Shape Formation by Self-Disassembly in Programmable Matter Systems

by

Kyle William Gilpin

B.S., Massachusetts Institute of Technology (2006)
M.Eng., Massachusetts Institute of Technology (2006)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2012

Certified by .....
Daniela Rus
Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Shape Formation by Self-Disassembly in Programmable Matter Systems

by

Kyle William Gilpin

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Programmable matter systems are composed of small, intelligent modules able to form a variety of macroscale objects with specific material properties in response to external commands or stimuli. While many programmable matter systems have been proposed in fiction, (Barbapapa, Changelings from Star Trek, the Terminator, and Transformers), and academia, a lack of suitable hardware and accompanying algorithms prevents their full realization. With this thesis research, we aim to create a system of miniature modules that can form arbitrary structures on demand.

We develop autonomous 12mm cubic modules capable of bonding to, and communicating with, four of their immediate neighbors. These modules are among the smallest autonomous modular robots capable of sensing, communication, computation, and actuation. The modules employ unique electropermanent magnet connectors. The four connectors in each module enable the modules to communicate and share power with their nearest neighbors. These solid-state connectors are strong enough for a single inter-module connection to support the weight of 80 other modules. The connectors only consume power when switching on or off; they have no static power consumption.

We implement a number of low-level communication and control algorithms which manage information transfer between neighboring modules. These algorithms ensure that messages are delivered reliably despite challenging conditions. They monitor the state of all communication links and are able to reroute messages around broken communication links to ensure that they reach their intended destinations.

In order to accomplish our long-standing goal of programmatic shape formation, we also develop a suite of provably-correct distributed algorithms that allow complex shape formation. The distributed duplication algorithm that we present allows the system to duplicate any passive object that is submerged in a collection of programmable matter modules. The algorithm runs on the processors inside the modules and requires no external intervention. It requires $O(1)$ storage and $O(n)$ inter-module messages per module, where n is the number of modules in the system. The algorithm can both magnify and produce multiple copies of the submerged object.

A programmable matter system is a large network of autonomous processors, so these algorithms have applicability in a variety of routing, sensor network, and distributed computing appli-

cations. While our hardware system provides a 50-module test-bed for the algorithms, we show, by using a unique simulator, that the algorithms are capable of operating in much larger environments. Finally, we perform hundreds of experiments using both the simulator and hardware to show how the algorithms and hardware operate in practice.

Thesis Supervisor: Daniela Rus
Title: Professor

Acknowledgments

Many people deserve my gratitude and thanks for helping to make this thesis a reality. First, my advisor, Daniela Rus, was instrumental in the process. I have been incredibly fortunate to find such an amazing advisor who's always been a staunch ally and a supportive mentor. More than anything, Daniela has taught me to dream big.

My committee members, Rob Wood and Anantha Chandrakasan, also deserve a great deal of thanks. Not only have Rob and Anantha provided useful feedback as I have developed my thesis, they have been long-term collaborators and mentors as well. Under Anantha's guidance, I had the chance to develop several high-performance FPGA systems that were essential in securing my first job outside academia. Rob has been an ideal collaborator as part of the DARPA Programmable Matter project. From day one, he has been incredibly generous with his time and equipment, no questions asked.

Ara Knaian developed the electropermanent magnets that are essential to the Smart Pebble modules. Ara was always full of energy and wild ideas. Fueled by Ara's enthusiasm alone, we painstakingly built more than 250 electropermanent magnets by hand. Kent Koyanagi helped with the Robot Pebbles project over the course of two summers. Kent's dedication and work ethic were amazing. Despite some awfully boring tasks, he never complained, and he often was in lab for longer hours than I was.

I also owe my thanks to my fellow graduate students in the Distributed Robotics Laboratory. Few groups function so well together with so little conflict or competition. In particular, there are several alumni who were instrumental to my quick integration and success with the group: Keith Kotay, Marty Vona, Carrick Detweiler, and Iuliu Vasilescu. Outside of MIT, I have the BMG to thank for reminding me not to take life too seriously. Scott, Eric, Sangeen, Brad, KC, Mahmoud, Ihsanul, Matt, Joe, and Paul, thanks for all the good times.

Finally, I have my parents, Bill and Linda, sister, Amy, and wife, Erin to thank for their support, encouragement, and understanding. My parents fostered my love of all things electronic and mechanical from an early age with many trips to yard sales for old radios and record players to take apart in the basement. As I grew older, there was never a shortage of Capsella modules, Radio

Shack electronic project kits, Legos, and, most important, encouragement to explore. Amy, you were always a good, if not willing, test subject for my contraptions. More importantly, you have always taken interest in my life and, by doing so, encouraged me to aim high. Erin, your generosity, unwavering support, perpetual excitement, and reminders that there are more important things in life than working all the time have been essential. I could not ask for a better partner, and I could not have done this without you–thank you.

