstrate that this result is preserved and that our method is asymptotically as efficient.

First, as explained in Section 5.1, all coating layers that are not at the level of scaffold tile roots (every b layers, with b the scaffold branch length parameter) and thus do not have structural supports, are simply executing the Tucci algorithm to assemble the coating, and thus do not require any additional message exchange. It is therefore sufficient to show that the assembly of a coating layer that has structural supports is also in $O(N_i)$, where N_i is the number of modules at layer i of the coating, and such that $\text{mod}(i, b) == 0$.

To that purpose, the list of all types of messages (and not part of Tucci's algorithm) used by our method is reviewed below:

- **Support *segment* attraction and completion messages:** A support can have a maximum of two segments growing from it according to our support selection

criteria, and each segment will have a length that is a fraction of N_i . The growth of a segment of length l takes l messages for its construction and l messages to notify the parent support of completion. Thus it is in $O(N_i)$.

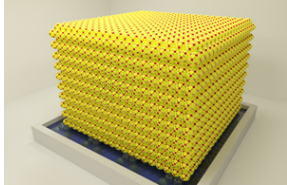
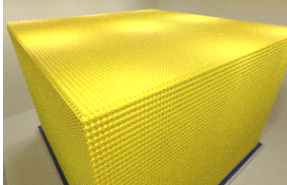
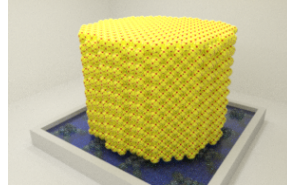
- **Support ready detection messages:** *NextPlaneSupportReadyRequest* messages are propagated along the border of the coating, reaching at most once every module of the coating that is not a neighbor to a structural support on the same plane. For all modules that have a support neighbor, it takes an additional message to request the state of the support, and another one for its response. This process thus also takes $O(N_i)$ messages.
- **Border completion messages:** This is propagated once from the attractor of the previous plane to the seed of the current one, and then once per module of the coating, it is therefore also linear in the number of modules in the coating layer.

Since all of these messages are used in a number linear to the N_i , we can therefore deduce that the total additional number of messages used by our method compared to Tucci's algorithm is also linear in the number of modules in the shape, and thus the message complexity of the overall coating algorithm is also in $O(N)$, with N the number of modules in the coating.

5.2.2/ SIMULATIONS

The following simulations have been performed on VisibleSim (Piranda, 2016), a lattice modular robot simulator. We have constructed the coating of objects of varying complexity to show that our method works on very diverse styles of shapes. A video of these simulations is provided¹ to better illustrate the method and results, which also contains additional explanations of the coating algorithm. The *Cube 100* is excluded from the video, as it is just like the *Cube 20*, but bigger. The results are presented on Table 5.1 and additional details regarding the specificities of each shape are provided below:

- *Cube* ($l = 20$ modules): Uses Tucci's algorithm only as all structural supports are non-blocking.
- *Cylinder* ($h = 20, d = 20$): Simple border completions.
- *Chair*: Concavities. Merge of disjoint components.
- *Sandcastle*: Large complex shape. Splitting components.
- *Cube* ($l = 100$ modules): Uses Tucci's algorithm only. Very high volume.

	Cube (l=20 modules)	Cube (l=100)	Cylinder (h=20, d=20)
			
Coating Count	1769	49404	1820
Coating Time	773	20336	1476
Coating Rate	2.29	2.43	1.23
Density Ratio	31.53%	19.87%	37.35%
Coating Ratio	69%	24.73%	76%

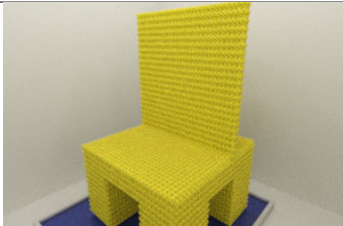
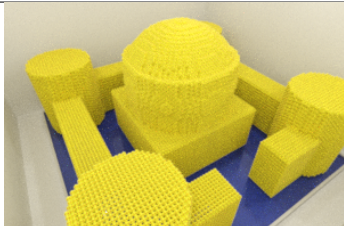
	Chair	Sandcastle
		
Coating Count	39225	8430
Coating Time	23652	4155
Coating Rate	1.66	2.03
Density Ratio	29.12%	38.89%
Coating Ratio	60%	84%

Table 5.1: Simulation results of our coating algorithm on shapes of various sizes.

The number of modules in the coating and the coating formation time are represented by the *coating count* and *coating time* values on Table 5.1, respectively. The coating time is expressed as a number of time steps, with a single time step corresponding to the insertion time for one module. We can thus deduce from the coating time and coating count the coating rate, the average number of modules attracted per time step. As we will see, the coating rate depends on the number of disjoint subparts of an object, and the portion of the coating that is on horizontal planes, which can be easily built in parallel.

Then, the *density ratio* expresses the ratio of modules in the coated scaffold object over the number of modules in the dense version of that object. It appears that the higher the volume of an object, the lower its density ratio, as the gain in modules is essentially the result of the scaffold.

Finally, the *coating ratio* expresses the ratio of coating modules over all modules in the final shape. For shapes big enough, the number of coating modules will become lower than the number of scaffold modules. Though not only the size matters as ultimately the coating ratio just reflects the size of the surface of an object relative to its volume. In our

¹<https://youtu.be/5nQVQgAu3SQ>

case, the actual volume of the shape in number of modules grows slower than for dense shapes because of the porous nature of the scaffold. The *Chair* for example, despite its relatively big size, has a very low volume compared to its surface, and this is reflected in its coating ratio. As our coating algorithm is inherently slower and less parallel than the scaffold construction algorithm, our scaffold coating method will therefore become even more efficient compared to the construction of dense objects for shapes with a large volume (e.g., *Cube of side 100*), where most of the modules belong to the scaffold.

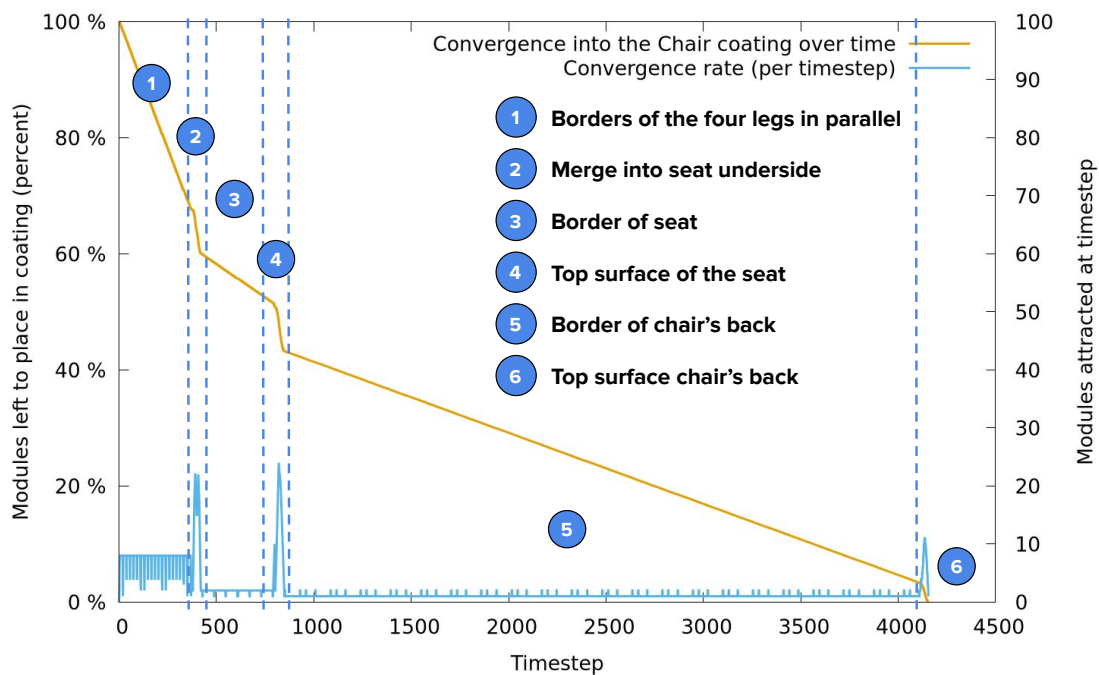


Figure 5.2: Convergence of the coating algorithm into the coating of the *Chair* shape over time.

While it is always linear, the convergence rate of our method is very uneven depending on the portion of the coating being built, as can be seen in Figure 5.2 with the example of the *chair*. Whenever the current coating layer is a border, the convergence rate is rather low (1 or 2 modules per time step and disjoint subpart), whereas planar layers cause great accelerations in the convergence rate, as all modules along the diagonal of the plane can be attracted at the same time. We can therefore conclude that the speed of the construction of the coating of a shape depends on the percentage of the coating that is part of a horizontal plane, as well as the number of disjoint subparts of that shape.

5.2.3/ COMPLEXITY

We have seen in the previous section that the time to assemble the coating of an object — regardless of the motion of the modules from the sandbox to their attraction position — was linear in the number of modules composing the coating. We now want to show

that even when considering the motion of the modules, the construction time is still linear in the number of modules, as long as trains of modules moving in parallel are formed, moving from a sandbox source location to the available positions of the coating. Indeed, let us consider a single source of sandbox modules at the base of the scaffold, or rather a single source per disjoint component of the object, and where modules are introduced at regular intervals leaving only one free position between them, forming a train of modules. Let t_i the time at which the coating position number i in the assembly order is filled, and assuming that it would take a constant number of time steps c (realistically 2 to 4) for the next module along the train to take its position, then:

$$t_0 = 0, t_1 = t_0 + c, t_2 = t_1 + c = t_0 + 2 \times c, \dots$$

Therefore, $t_n = t_0 + n \times c$, the construction time is in $O(n)$.

Our self-reconfiguration algorithm for building the scaffold of a shape has been shown to have an $O(\sqrt[3]{N})$ reconfiguration time for all *semi-convex* shapes, e.g., having no vertical concavities, and with no hole between the sandbox and the shape. This makes the total reconfiguration time $O(n_{\text{coating}}) + O(\sqrt[3]{n_{\text{scaffold}}})$, or $O(n_{\text{coating+scaffold}})$ for that class of shape. But more importantly, our method reduces the number of modules to reconfigure to a fraction of the number of modules in the dense version of the object for the same visual result, massively cutting down on reconfiguration time. Furthermore, the bigger the object, the bigger the gain in saved modules. Nonetheless, for the scaffold and coating method to reach its true potential, the current performance of the coating algorithm will need to be greatly improved.

5.3/ DISCUSSION

In this section, we discuss the capabilities and limits of the current coating method, provide insights on how the motion planning of modules from the sandbox to the coating could be implemented, and propose several perspectives for generalizing the current method or replacing it with a more parallel and efficient algorithm.

5.3.1/ LIMITS OF THE CURRENT METHOD

We have shown that the current simplistic coating method is well suited to build the coating of scaffolds that belong to the *semi-convex* geometric class. Furthermore, while we have not stated it explicitly until now, the class of object geometries supported by our coating method is in fact larger than *semi-convex* shapes alone. Indeed, as shown in Section 5.2.2 our algorithm was able to build the coating of the scaffold of a chair (which

was generated programatically but not through the scaffold algorithm from the previous chapter), which does not belong to the *semi-convex* class as the seat of the chair creates a concavity with regard to the sandbox.