This work was supported by DARPA and the US Army Research Office under grant number W911NF-08-1-0228, NSF EFRI grant number 0735953, Intel, and the NDSEG fellowship program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the sixty-five plus years since the advent of the first computer program, the concept of programming software has been refined, popularized, and become ubiquitous. We propose moving beyond programming software to programming the properties, such as shape, stiffness, texture, and conductivity of physical matter. Since their advent, computers have changed from room-filling mainframes, to bulky desktops, to portable laptops, to lightweight tablets, to smart phones. We see *programmable matter* as the next step in this progression. Our long-term goal is to create a programmable matter system that is able to programmatically modify its physical properties. The ability to form objects with specific form and material properties would be the ultimate universal toolkit. Such a system would be immensely useful for an astronaut on an inter-planetary mission or a scientist at an isolated research station. For mechanics and surgeons, the ability to form highly customized, task-specific tools would be immensely valuable.

Our particular approach relies on *modular* programmable matter. Modular programmable matter systems modify their bulk material properties by changing the characteristics of, or relationships between, the small unit modules that, as a group, compose the larger programmable matter system. We envision a bag containing millions of these tiny, intelligent particles that the user shakes in order to form some goal object. As they are agitated, the modules contained within the bag selectively bond with their neighbors in order to form the user's goal shape. After the bonding process is complete, the user can retrieve the object from the bag and dust off the extra modules. When

the user is done using the object, he or she returns it to the bag where it disintegrates back to its component modules for reuse.

The goal shape may be a robot built for a specific task, (a snake to pass through a tunnel or a wheeled rover to quickly cover open ground), or an object designed for a particular job, (such as a wrench or surgical instrument). In all cases, the resulting structure, due to the intelligent nature of its component parts, is imbued with unique properties. An object formed from programmable matter could provide real-time, in-situ feedback on internal stresses; disintegrate at a controlled rate; or dynamically change its rigidity to correspond with the task at hand.

The *Smart Pebbles* (shown in Figure 1-1) and associated algorithms that we propose in this thesis are a first step towards what we believe will be the long legacy of programmable matter.

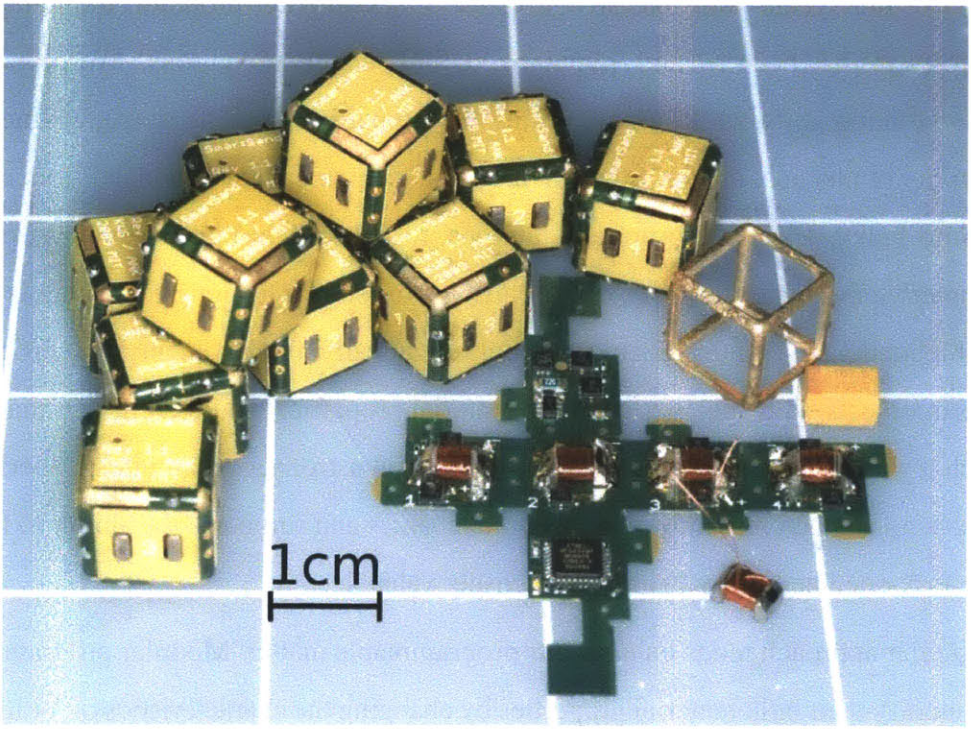


Figure 1-1: The Smart Pebbles form a modular programmable matter system that is capable of creating 2D objects by selectively forming and severing bonds between the constituent modules. Each module can communicate and bond with its nearest neighbors and has a small processor inside.

1.1 Challenges

In the quest to create a universal modular programmable matter system, there are challenges which must be overcome. Each unit module must contain computation, sensing, connectors, and a power source in as small of package as possible. The fabrication of these modules should be streamlined so that it becomes practical to create systems with millions of modules. The algorithms controlling the modules must be highly efficient and distributed so that they can run on the modules themselves. In the design process, there is inherent competition between hardware complexity, computational demands, and system capabilities. Sophisticated hardware may ease computational demands and enable additional system functionality, but it comes at the price of additional manufacturing complexity and cost. There are many competing requirements that must be taken into account during the design process.

With regard to hardware, the modules must be electrically and mechanically active in order to be called programmable matter, and not just a large distributed computer. The modules should not rely on a complicated external apparatus to function. The system should not be constrained to a lab environment in which the modules' computation and actuation capabilities are offloaded to auxiliary machinery. Randomly shaking a bag of modules is as much outside intervention as is reasonable while maintaining the system's autonomy.

Fabricating millimeter-scale three dimensional structures that are electrically and mechanically active is difficult. While micro- and nano-fabrication processes exist to fabricate 2D and 2.5D electromechanical structures, there are currently few ways to fabricate 3D structures at the same scale. What options do exist are difficult to employ in mass production. While 2D systems will be part of the natural progression to 3D structures, practical programmable matter systems must be capable of creating 3D objects. Algorithms developed for 2D hardware must be extensible to three dimensions.

Each module in a programmable matter system must be capable of programmatically bonding with its neighbors. Small, easily switchable, high-strength connectors do not exist. For a programmable matter system to form objects that are more than purely decorative, the inter-module bonds must be strong. Additionally, the connection mechanisms must have quick response times,

consume minimal power, and be reversible. The connectors, processor, and communication system of each module must also be powered. Finding a way to supply power to millions of tightly packed, millimeter-scale modules is not trivial. With current technology, it is infeasible to embed a battery in each module. Even with batteries, recharging such a large collection of modules would be problematic. Any practical system must find some other approach to supplying the modules with power.

There are also algorithmic challenges. The programmable matter modules must be able to self-assemble. As individual modules continue to shrink, and the number of modules in a system grows into the millions, the modules must be capable of self organizing so that they can autonomously bond and communicate with their neighbors. Once assembled, the modules must have a reliable communication scheme that can quickly recover from any errors. Any system with millions of modules is certain to contain many defective units that will threaten the system's high-level functionality. A practical system must be designed with inherent robustness in every component.

Conveying the user's desired object to the programmable matter system is challenging. The user interface must be sophisticated enough to provide an abstraction barrier between the high-level object description and the behavior of each individual module in the system. Furthermore, due to power, storage, and processing constraints, a programmable matter system must have an efficient way of distributing information about the object that the user wishes to form. Individually informing each module in a million module system of its role is impractical. The system must employ a better approach whose worst case space and time requirements scale favorably.

Finally, the development of vast modular programmable matter systems requires advanced debugging tools. Current tools struggle to support massively parallel systems. The challenges stem from several factors. The chief difficulty is that the system's functionality is governed by the interactions of many individual modules, not by each module in isolation. Debugging one module may not be difficult, but debugging its interactions with the other thousand or million modules is a formidable challenge. Aggregating just a portion of the system's state so that it is displayed in a useful manner is a significant undertaking. Even addressing a particular module in order to query its particular state is challenging in a system with millions of modules.

1.2 Current State of the Art

Modular programmable matter systems are an outgrowth of the modular robotics field [133, 36]. Much effort has been devoted to designing novel hardware, but comparatively little effort has been made to develop advanced algorithms for million-module programmable matter systems. Existing programmable matter hardware spans many scales ranging from micrometers [85, 111, 24] to many centimeters and everywhere in between. Existing hardware platforms use modules that are of many different shapes including triangles [8], squares [121, 38], cubes [34, 113], circles [93], and cylinders [37, 47]. The majority of systems operate in two dimensions, but there are also many three dimensional systems. Typically, the modules used in three dimensional systems are larger due to their additional mechanical complexity.

Modular systems with smaller unit modules typically transfer computation or actuation abilities away from the modules. Many of the smallest modules are completely passive and rely on external fixtures that manipulate the modules with electromagnetic [3, 78] or fluidic forces [122, 112], for example. Some even employ micro-robots to aid the assembly process [84]. It will be difficult to move these types system out of the lab and into the everyday environments unless their support equipment follows. Even larger systems that have on-board computation and actuation abilities often rely on external controllers to instruct the modules. With the easy availability of lithium-ion batteries, newer systems tend to be battery powered, but many systems are often tethered to external voltage supplies.