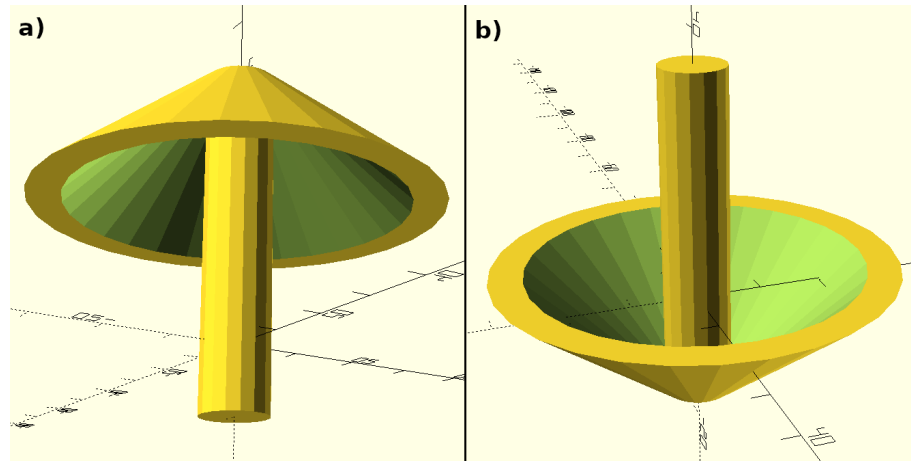


Figure 5.3: Two very problematic shapes given the current coating strategy: **(a)** Mushroom-like shape where part of the coating on layer n is not connected to any layer $n - 1$ coating section; **(b)** Spinning-top shape with a bowl or depression concavity, which is also a case of disconnected coating.

What sorts of scaffolds, then, can this method coat? This has to do with the general strategy for assembling the coating, which in this case we defined as *bottom-up layering*. The unfortunate condition for such a straightforward approach to do the job is that **any disjoint portion of a coating layer has to be directly adjacent (or connected) to the layer below** — all shapes matching this description should thus be supported. Otherwise this would be akin to the construction of an unconnected overhang in 3D printing. In our chair example, even though the four legs are disjoint coating parts, they all merge into the underside of the seat, which is connected to them, so our method operates with no hurdle.

Conversely, take the two examples of shapes shown in Figure 5.3, which are just a 180° rotation of one another along the \vec{y} axis. In the mushroom example, coating would start along the handle and our method would fail at the layer where the bottom of the mushroom cap starts, as modules would be unable to reach the bottom edge of the cap since it is not connected to the coating on of the handle from the previous layer. Then, in the spinning-top example, the challenge is to coat the interior of the bowl at the center of which the handle sits. The cone forming the tip of the spinning-top would have to be coated both on its inside and outside for each of its coating layers — this is not currently supported in our method.

There are several ways to address the problem, which will only be briefly mentioned below. The first possible method is to keep a *bottom-up* approach as a general rule, and find a path for modules to reach the disconnected coating parts. This could either

be done by finding a path through the scaffold, but planning could be complex and the path not very direct, or we could rely on an additional temporary scaffold, whose sole purpose is to provide a direct path to a portion of the coating that is disjoint from the rest. In the mushroom example, it could be a single *scaffold “tower”* located in the sandbox directly under part of the edge of the mushroom cap, and connecting it to the sandbox. In that way, modules could be sent there from the sandbox and used to coat the edge and top surface of the cap, in parallel with the rest of the handle and underside of the cap. Similarly in the spinning-top example, a scaffold could connect the edge of the bowl to the sandbox so that modules can climb up the sandbox and down the bowl to coat its inside. This is likely to have a negligible impact on reconfiguration time, as this temporary scaffold could be built during the construction of the target object’s scaffold itself, in parallel, and deconstructed once it needs not supply any more modules. Another approach to support more complex shapes consists in coating by following the surface of the object, which would yield the following order for the mushroom example: (1) handle; (2) cap underside; (3) cap edge; (4) cap top. This ensures that any coating portion is always connected to the (partial) layer that was built in the previous round.

5.3.2/ DISCARDED COATING STRATEGIES

A number of alternative coating strategies were investigated while researching this work, and we explain why they have not made the cut below.

We have not mentioned a strict top-down coating approach in the previous section, because it does not sound like a good strategy in the general case, since modules would have to climb through the scaffold and then escape it to reach the coating, but as the coating would be assembled, the coating itself could then prevent modules from escaping the scaffold. This is why we had to resort to using the coating itself as a path for dispatching modules. For this same reason, even with a bottom-up approach, dispatching modules through the scaffold is also an issue as the coating would block many of the scaffold branches on the edge of the shape, drastically reducing the benefit of the scaffold and even leaving no internal path with very narrow shapes.

We have also studied the construction of the coating in parallel with the construction of the scaffold, but this also has the effect of blocking too many of the scaffold paths, thus decreasing the parallelism of the method, and hindering the geometric separation of concern enabled by pipelining, thus making planning more complex.

5.3.3/ MODULE DISPATCH FROM THE SANDBOX

The algorithm introduced in this chapter only addressed the assembly strategy of the coating, that is to say the scheduling of the construction, without regard to the actual dispatch of modules from the sandbox. This section hence touches on how modules can be routed from the sandbox to open positions within the shape using a gradient system.

We have started studying a self-reconfiguration planning method based on attraction gradients emitted by modules adjacent to coating positions that is ready to be filled. This gradient is propagated through all the modules from the layer. Assuming there is a single source of modules from the sandbox to the coating, modules then move in a train-like fashion until reaching the coating and then simply follow the existing coating from the current layer being built. Every time they plan a motion, they probe their neighbors that are already inside the coating layer for the attraction gradient values (essentially a pair of positions and Manhattan distances), and pick the destination with the smallest distance. The gradients are invalidated when a module reaches an attraction position.

While this gradient is not very important on simple borders with a thickness of only one module, as on these borders modules could have just followed the borders until a free and ready spot is reached, it is a lot more important on more complex borders. For large borders or entire planes of coating, the assembly order of the coating is non-trivial and many positions can be ready to receive a module at the same time. This thus requires some way of prioritizing the construction and dispatching the modules to the closest available coating positions — this is where the attraction gradient and gradient descent come in.

The major challenge of this approach, however, is to deal with sudden changes in the gradients that could make modules abruptly change direction, which might trigger collisions and other coordination issues. This problem will become even more salient when considering multiple sources of modules. This motion aspect of the coating problem will require more research, which existing works on self-reconfiguration and distributed motion planning can certainly guide.

5.3.4/ TOWARDS MORE EFFICIENT COATING METHODS

The current version of the algorithm might be too slow at reconfiguration time for some minor changes into the configuration, as would be required to first disassemble the coating, modify the scaffold, and then reassemble the coating. Otherwise, minor adjustments could be made simply by adding coating modules, and only when surpassing a certain threshold could the scaffold be also altered, thus saving precious time.

While our current method of coating is quite straightforward and by itself leads to the

construction of a coating in linear time, it is obviously unsatisfactory that only wide or planar borders can achieve high parallelism. An ideal coating solution would build **any** layer with some degree of parallelism, truly leveraging the benefits of the sandbox and multiple sources of modules.

We are therefore investigating two advanced strategies to improve the parallelism of the method.

- 1. Parallel Single Layer:** For all non-planar layers, we can segment the coating border into sections of the coating separated by corners, which can be formed in parallel. Once two adjacent segments are complete, the corner modules in between can be added. On a cube, that would mean that horizontal segments from all four horizontal faces of the cube can be built in parallel, with a synchronization of the construction on its vertical edges. But shapes with many more edges could therefore be built even faster. Ultimately the coating speed using this method would be a function of the number of modules in the shape, its number of corners, and the number of module sources.
- 2. Parallel Multilayer:** Allow the attraction of a module to a coating position as soon as all its neighbor positions that are in the previous layer are filled (and, of course, its horizontal dependencies too). In that way, the construction of the coating can also proceed in a vertical diagonal manner, building multiple planes at once. However, special rules would have to be designed regarding the introduction of support modules so that they do not hamper the construction. In a sense, this would be equivalent to extending the multilayer version of the Tucci algorithm, that can produce a highly parallel construction plan for any shape, to support the presence of obstacles.

These strategies can be iterative, with the former probably easier to implement. With both strategies, numerous sources of modules are needed to dispatch them optimally, and the concurrent planning of the motion of the modules from their source to their destination becomes non-trivial if there are fewer sources than coating sections that can be built in parallel.

CONCLUSION

CONCLUSION

Contents

Summary of the PhD thesis	158
Discussion	161
Perspectives	167

SUMMARY OF THE PHD THESIS

The work presented in this thesis is part of a larger effort to create matter with programmable properties and enable humans to fully master their environment so it can be reshaped, adapted, repaired, or customized at will. The current best substrate for programmable matter is modular robotic systems — robotic systems composed of interconnected modules that must coordinate through motion and communication to achieve a task. This work focuses on using programmable matter technology to achieve tangible and interactive 3D display systems that could revolutionize the ways in which we interact with data and the virtual world. The main algorithmic challenge of such task, however, lies at the center of both programmable matter and modular robotic systems: reconfigurability. Very large modular robotic systems with up to hundreds of thousands of modules in a particular arrangement can thus be used to form tangible shapes, which can be transformed at will into another arrangement of the robot thanks to the motion of the robotic modules — a process called self-reconfiguration. The self-reconfiguration problem is hence to find a sequence of individual module motions that can transform an initial arrangement of modules into a goal one. Due to the kinematic, communication, control, and time constraints imposed on the modules during this process, self-reconfiguration is a very hard problem. While the problem has already received much attention from researchers in the past, we argue in this thesis that a paradigm shift in self-reconfiguration leading to improved reconfiguration speeds can be achieved by changing the way a programmable matter object is defined and by considering a different set of assumptions about the problem, such as the presence of a dedicated self-reconfiguration platform. Therefore, we propose a framework and various algorithms implementing this new approach and show that current results corroborate the thesis.

Chapter 1 discussed the general frame to which the present research belongs, introduced the *Programmable Matter Consortium* and its current efforts towards the creation of programmable matter. The various contributions and involvement of the author to the project have also been highlighted. Then, Section 1.2 discussed the foundation of this work by introducing the *3D Catom*, the modular robotic model that it relies on, as well as *VisibleSim*, our dedicated simulation framework that is the basis of most of our experiments, and to which the author has heavily contributed. The geometry and capabilities of the *3D Catoms* have been thoroughly discussed, especially the motion constraints of the model. We have called attention to one motion constraint in particular, named the *bridging constraint*, which forces self-reconfiguration to occur according to a very restrictive construction order to avoid deadlocks, while also having a much deeper impact on what can be achieved using this model. Other motion constraints and resulting problems were also presented, such as the *remote blocking conundrum* and the *motion coordination challenge*, which, together, motivate the design our proposed approach to

self-reconfiguration.

We have then examined in Chapter 2 the literature on self-reconfiguration methods in three-dimensional lattice-based modular robotic and self-organizing particle systems. From this analysis, we attempted to provide an extensive survey of the current state of the art on these topics, with a thorough study of the strengths and weaknesses of these works from the standpoint of their application to programmable matter. From more than two dozen of those published works that we have classified into three different approaches to the self-reconfiguration problem, we aimed to dispense a comprehensive understanding of current challenges and directions to present and future research on this problem. Although several areas of improvement have been proposed, the inclusion of mechanical constraints and an emphasis on the robustness of reconfiguration methods stand out as particularly important for the prospect of realizing practical robot-based programmable matter systems. Furthermore, from the assessment that the hardware and software problems of self-reconfiguration cannot be considered in isolation, we argue that the three different approaches uncovered in Section 2.1 will have to merge in order to converge into practical solutions. Lastly, we contend that any progress in the field can only be the result of meaningful comparisons between proposed solutions, hence supporting the necessity for a unified evaluation system for existing and future self-reconfiguration algorithms. Even though there are many interesting ideas in the state of the art of the field, we felt that a new paradigm for self-reconfiguration was required in order to achieve algorithms with even faster reconfiguration speeds and that would better suit our object representation application.

Drawing upon our conclusion from our analysis of the state of the art of self-reconfiguration, Chapter 3 has 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 *scaffold*, 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 named a *sandbox*. These additions to the self-reconfiguration model promise to enable a generation of algorithms that are time- and motion-efficient, and simple with regard to motion planning — at the cost of a dependency on an external system (the sandbox), and a surplus of modules involved in the reconfiguration (though using a scaffolding actually requires fewer modules than the equivalent dense object). This trade-off might not be acceptable in some applications of modular robotics or programmable matter, but appears reasonable in the context of object representation. We have also introduced *the coating problem*, in modular robotics, which seeks to cover an object using one or several layers of modular micro-robots, as a way to compensate for the impaired visual fidelity of scaffolded objects. We have explained how the coating could be designed so that it can be built easily without having to sacrifice the mechanical structure of the object, thanks to

the addition of special modules named a *structural support*.