Inter-module connection mechanisms are an active area of research. Traditionally, modular robotic systems have relied on active mechanical latching for its rigidity and strength [19, 63]. As modules sizes have continued to shrink, designers have looked to alternative connection mechanisms including both permanent magnets [134] and electromagnets [37, 50]. The crucial drawback to electromagnets is that they consume a large amount of power making it difficult to untether the modules from an external power source. One alternative to electromagnets is to use mechanically switchable permanent magnets [35]. Some modules also use semi-permanent passive mechanical bonding mechanisms [123], but this requires external actuation to control which connectors are latched. Other researchers have attempted to use electrostatic forces [47], but electrostatic connec-

tors are not practical at large scales where they exert negligible forces. As modules continue to shrink, electrostatic connectors will become more attractive both for their potential holding forces and easy of fabrication. Researchers are also developing promising connectors based on van der Waals forces [79, 57].

The algorithms for controlling modular robots and programmable matter systems are more difficult to characterize although there are some common themes. Most research assumes that the modules in a system are arranged in regular lattices or chains. A few algorithms exist that assume the modules may be arranged randomly [95, 29], but few hardware examples of such systems exist [102]. As modules continue to shrink and precise inter-module alignment becomes more difficult, such free form system will become the norm.

There are many algorithms devoted to locomotion or reconfiguration in modular systems [27, 118, 115]. These algorithms can only be executed on systems whose modules support relative locomotion. Some algorithms have focused on self-repair [134, 94]. Self-assembly is another common goal, and there have been many theoretical proposals [1, 2, 92, 48] and hardware implementations [78, 73]. Often, the goal shape is encoded with a set of generic rules shared by all modules in the system. Other approaches to programming shape attempt to convey the desired shape's description to those modules on the exterior of the ensemble [37] or only those modules that are part of the desired shape [34]. Finally, duplication is another approach to shape formation [38, 64]. One particular proposal uses a centralized algorithm running as part of a modular programmable matter system to duplicate arbitrary three dimensional objects [88].

1.3 Our Approach

Our approach improves on existing hardware and algorithms in several ways. It is best understood in the context of traditional fabrication methods, which rely sequential processes performed on heterogeneous materials. For example, to fabricate a digital torque wrench with traditional methods, one machines the head of the wrench, casts the body, molds the rubber handle, fabricates the electronics, and finally assembles all of the components. Once built, the wrench is difficult to repurpose for other tasks. In contrast, our approach to fabrication with modular programmable

matter is a cyclic process that utilizes homogeneous material. Each tiny cubic module has on-board computation, nearest-neighbor communication, and latching capabilities. Using their on-board intelligence, in collaboration with their neighbors, the modules can selectively bond together to form specific shapes.

To fabricate the same torque wrench with our modular programmable matter system, one begins with a model of the wrench. This model could be miniaturized, and it only needs be a rough approximation of a wrench made out of plastic, Styrofoam, clay, or even paper. The user surrounds the model wrench with programmable matter modules by burying the model in a bag containing thousands or even millions of individual modules. Working in collaboration, the modules sense the shape of the model and set about making a duplicate. The duplicate is formed when the extra modules in the bag, those in the vicinity of, but not in direct contact with, the original model selectively bond with their neighbors. If the original model is a miniature, the duplicate can be magnified. The number of copies is only limited by the amount of programmable matter material in the bag. Once the modules that form the duplicate wrench have finished mechanically bonding with their neighbors, the user can remove both the original and duplicate wrenches from the bag.

The new torque wrench is imbued with, not only the desired form, but the desired sensing and computation capabilities. The modules retain their sensing, computation, and bonding abilities once removed from the bag. If the user needs to refine the wrench's shape, he can selectively remove additional modules. Taken to the extreme, when the user is done with the wrench, he drops it back into the bag where it disintegrates, and the particles can be reused.

Our approach to realizing programmable matter is unique in several specific ways. First, we have developed a novel modular programmable matter hardware platform, the Smart Pebbles. Each Smart Pebble is a 12mm cube, (see Figure 1-1), capable of mechanically bonding to and communicating with four planar neighbors. The modules employ unique electropermanent magnets that serve as mechanical bonding, communication, and power transfer interfaces. Each module contains an 8-bit microprocessor that controls its four electropermanent magnets. Using a set of external connections, we can provide power to a single module from which power is distributed to all other modules in the system using the electropermanent magnet connectors. When activated, each con-

nector is strong enough to support the combined weight of eighty other modules. When placed on an inclined vibration table, (the two-dimensional equivalent of a shaking bag), the Smart Pebbles self-assemble. We have also developed and deployed algorithms that allow the modules to self-disassemble in an organized fashion. As a result of these abilities, the Smart Pebbles serve as an excellent platform on which to implement the shape formation algorithms that we have developed.

The two unique aspects our of approach to high-level, arbitrary shape-formation functionality are captured by Figure 1-2. First, we propose a two-step shape formation process in which the modules in the system first coalesce and self-assemble into a completely uniform solid block of material that encases the original object that is being duplicated. Once this initial block of material is complete, then the system utilizes self-disassembly to remove modules that are not a part of the manufactured duplicate.

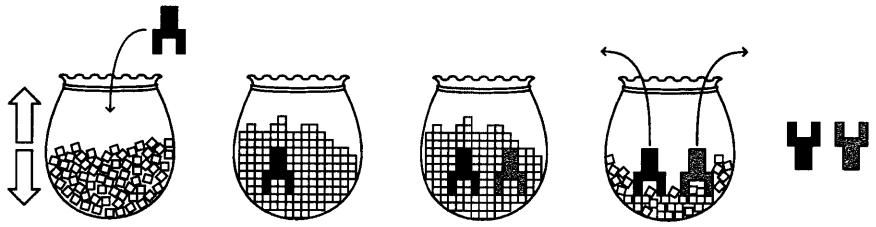


Figure 1-2: To form shapes using the Smart Pebbles system, a passive object to be duplicated (shown in black) is submerged in a large collection or programmable matter particles. The particles solidify in a regular lattice to encase the passive shape. Once solidified, modules bordering on the passive shape sense its topology and inform other modules in the vicinity that they will become part of the duplicate shape. Once these duplicate modules (shown in grey) have been notified, all other modules break their mechanical bonds leaving just the original and duplicate shapes behind. When the user is done with the duplicate, he may drop it back into the bag where it will disintegrate, and the modules can be reused.

Second, we introduce new user interface for programmable matter systems in which a scaled-down physical replica of the desired object is all that is needed to describe the goal shape. The user does not need a complicated CAD model or any other technical description. Given a bag of smart particles, we envision dipping a replica of the object we wish to create into the bag. The modules surrounding the object sense and learn its shape. Then, using programmed communication and connections, they replicate the object at the desired scale using the spare modules in the bag. Once

the solid replica is created, all other inter-module connections are broken, and the user can retrieve the duplicate object from the bag. This approach eliminates the need for external computation and actuation along with the associated external communication links. It also reduces the internal neighbor-to-neighbor communication burden on the modules themselves.

1.3.1 Self-Assembly and Disassembly

Traditional self-assembling systems aim to form complex shapes in a direct manner. As these structures grow from a single module, new modules are only allowed to attach to the structure in specific locations. By carefully controlling these locations and waiting for a sufficiently long period of time, the desired structure grows in an organic manner. In contrast, our system greatly simplifies the assembly process by initially aiming to form a regular crystalline block of fully connected modules. We only limited attempts to restrict which modules or faces are allowed to bond with the growing structure. After we form this initial block of material, we complete the shape formation process through self-disassembly and subtraction of the unwanted modules.

Our approach has a distinct advantage over techniques based solely on self-assembly. Self-disassembly does not rely on complicated attachment mechanisms that require precise alignment or careful planning. Self-disassembly excels at shape formation because it is relatively easy, quick and robust. Our system does not need to carefully select exactly where new modules are allowed to bond to the growing structure. In the process, it also avoids the need to consider the structural integrity of the growing ensemble. By first forming a regular lattice, the system serves as its own support scaffolding. As an example of why our two-step process is advantageous, consider how much easier it is to carve an arch out of solid marble than it is to construct an arch from loose stones.

As the individual modules in self-reconfiguring and programmable matter systems continue to shrink, it will become increasingly difficult to actuate and precisely control the assembly process. In particular, designing modules capable of exerting the forces necessary to attract their neighbors from significant distances will be challenging. Instead, these systems may find assembly and disassembly much simpler when driven by stochastic environmental forces. The Smart Pebble

modules, which are able to latch together from distances approximately 20–35% of the module dimensions, could easily take advantage of these stochastic assembly mechanisms to form an initial structure. Our particular system also relies on external forces to carry the unused modules away from the final shape. (The connectors that we employ cannot both attract and repel.) In our system, this force is often gravity, but it could also be vibration, fluid flow, or the user reaching into the bag of particles to extract the finished object.

1.3.2 Distributed Duplication

A big challenge associated with fabricating composite objects from a large collection of intelligent particles is conveying the desired shape of the object to be produced to the ensemble of modules. One approach is to manually inform each module whether or not it should bond with its neighbors to become part of the goal shape. Alternatively, we can use a graphical interface to define the shape and automatically generate a list of messages to inform each module whether it is part of the structure. Both of these strategies require a reliable communication link with the modules and a large volume of information to be exchanged. Using the naive approach, one message must be sent for each module included in the final shape. Assuming that each module is roughly one cubic millimeter, it would require over 200,000 modules, and hence 200,000 messages, to form an object the size of a baseball. More advanced approaches reduce the communication burden by relying on shape abstraction to convey the desired form to the initial collection of modules. Even so, the user must know how to best fit the desired shape into the collection of modules. That requires the user know detailed information about the initial arrangement, information that takes time and energy to collect and transmit to the user. Furthermore, even if the user can blindly guess how to orient the desired shape in the initial block of material, the system must still have a robust external communication interface in order receive the shape description from the user.