Chapter 4 addressed the first step of this new approach, the construction of the scaffold. It proposed a local and distributed algorithm for constructing the scaffold of a subclass of convex shapes coined *semi-convex* shapes from the sandbox underneath. We have shown that this can be done in sublinear 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 classes of shapes remains a future work, but the associated challenges have been introduced, and a number of lines of approach have been proposed nonetheless.

Now that the scaffold of objects could be built, it needed to be covered by modules to restore the original visual aspect of the object. The coating problem of modular robotics is, therefore, what we have tackled in Chapter 5, where a modular robot forming a scaffold of an object has to be covered with a thin layer of modules so that it appears dense to the eyes. We have proposed a method that provides an assembly order for constructing this coating from a reserve of modules in the form of a sandbox, using the Tucci algorithm and our Border Completion algorithm. Finally, we have provided simulation results with our coating method applied to various kinds of shapes, outlining its performance and current limitations, while showing that even in its current state it could be used to achieve the construction of a coating in a time linear to the number of modules in the envelope. Finally, we have shown that together, the sandbox, scaffold construction, and the coating could be used to greatly speed up the construction of modular robotic objects compared to the regular construction of dense shapes, and without altering their resulting external aspect. Though this algorithm has been designed for creating a coating that envelops a scaffold, this method could in principle work for any shape inside an FCC lattice, though the location and definition of the *structural supports* might need to be adapted to the problem at hand.

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* and *scaffolding*. 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.

In the coming sections, we will take a step back and reflect critically upon our present results and attempt to derive a number of insights and perspectives for pursuing this line of work or supporting similar research and the field of self-reconfiguration in general.

DISCUSSION

REFLECTION ON THE *3D Catom* MODEL AND CONSTRAINTS

We have seen in Section 1.2 that the *3D Catom* model had a geometry that produces *3D Catom* ensembles with an FCC lattice structure. As it turns out, this is one of the densest possible arrangements of matter. Coupled with the quasi-spherical geometry of *3D Catom*, this can create a very detailed representation of objects compared to other geometries, with smooth curves — recall Figure 1.2.

The high density of this FCC lattice geometry combined with the non-deformable nature of *3D Catoms* pose nevertheless serious complications with regard to the motion constraints of modules as introduced in Section 1.2.1. The *bridging constraint* may be the most infamous of these constraints due to its numerous and far-reaching effects.

We will see below that such constraint can be a serious impediment to both parallelism, robustness, and emergence in self-reconfiguration applications, stifling the range of practical solutions to self-reconfiguration problems.

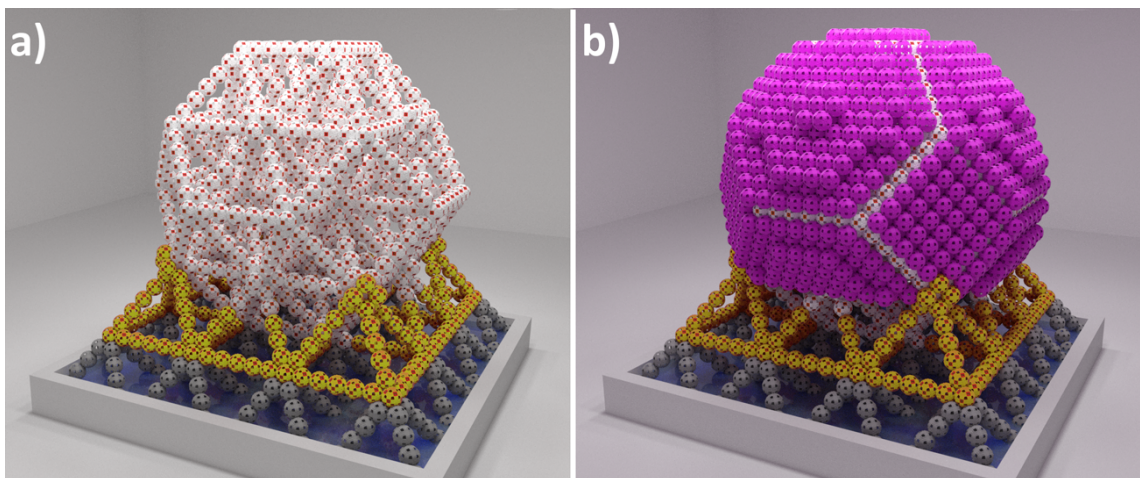


Figure 5.4: Original self-reconfiguration pipeline idea (artistic impression): using a temporary scaffold (yellow modules) both for mechanically supporting the growing structure and removing concavities during construction.

It turns out that our original intuition regarding the construction of scaffolded shapes in a sandbox was that concave shapes were a problem for self-reconfiguration, both in terms

of mechanical constraints on the growing shape, and because it would sever a direct vertical connection to the sandbox, which yields an optimal module throughput during reconfiguration and maximizes pipelining. For that reason, we investigated using a temporary scaffolding that would fill the concavities of the target shape during construction, and that would then be deconstructed at the end of self-reconfiguration. This required more modules than just building the scaffold (though less than the dense shape in most cases), but provided the highest reconfiguration speed throughout the structure and, we believe, supported the mechanical stability of intermediate configurations during self-reconfiguration. This is illustrated with the yellow temporary scaffold in Figure 5.4. Sadly, because of the bridging constraint, the temporary scaffold could never be deconstructed, and this promising idea had to be thrown away.

Furthermore, as the bridging constraint imposes a single direction of construction, this prevents us from starting the construction process from multiple opposite directions, which could further speed up self-reconfiguration, even though it would be harder to coordinate the construction.

Finally, robustness is a crucial aspect of self-reconfiguration, and this has to be factored in if large-scale self-reconfigurable systems are to become a real thing. It is guaranteed that in such systems consisting of hundreds to hundreds of thousands of modules, some sort of hardware malfunctions will happen... every minute. Whether it is a communication interface malfunction, a botched motion, or a more debilitating hardware issue, there needs to be some sort of mechanism in place in the system to remediate the presence of faulty modules and replace them with functional ones. Unfortunately, the bridging constraint means that if there is faulty module somewhere in the configuration, it is most likely impossible to extract and replace (even more in a scaffold as it turns out, where modules are arranged in lines). The faulty module would not be able to leave its position without displacing a significant number of other modules, and the replacement would not be able to fill the gap left by the faulty module without potentially displacing another large number of modules.

While deformable modules are possible, they would likely require moving parts to achieve deformability, which is another problematic feature of modules since it might increase the risks of hardware faults and could be more costly to assemble. Structures made of deformable modules are also at risk being less mechanically stable than with non-flexible modules, due to the semi-rigid state of deformable systems — though this is mostly speculative. Furthermore, such systems might turn out to be less capable in terms of possible motion, where a deformable *3D Catom* might not be able to reach as many adjacent positions due to the difference in the actuation system. A deformable catom has in fact been studied in (Piranda et al., 2020).

This is an important variable that roboticists must take into account when designing mod-

ular robotic systems, and this process of requirement feedback between roboticists and computer scientists is crucial to achieve more practical modular robotic applications, as discussed in Chapter 2.

We have also seen that the bridging constraint is not the only limit of the *3D Catom* hardware, as two problems, the *motion coordination challenge* and *remote blocking conundrum* were a consequence of the limited range of communications of *3D Catoms*. We have already discussed in Section 1.2.1 the potentially huge communication overhead that would come with every module motion to satisfy these constraints with only communication between neighbor modules. Building modules with a wider range of communication or at least some sort of extended neighborhood sensing would produce a much more powerful model for self-reconfiguration. It remains an open question whether such capabilities can reasonably be implemented at such scale. Wireless sensors seem problematic because of the interference that this would produce at the micro scale, hence some sort of global or semi-local communication buses might be worth investigating in this context.

Ultimately, systems that are deterministic, rigid, and collision-sensitive, might not be suited at all for self-reconfigurable matter, and future programmable matter systems may resemble more biological systems like ant colonies, that can tolerate much more errors from its constituent and that appear much more fluid and disorganized, than well-ordered systems that are more sensitive to uncertainty and errors. Only years and years of research will be able to tell.

THE ELEPHANTS IN THE SANDBOX

Ground and Underside Coating One particularly problematic aspect of object representation in a sandbox (and under our current model constrained under the bridging constraint) that has remained unaddressed up to this point is the fact that the underside of the object cannot be coated — and in fact remains tethered to the sandbox which might prevent manipulation by the user. While the sandbox could provide a way to disable the connection between the sandbox and the scaffold on demand so that the object might be manipulated (which would require an alternative power source if the matter is to remain active), adding the coating to the underside of the object seems harder to perform. The coating of the underside of the object cannot be performed according to the same process as the rest of the shape for two reasons: (i) because of the bridging constraint, it would not be possible to add all modules between the horizontal branches of the ground layer of the object (or between the sandbox branches if coating one layer below the scaffold instead); (ii) even if there was a way to do it, this would prevent any additional module from entering the scaffold from the sandbox, so this would have to be the last process of

the construction to take place, or the first of the deconstruction. The only realistic way to perform that coating of the underside of the object that we have been able to conjure would be to manually rotate the object on the sandbox and then perform the coating of the underside. Needless to say, this is highly undesirable as this requires an external intervention, which breaks autonomy. Changing the model to get rid of the bridging constraint could be a lot more desirable if the functional trade-off is not too bad.

Nonetheless, for the purpose of a 3D multi-sensory display system based on programmable matter, it might not be necessary to allow the represented scene to be detached from its substrate and manipulated by hand. Such a display without the detachability features would already be quite sufficient for providing unprecedented sensory experiences to users.

Object Fidelity and Special Cases On the topic of fidelity to the target object, and besides the absence of an underside coating, objects built using our method will in most case appear identical to their dense counterpart. When using a scaffold, however, the base unit of the shape changes from a 1-voxel module to a multi-voxel scaffold tile. This means that the scaffold is at best an approximation of the target shape or a lower-resolution version. In contrast, the coating preserves the original resolution of the object and is thus a great addition to the scaffold as it restores the original apparent resolution of the target shape. Nonetheless, this is a context where size matters — there are several caveats of our method that need to be kept in mind when building objects in this way:

- Very small objects (smaller or in the range of the dimensions of a scaffold tile) will have little to no internal structure, only coating, as there would not be enough internal space to fit a scaffold. The resulting object would be even closer in density to its dense version. Furthermore, the downsides of our method might be greater than the benefits in this context, and it is possible that a classic self-reconfiguration would be more appropriate.
- Very detailed objects, or at least objects with very small details will have many substructures with low internal structure (a partial absence of scaffold) and only coating. Much like in the previous item, the scaffold will approximate the shape, ignoring details smaller than a tile (though in most cases partial tiles will be built, but this is highly contextual), and the coating will then restore these details.
- In both previous cases, it remains to be determined what kind of mechanical stress the partial or total absence of scaffold would put on the overall structure. Regarding the former scenario, it is likely that with such small objects do not really require an internal scaffold since it can be expected to be much more mechanically stable due to its size.

Mechanical Validation of our Approach The most critical aspect of our approach that is missing from this research is a mechanical validation of the scaffold and coating. It is not sufficient that the use of a scaffolding appears mechanically equivalent, or at least mechanically stable enough for *3D Catom*-based programmable matter, this has to be mathematically and physically proven. This depends, however on the expected weight of future hardware *3D Catoms* and the size of their electrostatic actuators, but fortunately, the parameterizable nature of the scaffold with its tile branch length b could give us flexibility in designing the most stable and compact scaffold possible.