As an alternative, we propose shape formation through duplication. In our system, the user surrounds a passive original object with programmable matter modules. Without anything more than a single start command, the system employs a set of robust, distributed algorithms to sense the shape of the passive object and inform the appropriate modules within the collection that they

are to become part of a duplicate object. This approach eliminates the communication link from the user to the system and all of the communication overhead associated with it. It also removes the need for the graphical interface to the system. Because the algorithm is distributed and does not rely on a centralized, external controller, the distributed algorithm provides a scalable solution. No module ever stores the complete goal shape nor the global state of the system; the memory required by each module is $O(1)$. Furthermore, the number of inter-module messages exchanged is $O(n)$ per module, where n is the number of modules in the system.

1.4 Thesis Contributions

In this document, we aim to examine the following thesis:

Digital fabrication can be accomplished with smart particles capable of self-disassembly.

To do so, we focus on programmable matter systems consisting of collections of physically connected robotic modules that have the ability to communicate with and bond to their immediate neighbors. Using this functionality, we show that a system of these modules is capable of autonomously creating user-specified goal shapes through a process of self-assembly followed by self-disassembly. This thesis makes a number of contributions to the programmable matter and modular robotics communities. First, it presents novel hardware that is among the smallest of autonomous modular robot systems. Second, it develops new distributed algorithms that are applicable to both programmable matter systems and broader networking challenges. Finally, it illustrates the use of a unique simulator that allows for the same code to be used in hardware and simulation. These main contributions can be divided into a number of tasks that are illustrated in Figure 1-3.

The majority of the tasks illustrated in Figure 1-3 are centered around algorithms that allow several different approaches to shape formation in modular robotic systems:

- the user can *virtually sculpt* the goal shape using a simple GUI that transmits a description

Figure 1-3: This thesis can be divided into three major components: hardware, simulator, and shape formation algorithms, each of which is explained in the remainder of this document.

of the goal shape back to the initial block of material using a minimal number of inclusion messages;

- a miniature description of an object can be expanded into a much larger object through *magnification*;
- an original description of an object can be *replicated* to form an arbitrary number of copies;
- *sensing* allows the replication of a passive object engulfed by a collection of modules;
- any number of the above approaches can be combined to form arbitrarily complex shapes from an initial block of material.

To complement the algorithms and demonstrate that they operate correctly in practice, we have developed the Smart Pebbles. The Pebbles are one cubic centimeter autonomous modules capable of shape formation through self-assembly and self-disassembly. They have a number of key features:

- The 12mm cubes are formed by wrapping a flexible circuit around a brass frame;
- Four genderless connectors enable the formation of 2D structures;
- The connectors are based on solid-state electropermanent magnets which are strong enough to support the weight of 80 modules yet consume no power except when changing state;
- The connectors are used for latching, communication, and power transfer (the modules do not contain batteries).

In presenting the hardware, simulator, and algorithms covered by this thesis we have explored theory, implementation, and experiments. Where possible, the experiments have been performed using physical hardware and supplemented with the results of realistic simulations. In summary, this thesis makes the following contributions:

- the concept of, and the algorithms to implement, shape formation through the process of self-assembly followed by organized self-disassembly;

- a shape formation algorithm that is capable of duplicating, magnifying, and creating multiple copies of an original shape submerged into a collection of smart particles;

- a system employing unique electropermanent magnets and related hardware to achieve mechanical bonding, communication, and power transfer from a single component;

- an easily reconfigurable high-level computing platform composed of 50 12mm-cubic nodes capable of communicating with their nearest neighbors to develop, debug, and deploy distributed algorithms;

- a unique simulator that allows a single code base to be used in software-only simulation, hardware in the loop simulation, and hardware-only experiments.

1.5 Thesis Outline

The remainder of this document is organized as follows. In Chapter 2, we present existing research related to this thesis. Specifically, we address how programmable matter evolved from the larger modular robotics field. In doing so, we highlight key hardware and software systems and present more details on the current state of the art. In Chapter 3 we present the Smart Pebbles hardware platform. We detail both the construction of the Pebbles as well as their physical and electrical attributes. In particular, we present our miniaturized electropermanent magnet connectors which allow the Pebbles to mechanically bond, communicate, and share power. Chapter 4 introduces the distributed simulator, called Sandbox, which we created in order to test the application-level algorithms that drive the Smart Pebbles to form specific shapes. The benefit of the Sandbox simulator is that it emulates both the low-level hardware and communication software so that the exact same high-level source code can be easily re-targeted for either the hardware Pebbles or the simulator. Chapter 5 extends the discussion of inter-module communication started in Chapters 3 and 4. It explains the low-level algorithms used to exchange messages between neighboring modules. The chapter also presents results that characterize the performance of the communication infrastructure.

Having described both the hardware and software platforms, Chapter 6 presents the basic shape

formation algorithm; it demonstrates that the Smart Pebbles are capable of self-assembling. Once the modules have formed an initial block of material, the chapter also shows how the user can virtually sculpt multiple shapes out of the initial block of material using self-disassembly. Results of both the self-assembly and sculpting processes are included in dedicated sections of the chapter.

Chapter 7 presents our distributed shape duplication algorithm. The algorithm is capable of duplicating a passive shape that is engulfed by a collection of modules. We present the algorithm, analyze its running time, and verify that it operates correctly with experiments in both simulation and hardware. Chapter 8 shows how we have extended the duplication algorithm to three dimensions. The chapter includes simulated experiments with over 1000 modules. Finally, Chapter 9 discusses what we have learned while developing our system, and it also presents suggestions for how to proceed in making programmable matter as ubiquitous as personal computers.

Chapter 2

Related Work

Programmable matter defines any system which is able to modify its physical properties in a programmatic way. Such systems can modify their elasticity, coefficient of friction, conductivity, or geometry. To date, most programmable matter systems have been designed as collections of particles, or modules, whose local inter-module interactions govern the macro-scale behavior of the system. While these *modular* programmable matter systems are popular, they do not fully define the field.

The term programmable matter rose to popularity around 2002 within the robotics community. MacLennan [69] described, in high-level terms, using algorithms to dictate the behavior of a material at a molecular level. He proposed "universally programmable intelligent matter" that could be programmed to fill any number of material needs. Nagpal captured the vision of programmable matter when she wrote, "imagine a flexible substrate, consisting of millions of tiny interwoven programmable fibers, that can be programmed to assume different global shapes" [77]. The distinction between programmable matter and other robotic systems can be arbitrary. Many robotic systems developed before "programmable matter" was popularized still meet all requirements to be called programmable matter.

Due to programmable matter's origins within the robotics community, there are two themes which permeate most programmable matter systems: 1) they are based on modular robotic systems; and 2) the systems are primarily concerned with programming shape. In particular, most

programmable matter systems are composed of robotic modules that can communicate with, and selectively bond to, their neighbors. These system change their shape by either rearranging, adding, or subtracting constituent modules.

Our research builds on previous work in self-reconfiguring robotics and robotic self-assembly, both of which grew out of the modular robotics field which began with a paper presented at International Conference on Robotics and Automation in the Spring of 1988 by Toshio Fukuda et al. [28] titled "Dynamically Reconfigurable Robotic System." In their paper, Fukuda et al. describe the abstract concept of a reconfigurable robotic system that can assume different shapes. In that paper, Fukuda and Nakagawa envisioned a robot system composed of different types of modules that can combine to accomplish a variety of tasks. Over the past twenty years, modular robotics research developed many facets: hardware design; planning and control algorithms; the trade-off between hardware and algorithmic complexity; efficient simulation; and system integration.

2.1 Modular Robotics

Modular robots are collections of physically connected, electromechanically active modules that, as a whole, form robotic systems that exhibit capabilities greater than those of the individual modules. Typically modular robots can change their shape or configuration in order to adapt to a variety of different tasks. For example, a collection of modules could reconfigure from a closed chain that rolls quickly over open ground to a legged robot that more easily traverses rough terrain. Modular robots are typically touted for their adaptability, their fault-tolerance, and the relative simplicity of the unit modules. Modular robotic systems can be described and classified on several axes using a variety of properties. In what follows, we choose the traditional route of classifying modular robotic systems by the geometry of the system: chain, lattice, truss, or free form. For a more detailed history of the modular robotics field, consult [133, 36].

2.1.1 Chain Systems

The defining characteristic of chain-type modular robot systems is the fact that the modules, when connected to their neighbors, are arranged in a chain. These chains may be one-dimensional, or two-dimensional, but three-dimensional chains are not as common. The fact that a chain-type modular robot is two-dimensional, or even one-dimensional, does not mean that it cannot operate in three dimensions. In fact, snake-like modular robots composed of segments with orthogonal joints are quite common.

One of the first chain-type modular robotic systems was the Polypod system developed by Yim [129, 130]. The Polypod system was composed of two types of modules: segments and nodes. It could form a variety of shapes including rolling loops and hexapods, and it went on to inspire many other chain-based systems. One was the CONRO system [11, 100, 12] in which each module was composed of two orthogonal servo motors controlling each module's pitch and yaw.

Murata et al. developed the M-TRAN modular robotic system [76, 46, 61, 74] which has undergone multiple revisions and improvements. In [46], Kamimura et al. employ a set of interconnected, out of phase oscillators to achieve walking gaits in the M-TRAN system. Marbach and Ijspeert improved upon the ability of systems like M-TRAN to generate gaits in real-time by applying function optimization to their modular system, YaMoR [70]. Murata et al. added cameras to the M-TRAN system so that a set of M-TRAN modules could separate, perform independent tasks, and then rejoin into a larger structure [74].