WHY COATING MAY ULTIMATELY NOT MATTER

Following our discussion on fidelity to the object to represent in the previous section, one may wonder what an acceptable level of detail is for programmable matter and our application. With current hardware *3D Catom* prototypes in the millimeter range, further advances in the design of microelectromechanical systems might enable the production of *3D Catoms* in the micrometer range in a few years. Is it not possible that given sufficient miniaturization of the hardware, a dense object and a scaffolded object might be visually identical? If the constituents of the matter are small enough, tile might also be small enough so that to the human eye, the details of the shape would be precise enough so that the coating phase would not be necessary. This is again highly speculative and preconditioned on the achievement of sufficient miniaturization of the modules.

Why Coating Will Most Likely Matter Regardless A counterpoint to the previous paragraph could, however, be that it might not be possible to produce visually satisfactory curves using the scaffold alone, something which is supported by the poor aspect of the curvatures of the mug at low resolution on Figure 1.2 and that of the cylinder on Figure 4.15. Furthermore, as mentioned in the fidelity discussion, some details of the object might not even be constructed at all if they are small enough, which also raises doubts that the coating phase could ever be avoided. All things considered, the most favorable solution would probably be to design a robotic where scaffolding and coating can be built in parallel and seamlessly, cutting down on assembly time even more.

FROM SCAFFOLD CONSTRUCTION TO SCAFFOLD SELF-RECONFIGURATION

Finally, while this work has laid the foundations for a new approach to self-reconfiguration, no actual shape-to-shape reconfiguration have has been produced yet. Therefore, we discuss in this section how this scaffold construction and coating framework can be extended to shape-to-shape reconfiguration.

It must be first noted that there is in principle no reason that would prevent our scaffold construction method, from being reversible — the exact inverse process of the construction of the scaffold can be used to deconstruct it, within the same algorithmic bounds. Similarly, while the exact reverse of our coating algorithm would require new primitives to replace the Tucci assembly routine, an equivalent top-down disassembly strategy in linear time should also in principle exist. In the worst case, then, updating the ensemble from an initial configuration \mathcal{I} to a goal configuration \mathcal{G} would mean:

1. Coating removal from \mathcal{I}
2. Scaffold disassembly of \mathcal{I}
3. Scaffold assembly of \mathcal{G}
4. Coating of the scaffold of \mathcal{G}

In terms of complexity, this would be equivalent to running our entire construction pipeline twice in a row (once for disassembly, once for reassembly), and thus would preserve of the $O(\sqrt[3]{N}) + O(N)$ complexity of the construction phase alone.

Nevertheless, entirely deconstructing the scaffold seems like a redundant and inefficient approach, and this can certainly be optimized. An advantage of the scaffold is that internal structure of any object we could build is similar. Thanks to that, self-reconfiguring from a shape \mathcal{I} to a shape \mathcal{G} could instead be the following, more efficient, process:

1. Coating removal from \mathcal{I}
2. Computing an optimal overlap of \mathcal{I} and \mathcal{G}
3. Scaffold disassembly of non-overlapping portions of \mathcal{I}
4. Scaffold assembly of non-overlapping portions of \mathcal{G}
5. Coating of the scaffold of \mathcal{G}

In other terms, we could preserve the parts of the original scaffold that is common to both the initial and goal scaffolds, and only disassemble and reassemble the non-common parts. This would be very simple on non-convex shapes, but harder to coordinate in the general case as flows of modules from multiple directions might converge into the tiles vertically connected to the sandbox. In both cases, however, an important preparation phase for this scaffold reconfiguration would be finding the best overlap between the initial and goal shape, in order to minimize the number of modules to discard from \mathcal{I} and supply to \mathcal{G} . This overlap problem concerns both the positioning of the goal 3D object in the regular grid, as well as its orientation. Nonetheless, this is not specific to scaffold

reconfiguration, but this is a general problem in self-reconfiguration, as computing these hyper-parameters of self-reconfiguration could tremendously reduce the subsequent reconfiguration efforts.

The scaffold-to-scaffold self-reconfiguration problem can be thought of as a *resource allocation problem*, i.e., finding the optimal flow of modules between the sandbox, areas that must be grown, and areas that must be discarded. 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.

PERSPECTIVES

In this final section, and based on the various critical discussions from the previous section and past chapters, we compile a list of perspectives for improving our work and pushing the current limits of modular robotics systems and self-reconfiguration even further.

Regarding our self-reconfiguration based on sandboxing and scaffolding, here is what must to be done for our approach to be able to fully materialize:

Generalization, Improvements, and Self-Reconfiguration

- Regarding the current results, there are a number of optimizations that could be explored, including prioritizing the construction of branches that are connected to children tiles in the construction scheduling of the tile, as well as investigating the impact of displacing the seed tiles at the center of the shape, which would lead to a centrifugal growth and might help increase the parallelism of the method even further, though this is unlikely to improve on the current cubic root worst-case reconfiguration time.
- Implementing the generalized version of our scaffold-construction algorithm, and creating a taxonomy of scaffold geometries with a mathematical analysis of the expected worst-case reconfiguration time for each class of shapes.
- 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. A more systematic and robust approach is thus needed. This could either be replaced by an alternative and better-suited

motion planning method or benefit from improvements in design and compactness. It would be good to also prove the convergence and universality of the method once this is done.

- Design a coating assembly method that can coat any shape or scaffold, preferably achieving a high degree of parallelism in the construction of the scaffold. Accordingly, researching an adequate motion planning and coordination algorithm for implementing this assembly plan using sandbox-fed *3D Catoms*.
- Extending our method to shape-to-shape (or rather scaffold-to-scaffold) self-reconfiguration, with the goal to minimize the amount of matter that has to be displaced. Once this is done, perform a thorough comparison of the results of this method to state-of-the-art self-reconfiguration algorithms, studying metrics such as reconfiguration time, number of individual module motions, communication volume, raw matter usage (number of modules), and robustness.
- It would also be useful to study how this approach can be adapted to other models and in systems that reside in different lattices. This would maximize the usefulness of this approach, and perhaps show that it is preferable with some models but not others.

Feasibility

- A mechanical validation of our scaffold model and a study of the stability of a scaffold-based object made of *3D Catom* compared to its dense counterpart.
- Accordingly, determining the range of realistic values for the parameter b of our scaffold.
- Engineering research on the feasibility and design of a sandbox system has described in our work.
- Express self-reconfiguration time in terms of confidence intervals and study error distributions and the propagation of errors, to better report what practical results could be achieved with our method.

General Perspectives Regarding the general self-reconfiguration problem, researching the *overlap problem* between the initial and goal shape, and studying the impact of these hyper-parameters and others on the ensuing self-reconfiguration are essential. We have also provided in Section 2.3.3 a number of guidelines and perspectives for the field of 3D self-reconfiguration algorithms that we hope will be helpful to guide future research. Nonetheless, we would like to reiterate here that in order to accelerate progress the field

will need a real benchmark to test out and compare self-reconfiguration algorithms, as it is still extremely tricky to produce meaningful comparisons between published research works on the topic.

PERSONAL PUBLICATIONS

- Thalamy, P., Piranda, B., et 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*, pages 140–148, Montreal QC, Canada, doi: [10.5555/3306127.3331685](https://doi.org/10.5555/3306127.3331685).
- Thalamy, P., Piranda, B., et Bourgeois, J. (2019b). **A survey of autonomous self-reconfiguration methods for robot-based programmable matter**. *Robotics and Autonomous Systems*, 120:103242, doi: [10.1016/j.robot.2019.07.012](https://doi.org/10.1016/j.robot.2019.07.012).
- Thalamy, P., Piranda, B., et Bourgeois, J. (2020a). **3D Coating Self-assembly for modular robotic scaffolds**. In *In Proceedings 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, NV, USA.
- Thalamy, P., Piranda, B., Lassabe, F., et 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)*, pages 4840–4846, doi: [10.1109/IROS40897.2019.8967775](https://doi.org/10.1109/IROS40897.2019.8967775).
- Thalamy, P., Piranda, B., Lassabe, F., et Bourgeois, J. (2020b). **Deterministic scaffold assembly by self-reconfiguring micro-robotic swarms**. *Swarm and Evolutionary Computation*, 58:100722, doi: <https://doi.org/10.1016/j.swevo.2020.100722>.

BIBLIOGRAPHY

- Ahmadzadeh, H., et Masehian, E. (2015). **Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization**. *Artificial Intelligence*, 223:27–64, doi: [10.1016/j.artint.2015.02.004](https://doi.org/10.1016/j.artint.2015.02.004).
- Ahmadzadeh, H., Masehian, E., et Asadpour, M. (2016). **Modular Robotic Systems: Characteristics and Applications**. *Journal of Intelligent & Robotic Systems*, 81(3):317–357, doi: [10.1007/s10846-015-0237-8](https://doi.org/10.1007/s10846-015-0237-8).
- Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M. J., et Peralta, R. (2006). **Computation in networks of passively mobile finite-state sensors**. *Distributed Computing*, 18(4):235–253, doi: [10.1007/s00446-005-0138-3](https://doi.org/10.1007/s00446-005-0138-3).
- Ashley-Rollman, M. P., Pillai, P., et Goodstein, M. L. (2011). **Simulating multi-million-robot ensembles**. In *2011 IEEE International Conference on Robotics and Automation*, pages 1006–1013, Shanghai, China. IEEE, doi: [10.1109/ICRA.2011.5979807](https://doi.org/10.1109/ICRA.2011.5979807).
- Barraquand, J., et Latombe, J.-C. (1991). **Robot Motion Planning: A Distributed Representation Approach**. *The International Journal of Robotics Research*, 10(6):628–649, doi: [10.1177/027836499101000604](https://doi.org/10.1177/027836499101000604).
- Berman, S., Fekete, S. P., Patitz, M. J., et Scheideler, C. (2019). **Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 18331)**. *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany*, doi: [10.4230/dagrep.8.8.48](https://doi.org/10.4230/dagrep.8.8.48).
- Bie, D., Wang, Y., Zhang, Y., Liu, C., zhao, J., et Zhu, Y. (2018). **Parametric L-systems-based modeling self-reconfiguration of modular robots in obstacle environments**. *International Journal of Advanced Robotic Systems*, 15(1):1729881418754477, doi: [10.1177/1729881418754477](https://doi.org/10.1177/1729881418754477).
- Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., et Nguyen, T. (2005). **Programmable parts: a demonstration of the grammatical approach to self-organization**. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3684–3691, Edmonton, Alta., Canada. IEEE, doi: [10.1109/IROS.2005.1545375](https://doi.org/10.1109/IROS.2005.1545375).
- Bourgeois, J., Piranda, B., Naz, A., Boillot, N., Mabed, H., Dhoutaut, D., Tucci, T., et Lakhlef, H. (2016). **Programmable matter as a cyber-physical conjugation**. In Sys-

- tems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on, pages 002942–002947. IEEE, doi: [10.1109/SMC.2016.7844687](https://doi.org/10.1109/SMC.2016.7844687).
- Bresenham, J. E. (1965). **Algorithm for computer control of a digital plotter**. *IBM Systems Journal*, 4(1):25–30, doi: [10.1147/sj.41.0025](https://doi.org/10.1147/sj.41.0025).
- Butler, Z., Kotay, K., Rus, D., et 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*, volume 1, pages 809–816. IEEE.
- Butler, Z., et Rus, D. (2003). **Distributed Planning and Control for Modular Robots with Unit-Compressible Modules**. *The International Journal of Robotics Research*, pages 699–715, doi: [10.1177/02783649030229002](https://doi.org/10.1177/02783649030229002).
- Cannon, S., Daymude, J. J., Randall, D., et Richa, A. W. (2016). **A Markov Chain Algorithm for Compression in Self-Organizing Particle Systems**. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 279–288. ACM Press, doi: [10.1145/2933057.2933107](https://doi.org/10.1145/2933057.2933107).
- Carpin, S., Lewis, M., Wang, J., Balakirsky, S., et Scrapper, C. (2007). **USARSim: a robot simulator for research and education**. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1400–1405, Rome, Italy. IEEE, doi: [10.1109/ROBOT.2007.363180](https://doi.org/10.1109/ROBOT.2007.363180). ISSN: 1050-4729.
- Castano, A., Shen, W.-M., et Will, P. (2000). **CONRO: Towards deployable robots with inter-robots metamorphic capabilities**. *Autonomous Robots*, 8(3):309–324, doi: [10.1023/A:1008985810481](https://doi.org/10.1023/A:1008985810481).
- Christensen, D., Brandt, D., Støy, K., et Schultz, U. (2008). **A unified simulator for Self-Reconfigurable Robots**. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 870–876, Nice. IEEE, doi: [10.1109/IROS.2008.4650757](https://doi.org/10.1109/IROS.2008.4650757).
- Collins, T., Ranasinghe, N. O., et Wei-Min Shen (2013). **ReMod3D: A high-performance simulator for autonomous, self-reconfigurable robots**. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4281–4287, Tokyo. IEEE, doi: [10.1109/IROS.2013.6696970](https://doi.org/10.1109/IROS.2013.6696970).
- Collins, T., et Shen, W.-M. (2016). **ReBots: A Drag-and-drop High-Performance Simulator for Modular and Self-Reconfigurable Robots**. Technical Report 714, University of Southern California, Information Sciences Institute.
- Davey, J., Kwok, N., et Yim, M. (2012). **Emulating self-reconfigurable robots - design of the SMORES system**. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4464–4469, Vilamoura-Algarve, Portugal. IEEE, doi: [10.1109/IROS.2012.6385845](https://doi.org/10.1109/IROS.2012.6385845).

- Daymude, J. J., Derakhshandeh, Z., Gmyr, R., Porter, A., Richa, A. W., Scheideler, C., et Strothmann, T. (2018). **On the Runtime of Universal Coating for Programmable Matter**. *Natural Computing*, 17(1):81–96, doi: [10.1007/s11047-017-9658-6](https://doi.org/10.1007/s11047-017-9658-6). arXiv: 1606.03642.
- Daymude, J. J., Hinnenthal, K., Richa, A. W., et Scheideler, C. (2019). **Computing by Programmable Particles**. In Flocchini, P., Prencipe, G., et Santoro, N., editors, *Distributed Computing by Mobile Entities*, volume 11340, pages 615–681. Springer International Publishing, Cham, doi: [10.1007/978-3-030-11072-7_22](https://doi.org/10.1007/978-3-030-11072-7_22).
- Daymude, J. J., Richa, A. W., et Weber, J. W. (2020). **Bio-Inspired Energy Distribution for Programmable Matter**. *arXiv:2007.04377 [cs]*. arXiv: 2007.04377.
- Deng, Y., Hua, Y., Napp, N., et Petersen, K. (2019). **Scalable Compiler for the TERMES Distributed Assembly System**. In Correll, N., Schwager, M., et Otte, M., editors, *Distributed Autonomous Robotic Systems*, volume 9, pages 125–138, Boulder, CO, USA. Springer International Publishing, doi: [10.1007/978-3-030-05816-6_9](https://doi.org/10.1007/978-3-030-05816-6_9).
- Derakhshandeh, Z., Gmyr, R., Richa, A. W., Scheideler, C., et Strothmann, T. (2015a). **An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems**. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, pages 1–2, doi: [10.1145/2800795.2800829](https://doi.org/10.1145/2800795.2800829).
- Derakhshandeh, Z., Gmyr, R., Richa, A. W., Scheideler, C., et Strothmann, T. (2016). **Universal Shape Formation for Programmable Matter**. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–299, doi: [10.1145/2935764.2935784](https://doi.org/10.1145/2935764.2935784).
- Derakhshandeh, Z., Gmyr, R., Richa, A. W., Scheideler, C., et Strothmann, T. (2017). **Universal coating for programmable matter**. *Theoretical Computer Science*, 671:56–68, doi: [10.1016/j.tcs.2016.02.039](https://doi.org/10.1016/j.tcs.2016.02.039).
- Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R., Richa, A. W., et Scheideler, C. (2015b). **Leader Election and Shape Formation with Self-organizing Programmable Matter**. In *DNA Computing and Molecular Programming*, volume 9211, pages 117–132. Springer International Publishing, Cham, doi: [10.1007/978-3-319-21999-8_8](https://doi.org/10.1007/978-3-319-21999-8_8).
- Dewey, D. J., Ashley-Rollman, M. P., Rosa, M. D., Goldstein, S. C., Mowry, T. C., Srinivasa, S. S., Pillai, P., et 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*, pages 1338–1345, doi: [10.1109/IROS.2008.4651094](https://doi.org/10.1109/IROS.2008.4651094).

- Dhoutaut, D., Piranda, B., et Bourgeois, J. (2013). **Efficient Simulation of Distributed Sensing and Control Environments**. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCoM), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 452–459. IEEE, doi: [10.1109/GreenCom-iThings-CPSCoM.2013.93](https://doi.org/10.1109/GreenCom-iThings-CPSCoM.2013.93).
- Di Luna, G. A., Flocchini, P., Prencipe, G., Santoro, N., et Viglietta, G. (2018a). **Line Recovery by Programmable Particles**. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pages 1–10. ACM Press, doi: [10.1145/3154273.3154309](https://doi.org/10.1145/3154273.3154309).
- Di Luna, G. A., Flocchini, P., Santoro, N., Viglietta, G., et Yamauchi, Y. (2018b). **Shape Formation by Programmable Particles**. In Aspnes, J., Bessani, A., Felber, P., et Leitão, J., editors, *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, volume 95 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:16, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, doi: [10.4230/LIPIcs.OPODIS.2017.31](https://doi.org/10.4230/LIPIcs.OPODIS.2017.31).
- Doty, D. (2012). **Theory of algorithmic self-assembly**. *Communications of the ACM*, 55(12):78, doi: [10.1145/2380656.2380675](https://doi.org/10.1145/2380656.2380675).
- Fekete, S., Richa, A. W., Römer, K., et Scheideler, C. (2016). **Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271)**. *Dagstuhl Reports*, 6(7):1–14, doi: [10.4230/DagRep.6.7.1](https://doi.org/10.4230/DagRep.6.7.1).
- Fitch, R., et Butler, Z. (2008). **Million Module March: Scalable Locomotion for Large Self-Reconfiguring Robots**. *The International Journal of Robotics Research*, 27(3-4):331–343, doi: [10.1177/0278364907085097](https://doi.org/10.1177/0278364907085097).
- Fitch, R., Butler, Z., et 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*, pages 2460–2467, doi: [10.1109/IROS.2003.1249239](https://doi.org/10.1109/IROS.2003.1249239).
- Fitch, R., Butler, Z., et Rus, D. (2005). **Reconfiguration Planning Among Obstacles for Heterogeneous Self-Reconfiguring Robots**. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 117–124, doi: [10.1109/ROBOT.2005.1570106](https://doi.org/10.1109/ROBOT.2005.1570106).
- Fitch, R., Butler, Z., et Rus, D. (2007). **In-Place Distributed Heterogeneous Reconfiguration Planning**. In *Distributed Autonomous Robotic Systems 6*, pages 159–168, doi: [10.1007/978-4-431-35873-2_16](https://doi.org/10.1007/978-4-431-35873-2_16).

- Fitch, R., et McAllister, R. (2013). **Hierarchical Planning for Self-reconfiguring Robots Using Module Kinematics**. In *Distributed Autonomous Robotic Systems 10*, pages 477–490, doi: [10.1007/978-3-642-32723-0_34](https://doi.org/10.1007/978-3-642-32723-0_34).
- Fitch, R. C. (2004). **Heterogeneous self-reconfiguring robotics**. PhD thesis, Dartmouth College.
- Fukuda, T., et Kawauchi, Y. (1990). **Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator**. In *Proceedings., IEEE International Conference on Robotics and Automation*, volume 1, pages 662–667. IEEE Comput. Soc. Press, doi: [10.1109/ROBOT.1990.126059](https://doi.org/10.1109/ROBOT.1990.126059).
- Gerkey, B. P., Vaughan, R. T., et Howard, A. (2003). **The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems**. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323.
- Gilpin, K., Knaian, A., et Rus, D. (2010). **Robot pebbles: One centimeter modules for programmable matter through self-disassembly**. In *2010 IEEE International Conference on Robotics and Automation*, pages 2485–2492. IEEE, doi: [10.1109/ROBOT.2010.5509817](https://doi.org/10.1109/ROBOT.2010.5509817).
- Gilpin, K., Kotay, K., Rus, D., et Vasilescu, I. (2008). **Miche: Modular shape formation by self-disassembly**. *The International Journal of Robotics Research*, 27(3-4):345–372.
- Gmyr, R., Hinnenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C., et Strothmann, T. (2019). **Forming tile shapes with simple robots**. *Natural Computing*, doi: [10.1007/s11047-019-09774-2](https://doi.org/10.1007/s11047-019-09774-2).
- Goldstein, S. C., Campbell, J. D., et Mowry, T. C. (2005). **Programmable matter**. *Computer*, 38(6):99–101.
- Goldstein, S. C., et Mowry, T. C. (2004). **Claytronics: A scalable basis for future robots**. In *RoboSphere 2004*, Moffett Field, CA.
- Goldstein, S. C., Mowry, T. C., Campbell, J. D., Ashley-Rollman, M. P., De Rosa, M., Funiak, S., Hoburg, J. F., Karagozler, M. E., Kirby, B., Lee, P., et others (2009). **Beyond audio and video: Using claytronics to enable pario**. *AI Magazine*, 30(2):29.
- Gorbenko, A. A., et Popov, V. Y. (2012). **Programming for modular reconfigurable robots**. *Programming and Computer Software*, 38(1):13–23, doi: [10.1134/S0361768812010033](https://doi.org/10.1134/S0361768812010033).
- Haghighat, B., et Martinoli, A. (2017). **Automatic synthesis of rulesets for programmable stochastic self-assembly of rotationally symmetric robotic modules**. *Swarm Intelligence*, 11(3-4):243–270, doi: [10.1007/s11721-017-0139-4](https://doi.org/10.1007/s11721-017-0139-4).

- Hamann, H. (2018). **Swarm Robotics: A Formal Approach**. Springer International Publishing, doi: [10.1007/978-3-319-74528-2](https://doi.org/10.1007/978-3-319-74528-2).
- Hamann, H., Stradner, J., Schmickl, T., et Crailsheim, K. (2010). **A hormone-based controller for evolutionary multi-modular robotics: From single modules to gait learning**. In *IEEE Congress on Evolutionary Computation*, pages 1–8, Barcelona, Spain. IEEE, doi: [10.1109/CEC.2010.5585994](https://doi.org/10.1109/CEC.2010.5585994).
- Hamann, H., et Wörn, H. (2007). **EMBODIED COMPUTATION**. *Parallel Processing Letters*, 17(03):287–298, doi: [10.1142/S0129626407003022](https://doi.org/10.1142/S0129626407003022).
- Hawkes, E., An, B., Benbernou, N. M., Tanaka, H., Kim, S., Demaine, E. D., Rus, D., et Wood, R. J. (2010). **Programmable matter by folding**. *Proceedings of the National Academy of Sciences*, 107(28):12441–12445.
- Hou, F., et Shen, W. M. (2010). **On the complexity of optimal reconfiguration planning for modular reconfigurable robots**. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, doi: [10.1109/ROBOT.2010.5509642](https://doi.org/10.1109/ROBOT.2010.5509642).
- Hołobut, P., et 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*, pages 2103–2109, doi: [10.1109/ICRA.2017.7989242](https://doi.org/10.1109/ICRA.2017.7989242).
- Ijspeert, A. J., Martinoli, A., Billard, A., et Gambardella, L. M. (2001). **Collaboration through the exploitation of local interactions in autonomous collective robotics: the stick pulling experiment**. *Autonomous Robots*, 11(2):149–171.
- Jackson, J. (2008). **Microsoft robotics studio: A technical introduction**. *Robotics & Automation Magazine, IEEE*, 14:82 – 87, doi: [10.1109/M-RA.2007.905745](https://doi.org/10.1109/M-RA.2007.905745).
- Jorgensen, M. W., Ostergaard, E. H., et Lund, H. H. (2004). **Modular ATRON: modules for a self-reconfigurable robot**. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 2068–2073, Sendai, Japan, doi: [10.1109/IROS.2004.1389702](https://doi.org/10.1109/IROS.2004.1389702). ISSN: null.
- Kamimura, A., Yoshida, E., Murata, S., Tomita, K., et Kokaji, S. (2002). **A Self-Reconfigurable Modular Robot (MTRAN) – Hardware and Motion Generation Software –**. In *5th International Symposium on Distributed Autonomous Robotic Systems*, page 10.
- 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*, pages 3276–3283, doi: [10.1109/IROS.2015.7353832](https://doi.org/10.1109/IROS.2015.7353832).

- Kawano, H. (2016). **Full-resolution reconfiguration planning for heterogeneous cube-shaped modular robots with only sliding motion primitive**. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 5222–5229, doi: [10.1109/ICRA.2016.7487730](https://doi.org/10.1109/ICRA.2016.7487730).
- Kawano, H. (2017). **Tunneling-based self-reconfiguration of heterogeneous sliding cube-shaped modular robots in environments with obstacles**. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 825–832, doi: [10.1109/ICRA.2017.7989100](https://doi.org/10.1109/ICRA.2017.7989100).
- Ke, Y., Ong, L. L., Shih, W. M., et Yin, P. (2012). **Three-Dimensional Structures Self-Assembled from DNA Bricks**. *Science*, 338(6111):1177–1183, doi: [10.1126/science.1227268](https://doi.org/10.1126/science.1227268).
- Kernbach, S., Hamann, H., Stradner, J., Thenius, R., Schmickl, T., Crailsheim, K., Rossum, A. v., Sebag, M., Bredeche, N., Yao, Y., Baele, G., Peer, Y. V. d., Timmis, J., Mohktar, M., Tyrrell, A., Eiben, A., McKibbin, S., Liu, W., et Winfield, A. F. (2009). **On Adaptive Self-Organization in Artificial Robot Organisms**. In *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 33–43, Athens, Greece. IEEE, doi: [10.1109/ComputationWorld.2009.9](https://doi.org/10.1109/ComputationWorld.2009.9).
- Kernbach, S., Meister, E., Schlachter, F., Jebens, K., Szymanski, M., Liedke, J., Laneri, D., Winkler, L., Schmickl, T., Thenius, R., Corradi, P., et Ricotti, L. (2008). **Symbiotic Robot Organisms: REPLICATOR and SYMBRION Projects**. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems, PerMIS '08*, pages 62–69, New York, NY, USA. Association for Computing Machinery, doi: [10.1145/1774674.1774685](https://doi.org/10.1145/1774674.1774685). event-place: Gaithersburg, Maryland.
- Kim, J.-W., Kim, J.-H., et Deaton, R. (2011). **DNA-Linked Nanoparticle Building Blocks for Programmable Matter**. *Angewandte Chemie International Edition*, 50(39):9185–9190, doi: [10.1002/anie.201102342](https://doi.org/10.1002/anie.201102342).
- Knaian, A. N., Cheung, K. C., Lobovsky, M. B., Oines, A. J., Schmidt-Neilsen, P., et Gershenfeld, N. A. (2012). **The Milli-Motein: A self-folding chain of programmable matter with a one centimeter module pitch**. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1447–1453, Vilamoura-Algarve, Portugal. IEEE, doi: [10.1109/IROS.2012.6385904](https://doi.org/10.1109/IROS.2012.6385904).
- Koenig, N., et Howard, A. (2004). **Design and use paradigms for gazebo, an open-source multi-robot simulator**. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154, Sendai, Japan. IEEE, doi: [10.1109/IROS.2004.1389727](https://doi.org/10.1109/IROS.2004.1389727).

- Kotay, K. D., et 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*, volume 3, pages 2184–2193, doi: [10.1109/IROS.2000.895294](https://doi.org/10.1109/IROS.2000.895294).
- Kramer, J., et Scheutz, M. (2007). **Development environments for autonomous mobile robots: A survey**. *Autonomous Robots*, 22(2):101–132, doi: [10.1007/s10514-006-9013-8](https://doi.org/10.1007/s10514-006-9013-8).
- Kurokawa, H., Murata, S., Yoshida, E., Tomita, K., et 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*, page 6.
- Lengiewicz, J., et Holobut, P. (2019). **Efficient collective shape shifting and locomotion of massively-modular robotic structures**. *Auton. Robots*, 43(1):97–122, doi: [10.1007/s10514-018-9709-6](https://doi.org/10.1007/s10514-018-9709-6).
- Lyder, A., Garcia, R., et Støy, K. (2008). **Mechanical design of odin, an extendable heterogeneous deformable modular robot**. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 883–888, Nice. IEEE, doi: [10.1109/IROS.2008.4650888](https://doi.org/10.1109/IROS.2008.4650888).
- Martinoli, A., Easton, K., et Agassounon, W. (2004). **Modeling Swarm Robotic Systems: a Case Study in Collaborative Distributed Manipulation**. *The International Journal of Robotics Research*, 23(4-5):415–436, doi: [10.1177/0278364904042197](https://doi.org/10.1177/0278364904042197). – eprint: <https://doi.org/10.1177/0278364904042197>.
- Michail, O., Skretas, G., et Spirakis, P. G. (2017). **On the Transformation Capability of Feasible Mechanisms for Programmable Matter**. *arXiv:1703.04381 [cs]*. arXiv: 1703.04381.
- Michel, O. (2004). **Webots: Professional Mobile Robot Simulation**. *Journal of Advanced Robotics Systems*, 1(1):39–42.
- Miyashita, S., Guitron, S., Li, S., et Rus, D. (2017). **Robotic metamorphosis by origami exoskeletons**. *Science Robotics*, 2(10):eaao4369, doi: [10.1126/scirobotics.aao4369](https://doi.org/10.1126/scirobotics.aao4369).
- Moeckel, R., Jaquier, C., Drapel, K., Dittrich, E., Upegui, A., et Ijspeert, A. (2006). **YaMoR and Bluemove — An Autonomous Modular Robot with Bluetooth Interface for Exploring Adaptive Locomotion**. In Tokhi, M. O., Virk, G. S., et Hossain, M. A., editors, *Climbing and Walking Robots*, pages 685–692. Springer Berlin Heidelberg, Berlin, Heidelberg, doi: [10.1007/3-540-26415-9_82](https://doi.org/10.1007/3-540-26415-9_82).

- Mondada, F., Bonani, M., Raemy, X., Pugh, J., Cianci, C., Klapotocz, A., Magnenat, S., Zufferey, J.-C., Floreano, D., et Martinoli, A. (2009). **The e-puck, a Robot Designed for Education in Engineering**. In *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, volume 1, pages 59–65, Castelo Branco, Portugal.
- Naz, A. (2017). **Distributed Algorithms for Large-Scale Robotic Ensembles: Centrality, Synchronization and Self-reconfiguration**. PhD Thesis, FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS.
- Naz, A., Piranda, B., Bourgeois, J., et Goldstein, S. C. (2016a). **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.
- Naz, A., Piranda, B., Goldstein, S. C., et Bourgeois, J. (2015). **ABC-Center: Approximate-center election in modular robots**. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2951–2957, Hamburg, Germany. IEEE, doi: [10.1109/IROS.2015.7353784](https://doi.org/10.1109/IROS.2015.7353784).
- Naz, A., Piranda, B., Goldstein, S. C., et Bourgeois, J. (2016b). **A Time Synchronization Protocol for Modular Robots**. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 109–118, Heraklion. IEEE, doi: [10.1109/PDP.2016.73](https://doi.org/10.1109/PDP.2016.73).
- Naz, A., Piranda, B., Tucci, T., Copen Goldstein, S., et Bourgeois, J. (2018). **Network Characterization of Lattice-Based Modular Robots with Neighbor-to-Neighbor Communications**. In *Distributed Autonomous Robotic Systems*, volume 6, pages 415–429, Cham. Springer International Publishing.
- Oung, R., et D’Andrea, R. (2011). **The distributed flight array**. *Mechatronics*, 21(6):908–917.
- Pannuto, P., Lee, Y., Foo, Z., Blaauw, D., et Dutta, P. (2013). **M3: a mm-scale wireless energy harvesting sensor platform**. In *Proceedings of the 1st International Workshop on Energy Neutral Sensing Systems*, page 17. ACM.
- Park, M., Chitta, S., Teichman, A., et Yim, M. (2008). **Automatic Configuration Recognition Methods in Modular Robots**. *The International Journal of Robotics Research*, 27(3-4):403–421, doi: [10.1177/0278364907089350](https://doi.org/10.1177/0278364907089350).
- Patitz, M. J. (2014). **An introduction to tile-based self-assembly and a survey of recent results**. *Natural Computing*, 13(2):195–224, doi: [10.1007/s11047-013-9379-4](https://doi.org/10.1007/s11047-013-9379-4).

- Pescher, F., Napp, N., Piranda, B., et Bourgeois, J. (2020). **GAPCoD: A Generic Assembly Planner by Constrained Disassembly**. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, Auckland, New Zealand.
- Pincioli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Birattari, M., Gambardella, L. M., et Dorigo, M. (2012). **ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems**. *Swarm Intelligence*, 6(4):271–295, doi: [10.1007/s11721-012-0072-5](https://doi.org/10.1007/s11721-012-0072-5).
- Piranda, B. (2016). **VisibleSim: Your simulator for Programmable Matter**. In Römer, K., Scheideler, C., Fekete, S. P., et Richa, A. W., editors, *Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271)*, volume 6 of *Dagstuhl Reports*, page 12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. doi:10.4230/DagRep.6.7.1.
- Piranda, B., et Bourgeois, J. (2018). **Designing a quasi-spherical module for a huge modular robot to create programmable matter**. *Autonomous Robots*, 42(8):1619–1633, doi: [10.1007/s10514-018-9710-0](https://doi.org/10.1007/s10514-018-9710-0).
- Piranda, B., et Bourgeois, J. (2020). **Datom: A Deformable modular robot for building self-reconfigurable programmable matter**. *arXiv:2005.03402 [cs]*. arXiv: 2005.03402.
- Piranda, B., Laurent, G. J., Bourgeois, J., Clévy, C., Möbes, S., et Fort-Piat, N. L. (2013). **A new concept of planar self-reconfigurable modular robot for conveying microparts**. *Mechatronics*, 23(7):906–915, doi: [10.1016/j.mechatronics.2013.08.009](https://doi.org/10.1016/j.mechatronics.2013.08.009).
- Rister, B. D., Campbell, J., Pillai, P., et Mowry, T. C. (2007). **Integrated Debugging of Large Modular Robot Ensembles**. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2227–2234, Rome, Italy. IEEE, doi: [10.1109/ROBOT.2007.363651](https://doi.org/10.1109/ROBOT.2007.363651). ISSN: 1050-4729.
- Romanishin, J. W., Gilpin, K., et Rus, D. (2013). **M-blocks: Momentum-driven, magnetic modular robots**. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4288–4295, Tokyo. IEEE, doi: [10.1109/IROS.2013.6696971](https://doi.org/10.1109/IROS.2013.6696971).
- Rubenstein, M., Ahler, C., et Nagpal, R. (2012). **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. IEEE, doi: [10.1109/ICRA.2012.6224638](https://doi.org/10.1109/ICRA.2012.6224638).
- Rus, D., et Vona, M. (2000). **A physical implementation of the self-reconfiguring crystalline robot**. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat.*

- No.00CH37065), volume 2, pages 1726–1733, San Francisco, CA, USA. IEEE, doi: [10.1109/ROBOT.2000.844845](https://doi.org/10.1109/ROBOT.2000.844845).
- Rus, D., et Vona, M. (2001). **Crystalline Robots: Self-Reconfiguration with Compressible Unit Modules.** *Autonomous Robots*, 10(1):107–124, doi: [10.1023/A:1026504804984](https://doi.org/10.1023/A:1026504804984).
- Salemi, B., Moll, M., et Shen, W.-m. (2006). **SUPERBOT: A Deployable, Multi-Functional, and Modular Self-Reconfigurable Robotic System.** In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3636–3641, Beijing, China. IEEE, doi: [10.1109/IROS.2006.281719](https://doi.org/10.1109/IROS.2006.281719).
- Spröwitz, A., Laprade, P., Bonardi, S., Mayer, M., Moeckel, R., Mudry, P. A., et Ijspeert, A. J. (2010). **Roombots—Towards decentralized reconfiguration with self-reconfiguring modular robotic metamodules.** In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, doi: [10.1109/IROS.2010.5649504](https://doi.org/10.1109/IROS.2010.5649504).
- Spröwitz, A., Moeckel, R., Vespignani, M., Bonardi, S., et Ijspeert, A. (2014). **Roombots: A hardware perspective on 3D self-reconfiguration and locomotion with a homogeneous modular robot.** *Robotics and Autonomous Systems*, 62(7):1016–1033, doi: [10.1016/j.robot.2013.08.011](https://doi.org/10.1016/j.robot.2013.08.011).
- Stoy, K., Brandt, D., et Christensen, D. (2010). **Self-Reconfigurable Robots: An Introduction.** Intelligent Robotics and Autonomous Agents series. The MIT Press, 195.
- Stoy, K., et Kurokawa, H. (2011). **Current Topics in Classic Self-reconfigurable Robot Research.** In *In Proceedings of the IROS Workshop on Reconfigurable Modular Robotics: Challenges of Mechatronics and Bio-Chemo-Hybrid Systems*, page 4, San Francisco, CA, USA.
- Stoy, K., et Nagpal, R. (2004). **Self-Reconfiguration Using Directed Growth.** In *Int'l Symposium on Distributed Autonomous Robotic Systems*.
- Støy, K. (2004). **Emergent control of self-reconfigurable robots.** PhD thesis, University of Southern Denmark.
- Støy, K. (2006). **Using cellular automata and gradients to control self-reconfiguration.** *Robotics and Autonomous Systems*, 54(2):135 – 141, doi: <https://doi.org/10.1016/j.robot.2005.09.017>.
- Støy, K., et Nagpal, R. (2007). **Self-Reconfiguration Using Directed Growth.** In *Distributed Autonomous Robotic Systems 6*, pages 3–12, doi: [10.1007/978-4-431-35873-2_1](https://doi.org/10.1007/978-4-431-35873-2_1).

- Suh, J. W., Homans, S. B., et Yim, M. (2002). **Telecubes: Mechanical design of a module for self-reconfigurable robotics**. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 4095–4101. IEEE.
- Swissler, P., et Rubenstein, M. (2018). **FireAnt: A Modular Robot with Full-Body Continuous Docks**. In *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Proceedings - IEEE International Conference on Robotics and Automation*, pages 6812–6817, United States. Institute of Electrical and Electronics Engineers Inc., doi: [10.1109/ICRA.2018.8463146](https://doi.org/10.1109/ICRA.2018.8463146).
- Tan, N., Hayat, A. A., Elara, M. R., et Wood, K. L. (2020). **A Framework for Taxonomy and Evaluation of Self-Reconfigurable Robotic Systems**. *IEEE Access*, 8:13969–13986, doi: [10.1109/ACCESS.2020.2965327](https://doi.org/10.1109/ACCESS.2020.2965327).
- Thalamy, P., Piranda, B., et 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*, pages 140–148, Montreal QC, Canada, doi: [10.5555/3306127.3331685](https://doi.org/10.5555/3306127.3331685).
- Thalamy, P., Piranda, B., et Bourgeois, J. (2019b). **A survey of autonomous self-reconfiguration methods for robot-based programmable matter**. *Robotics and Autonomous Systems*, 120:103242, doi: [10.1016/j.robot.2019.07.012](https://doi.org/10.1016/j.robot.2019.07.012).
- Thalamy, P., Piranda, B., Lassabe, F., et 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)*, pages 4840–4846, doi: [10.1109/IROS40897.2019.8967775](https://doi.org/10.1109/IROS40897.2019.8967775).
- Thalamy, P., Piranda, B., Lassabe, F., et Bourgeois, J. (2020). **Deterministic scaffold assembly by self-reconfiguring micro-robotic swarms**. *Swarm and Evolutionary Computation*, 58:100722, doi: <https://doi.org/10.1016/j.swevo.2020.100722>.
- Tibbits, S., McKnelly, C., Olguin, C., Dikovsky, D., et Hirsch, S. (2014). **4D Printing and Universal Transformation**. In *34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, pages 539–548, Los Angeles.
- Toffoli, T., et Margolus, N. (1991). **Programmable matter: concepts and realization**. *Physica D: Nonlinear Phenomena*, 47(1-2):263–272.
- Tucci, T., Piranda, B., et Bourgeois, J. (2017). **Efficient Scene Encoding for Programmable Matter Self-reconfiguration Algorithms**. In *Proceedings of the Symposium on Applied Computing*, pages 256–261, doi: [10.1145/3019612.3019706](https://doi.org/10.1145/3019612.3019706).

- Tucci, T., Piranda, B., et Bourgeois, J. (2018). **A Distributed Self-Assembly Planning Algorithm for Modular Robots**. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 550–558, Stockholm, Sweden. Association for Computing Machinery (ACM).
- Varshavskaya, P., Kaelbling, L. P., et Rus, D. (2008). **Automated Design of Adaptive Controllers for Modular Robots using Reinforcement Learning**. *The International Journal of Robotics Research*, 27(3-4):505–526, doi: [10.1177/0278364907084983](https://doi.org/10.1177/0278364907084983).
- Vassilvitskii, S., Yim, M., et Suh, J. (2002). **A complete, local and parallel reconfiguration algorithm for cube style modular robots**. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 1, pages 117–122 vol.1, doi: [10.1109/ROBOT.2002.1013348](https://doi.org/10.1109/ROBOT.2002.1013348).
- Vonásek, V., Saska, M., Košnar, K., et Přeučil, L. (2013). **Global motion planning for modular robots with local motion primitives**. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2465–2470. IEEE.
- Wang, X., Zhu, Y., et Zhao, J. (2013). **A dynamic simulation and virtual evolution platform for modular self-reconfigurable robots**. In *2013 IEEE International Conference on Information and Automation (ICIA)*, pages 457–462, Yinchuan, China. IEEE, doi: [10.1109/ICInfA.2013.6720342](https://doi.org/10.1109/ICInfA.2013.6720342).
- Werfel, J., Petersen, K., et Nagpal, R. (2014). **Designing Collective Behavior in a Termite-Inspired Robot Construction Team**. *Science*, 343(6172):754–758, doi: [10.1126/science.1245842](https://doi.org/10.1126/science.1245842).
- White, P., Zykov, V., Bongard, J., et Lipson, H. (2005). **Three Dimensional Stochastic Reconfiguration of Modular Robots**. In *Robotics: Science and Systems I*. Robotics: Science and Systems Foundation, doi: [10.15607/RSS.2005.I.022](https://doi.org/10.15607/RSS.2005.I.022).
- Winkler, L., et Wörn, H. (2009). **Symbricator3D – A Distributed Simulation Environment for Modular Robots**. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Xie, M., Xiong, Y., Xiong, C., Liu, H., et Hu, Z., editors, *Intelligent Robotics and Applications*, volume 5928, pages 1266–1277. Springer Berlin Heidelberg, Berlin, Heidelberg, doi: [10.1007/978-3-642-10817-4_127](https://doi.org/10.1007/978-3-642-10817-4_127). Series Title: Lecture Notes in Computer Science.
- Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., et Yin, P. (2013). **Active Self-Assembly of Algorithmic Shapes and Patterns in Polylogarithmic Time**. *arXiv:1301.2626 [cs]*. arXiv: 1301.2626.

- Yim, M., Duff, D., et Roufas, K. (2000). **PolyBot: a modular reconfigurable robot**. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 514–20.
- Yim, M., Zhang, Y., Lamping, J., et Mao, E. (2001). **Distributed Control for 3D Metamorphosis**. *Autonomous Robots*, 10(1):41–56, doi: [10.1023/A:1026544419097](https://doi.org/10.1023/A:1026544419097).
- Yoshida, E., Murata, S., Kurokawa, H., Tomita, K., et Kokaji, S. (1998). **A distributed method for reconfiguration of a three-dimensional homogeneous structure**. *Advanced Robotics*, 13(4), doi: [10.1163/156855399X00234](https://doi.org/10.1163/156855399X00234).
- Zhu, L., et El Baz, D. (2019). **A programmable actuator for combined motion and connection and its application to modular robot**. *Mechatronics*, 58:9–19, doi: [10.1016/j.mechatronics.2019.01.002](https://doi.org/10.1016/j.mechatronics.2019.01.002).
- Zhu, Y., Bie, D., Wang, X., Zhang, Y., Jin, H., et 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, doi: <https://doi.org/10.1016/j.jpdc.2016.11.016>.
- Zykov, V., Williams, P., Lassabe, N., et Lipson, H. (2008). **Molecubes Extended: Diversifying Capabilities of Open-Source Modular Robotics**. In *IROS-2008 Self-Reconfigurable Robotics Workshop*, page 13.
- Ünsal, C., et Khosla, P. K. (2001a). **A multi-layered planner for self-reconfiguration of a uniform group of I-Cube modules**. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 598–605, doi: [10.1109/IROS.2001.973421](https://doi.org/10.1109/IROS.2001.973421).
- Ünsal, C., Kiliççöte, H., Patton, M. E., et Khosla, P. K. (2000). **Motion Planning for a Modular Self-Reconfiguring Robotic System**. In Parker, L. E., Bekey, G., et Barhen, J., editors, *Distributed Autonomous Robotic Systems 4*, pages 165–175, Tokyo. Springer Japan, doi: [10.1007/978-4-431-67919-6_16](https://doi.org/10.1007/978-4-431-67919-6_16).
- Ünsal, C., Kiliççöte, H., et Khosla, P. K. (2001b). **A Modular Self-Reconfigurable Bipartite Robotic System: Implementation and Motion Planning**. *Autonomous Robots*, 10(1):23–40, doi: [10.1023/A:1026592302259](https://doi.org/10.1023/A:1026592302259).

LIST OF FIGURES

1	Some instances of ancient, modern, and future display systems (from left to right): cave paintings; sculpture; touchscreen; holographic interface from the movie Avatar; programmable matter display system from the Claytronics (Goldstein et al., 2004) project	4
2	Examples of chain modular robots: (Left) Polybot (Yim et al., 2000) self-reconfigurable robot in various chain configurations. (Center) Tripod configuration of CONRO (Castano et al., 2000) modules. (Right) Five YaMoR modules (Moeckel et al., 2006) in a tripod configuration.	7
3	Examples of lattice modular robots: (Left) ATRON (Jorgensen et al., 2004) modular robots in different configurations. (Center) Rotating M-Block (Romanishin et al., 2013) modules. (Right) 2D Crystalline (Rus et al., 2000) modular robots with extensible arms.	8
4	Examples of hybrid modular robots: (Left) Superbot (Salemi et al., 2006) modular robot in a humanoid configuration. (Center) M-TRAN (Kamimura et al., 2002) modular robot in 4-legged configuration. (Right) SMORES (Davey et al., 2012) modular robot undergoing self-reconfiguration.	9
5	Sample self-reconfiguration of about 38,500 3D Catom modules from a cup into a plate. (a) Cup initial configuration; (b) An intermediate configuration from the self-reconfiguration process; (c) Plate goal configuration.	10
1.1	The <i>Programmable Matter Consortium</i> partners	20
1.2	Visual comparison of cups made of modular-robot-based programmable matter with cubic and spherical modules and at various resolutions in terms of the size of the modules.	21
1.3	The 3D Catom: geometry from two opposite angles, skewed coordinate system, and the two possible paths for the motion of a neighbor on its surface, using and a hexagonal actuator (R_h) or an octagonal actuator (R_o).	22

1.4	Arrangement of a 1-ball of <i>3D Catoms</i> in a Face-Centered Cubic (FCC) Lattice: (a) A single <i>3D Catom</i> at the center of the ball; (b) The four bottom neighbors of the center; (c) The four horizontal neighbors of the center; (d) The four top neighbors of the center module.	23
1.5	Five snapshots of two rotations of the orange module on a pivot: the first motion connects connector #11 of the orange module to #5 of the <i>pivot</i> and second #10 to #7 of the <i>pivot</i>	23
1.6	(Left) Two additional sample motions on octagonal and hexagonal actuators. (Right) The <i>bridging constraint</i> , in which the yellow module is unable to reach its destination because of the two blocking modules in orange. . .	24
1.7	The two main motion challenges posed by the <i>3D Catom</i> model: the re-mote blocking conundrum and the motion coordination challenge . . .	25
1.8	First <i>3D Catom</i> prototype. From left to right: 3.6mm catom shell, shell covered with electrostatic actuators, photovoltaic power conversion driver, catom-embedded M^3 Mote. (Left: Nanoscribe picture, courtesy of Gwenn Ulliac - FEMTO-ST.)	28
1.9	Several shapes of robots proposed in <i>VisibleSim</i>	33
1.10	Architecture of the simulator: <i>BaseSimulator</i> framework in <i>grey</i> ; framework instantiations for each modular robot in <i>blue</i> ; user applications for a given modular robot in <i>red</i> , here with 2 configuration files.	34
1.11	Main loop of the <i>VisibleSim</i> simulator	35
1.12	Select results from previous work using <i>VisibleSim</i> across several module types and tasks.	40
1.13	Number of motions and messages simulated during the stress test experiment.	42
2.1	The three approaches to designing self-reconfiguration methods and their characteristics.	47
2.2	Overview of common self-reconfiguration models and select hardware systems.	48
2.3	A snake-like formation of Roombot modular robots. (Courtesy of Prof Auke Jan Ijspeert, Biorobotics Laboratory, École Polytechnique Fédérale de Lausanne)	49

2.4	A sample configuration of modules from the Sliding-Cube model performing reconfiguration into a 2D A shape. (From the Smart Blocks project (Piranda et al., 2013))	53
2.5	Shape formation of a triangle (left) and of a hexagon (right) on a 2D triangular lattice with the Amoebot model. (Courtesy of Prof Andrea Richa, Self-organizing Particle Systems Lab, Arizona State University)	61
2.6	Overview of modular robotic and particle system self-reconfiguration methods.	63
2.7	Interrelationship of hardware and software programmable matter components.	74
3.1	Scaffolding structure in a Square Cubic (SC) lattice as proposed by Stoy et al. (2004)	81
3.2	Side-by-side comparison of: (a) regular cube made of 3D Catoms; (b) scaffold version of the object; (c) scaffold cube with added coating.	83
3.3	(Left) Overview of the sandbox, with the entry points used for supplying and discarding modules circled in orange. (Right) Scaffold of a cube of side 13 modules over the sandbox. Brown planes divide the object into several vertical areas. Scaffold modules from each area can be supplied exclusively through the sandbox entry points directly below them, enabling <i>pipelined</i> reconfiguration.	84
3.4	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.	87
3.5	(a) Scaffold of a cube of size $20 \times 20 \times 20$ modules, with highlighted target volume; (b) scaffold with horizontal branches extended into structural supports; (c) snapshot of the coating phase; (d) fully assembled coating of a cube, with scaffold inside.	91
4.1	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).	96
4.2	Diagram of the construction polytree of a $4 \times 4 \times 2$ -tile cube. (Left) Bottom tile layer; (Right) Top tile layer, with green arrows the edges between the bottom and above layer. Red edges highlight a possible critical path	97
4.3	Reminder of the anatomy of a scaffold tile. Entry Point Locations (EPL) in pink on the left image.	98

- 4.4 **Simplified view of the behavior of each module state and transitions between them.** 98
- 4.5 **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. . . 100
- 4.6 Diagram of the **construction polytree** of the ground layer of a shape with concavities at the ground level, requiring two seed tiles. Blue edges show dependencies from the *scaffold seed*; purple edges from the second seed tile; green edges from the merge tile once both processes are synchronized, and orange edges highlight the two edges that cause the synchronization. Either the blue process reaches the merge tile first and it has to wait for the purple process, or the other way around. 101
- 4.7 Visual comparison between: **(a)** a scaffold cube with its front-left corner aligned on an entry point, and **(b)** the same cube with its center aligned on an entry point. (This cube is actually smaller by one module in each direction to accommodate the coating.) The centering difference is most noticeable at the ground level. 104
- 4.8 **Light state transition diagram.** The two *Beam* routines are executed concurrently on pivot modules. 108
- 4.9 Construction of a 3D model of a pyramid using scaffolding ($b = 6$). **(a)** Support structure; **(b)** Scaffold of the 4-pyramid; **(c)** Envisioned coated 4-pyramid, after removal of support modules (differs from the actual implemented coating method). 110
- 4.10 Reconfiguration time relative to tile count and module count for increasing sizes of h -pyramid 118
- 4.11 *Request-based* (*sync.* in the legend) *feeding* vs. *continuous* (*asynchronous* in the legend) *feeding* variants comparison, and variable motion duration results. 119
- 4.12 Modules overuse by the *continuous flow* variant. 120

4.13 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 EPLs (in transparent pink).	122
4.14 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.	126
4.15 Overview of the shapes under study: (Left) OpenSCAD preview of the CSG model of the goal shape; (Right) Scaffold interpretation built by our algorithm. For canonical shapes, dimensions are set to $d = 6$ tiles.	127
4.16 Number of modules in canonical shapes, with varying sizes.	128
4.17 Global reconfiguration time, with varying sizes.	129
4.18 Reconfiguration speed in time steps per height level.	129
4.19 Reconfiguration speed in modules per time step.	130
4.20 Instant module placement for canonical shapes with $d = 6$	131
4.21 Instant modules placement: comparing simple and stacked cylinder. Both curves overlap until $t = 380$ time steps.	131
4.22 Sandcastle reconfiguration speed, modules in place per time step.	133
5.1 (a) Structural support producing no blocked positions; (b) support producing blocked positions and corresponding support segments; (c) support position that cannot be filled in the current implementation.	144
5.2 Convergence of the coating algorithm into the coating of the <i>Chair</i> shape over time.	149
5.3 Two very problematic shapes given the current coating strategy: (a) Mushroom-like shape where part of the coating on layer n is not connected to any layer $n - 1$ coating section; (b) Spinning-top shape with a bowl or depression concavity, which is also a case of disconnected coating.	151
5.4 Original self-reconfiguration pipeline idea (artistic impression): using a temporary scaffold (yellow modules) both for mechanically supporting the growing structure and removing concavities during construction.	161

LIST OF TABLES

1.1	Average execution time of ABC-CenterV1 on hardware Blinky Blocks and in simulations. Statistics on the execution time were computed over 25 runs for every configuration.	41
1.2	Number of robots for each grid size of stress test experiment.	42
2.1	Summary of complexity analyses and proofs provided in <i>Top-Down</i> works. Missing works did not provide any item. (p) means that completeness was only <i>partially</i> proven, for a limited class of reconfigurations.	71
3.1	Summary of motion constraints that are easier to satisfy in a scaffold setting. (See Section 1.2.1 on motion constraints.)	82
5.1	Simulation results of our coating algorithm on shapes of various sizes. . . .	148

Title: Distributed Algorithms and Advanced Modeling Approaches for Fast and Efficient Object Construction Using a Modular Self-reconfigurable Robotic System

Keywords: Programmable Matter, Distributed Algorithms, Modular Robotics, Self-reconfiguration, Multi-agent Systems

Abstract:

Humans have always been on a quest to master their environment. But with the arrival of our digital age, an emerging technology now stands as the ultimate tool for that purpose: Programmable Matter. While any form of matter that can be programmed to autonomously react to a stimulus would fit that label, its most promising substrate resides in modular robotic systems. Such robotic systems are composed of interconnected, autonomous, and computationally simple modules that must coordinate through their motions and communications to achieve a complex common goal.

Such programmable matter technology could be used to realize tangible and interactive 3D display systems that could revolutionize the ways in which we interact with the virtual world. Large-scale modular robotic systems with up to hundreds of thousands of modules can be used to form tangible shapes that can be rearranged at will. From an algorithmic point of view, however, this self-reconfiguration process is a formidable challenge due to the kinematic,

communication, control, and time constraints imposed on the modules during this process.

We argue in this thesis that there exist ways to accelerate the self-reconfiguration of programmable matter systems, and that a new class of reconfiguration methods with increased speed and specifically tailored to tangible display systems must emerge. We contend that such methods can be achieved by proposing a novel way of representing programmable matter objects, and by using a dedicated reconfiguration platform supporting self-reconfiguration.

Therefore, we propose a framework to apply this novel approach on quasi-spherical modules arranged in a face-centered cubic lattice, and present algorithms to implement self-reconfiguration in this context. We analyze these algorithms and evaluate them on classes of shapes with increasing complexity, to show that our method enables previously unattainable reconfiguration times.

Titre : Algorithmes distribués et méthodes de modélisation avancées pour une construction rapide et efficace d'objets avec un robot modulaire auto-reconfigurable

Mots-clés : Matière Programmable, Algorithmes Distribués, Robotique Modulaire, Autoreconfiguration, Systèmes Multi-agents

Résumé :

Les humains ont de tout temps cherché à contrôler leur environnement. Mais avec l'arrivée de l'ère numérique, une technologie émergente promet de devenir l'outil ultime de cette quête : la matière programmable. Bien que toute forme de matière pouvant être programmée pour réagir de façon autonome à un stimulus puisse prétendre à cette dénomination, son substrat le plus prometteur réside dans les systèmes robotiques modulaires. Ces systèmes robotiques sont composés de modules interconnectés, autonomes, et aux ressources limitées, devant se coordonner par leurs communications et leurs mouvements afin d'accomplir des tâches complexes.

La matière programmable pourrait être utilisée pour réaliser les systèmes de représentation de demain: des affichages tangibles et interactifs en 3D, qui promettent de révolutionner la façon dont nous interagissons avec le monde virtuel. Des ensembles de robots modulaires composés de plusieurs milliers de modules peuvent s'organiser pour former des objets tangibles capables de se transformer à l'infini sur demande. D'un point de vue algorithmique, cependant, ce processus d'autoreconfiguration représente un défi considérable à

cause des contraintes cinématiques, temporelles, de contrôle, et de communication, auxquelles sont soumis les modules.

Nous défendons dans cette thèse qu'il existe des moyens d'accélérer la reconfiguration des systèmes de matière programmable, et qu'une nouvelle classe de méthodes de reconfiguration plus rapide et mieux adaptée aux systèmes de représentation tangibles doit voir le jour. Nous soutenons qu'il est possible de parvenir à de telles méthodes en proposant une nouvelle façon de représenter les objets faits de matière programmable, et en utilisant une plateforme d'assistance dédiée à l'autoreconfiguration.

Par conséquent, nous proposons un cadre pour réaliser cette approche innovante sur des ensembles de modules quasi-sphériques arrangés en structures cristallines cubiques à faces centrées, et présentons des algorithmes permettant d'implémenter l'autoreconfiguration dans ce contexte. Nous analysons ces algorithmes et les évaluons sur des cas de construction de formes de complexité croissante, afin de montrer que notre méthode permet d'arriver à des durées de reconfiguration jusqu'ici inatteignables.