The ATRON system [81, 45] was developed to improve upon the M-TRAN. Lund et al. wanted to keep M-TRAN's ability to form dense lattices while taking advantage of the two orthogonal degrees of freedom, (pitch and yaw), found in the CONRO system. The Superbot system [98] also builds upon on the mechanical design of M-TRAN by adding an additional degree of rotational freedom between the two existing rotation axes.

The PolyBot is chain-type modular robot [131, 135] with a single rotational degree of freedom. PolyBot evolved into CKBot which has demonstrated the ability to reassemble itself after being accidentally or intentionally destroyed [134]. The Molecube system [139], developed by Lipson et al., is another example of a chain-type modular system with only one degree of freedom but still

able to achieve interesting 3D configurations. Lipson et al. showed that a short chain of Molecube modules, along with some free modules, can self-replicate.

Yim et al. designed another unique chain-type system named RATChET [123] which uses a connected chain of inter-latching right angle tetrahedrons to form structures. Neighboring RATChET modules latch together when the angle between them passes some critical value, and they unlatch through the use of shape memory alloy (SMA) springs when heated beyond 70 degrees Celsius. Interestingly, the RATChET modules possess no intelligence. Instead, they rely on an intelligent external actuator which rotates to control one end of the dangling chain. One unique property of the RATChET system is its relatively strength.

2.1.2 Lattice Systems

Lattice-type modular robot systems are collections of interconnected robotic modules in which the units are situated at the intersection points of a two or three dimensional grid. (A 1D lattice system is simply a chain-type robot.) The main characteristic separating a lattice system from a densely configured chain-type robot is the density of the interconnections between the modules. In a lattice-type system, each module is typically connected to all of its neighbors. In a dense chain-type system, two modules may be neighbors, but they will not be physically connected.

Additionally, lattice-type systems tend to be built with modules that contain no rotational degrees of freedom. While the modules in a lattice system typically have mechanisms which enable the modules to move relative to, and bond with, their neighbors, they generally cannot bend themselves. In comparison, chain-type systems are often built from modules that contain one or more rotational degrees of free so that the modules can flex like links in a chain. There is some overlap between between the two types of system.

Chirikjian et al. developed one of the first lattice-based modular robotic systems [17, 16, 83] in which the modules are deformable hexagons capable of bonding with their neighbors. Others, such as Walter, Tsai, and Amato [118] further analyzed these hexagonal type systems to create distributed motion planners capable of reconfiguring the system from one state to another.

Murata et al. were also early contributors to the development of lattice-based modular robotic

systems with their development of a roughly hexagonal module capable of rolling around its neighbors in two dimensions [75, 137]. Kurokawa et al. presented a three dimensional adaptation [60] composed of cubes with six protruding arms capable of rotation. Yoshida et al. improved on this system with a new design that used shape-memory alloy actuators to rotate one robot module around the perimeter of a neighbor [136].

One of the simplest lattice systems is the the Digital Clay project [132] project. The system was a set of completely passive modules that relied on the user to make changes to it topology. The 2.5cm rhombic dodecahedrons were able to sense and communicate with their neighbors in order to create a virtual model of the physical arrangement of modules.

Rus et al. also explored the idea of 3D modules capable reconfiguration through a series of latchings, rotations, and unlatchings with the Molecule system [54, 55, 56]. In [96, 97], Vona and Rus describe a different type of deformable lattice system. The Crystal system is composed of square modules able to expand and contract by a factor of two in the x-y plane. Suh et al. expanded on the Crystal concept with the Telecubes [105] that could move in three dimensions by expanding all six faces.

Chiang and Chirikjian analyzed how to perform motion planning in a lattice of rigid cubic modules able to slide past each other [15]. The CHOBIE robot developed by Koseki [53] is able to actually perform the sliding motion assumed by Chiang and Chirikjian in [15]. More recently, An developed the EM-Cube system [3] which is also capable of sliding motion.

Another unique lattice is the I-Cube developed by Khosla et al. [114, 90]. The 3D I-Cube system consists of passive cubes which are connected by active links with three rotational degrees of freedom that are able to grab, reposition, and release the cubes. The 3D I-Cube system was an improvement of the 2D system [42] developed by Hosokawa et al. for rearranging cubic modules in a vertical plane.

Goldstein, Campbell, and Mowry initiated the Claytronics project by publishing several papers [37, 22] proposing lattice-based "claytronic atoms" or catoms. These vertically-oriented cylindrical robots, which were incapable of independent motion, used 24 electromagnets around their perimeters to achieve rolling locomotion about their neighbors. Goldstein et al. envisioned a sys-

tem in which millions of smaller catoms could form arbitrary shapes using a randomized algorithm that avoided conveying a complete description of the shape to each module in the system.

The catoms continue to evolve. One of the newest instantiations [47] employs hollow cylinders rolled from SiO_2 rectangles patterned with aluminum electrodes. The authors hope that two of these cylinders, when placed in close proximity with their axes aligned, will be able to rotate with respect to one another using electrostatic forces. Specifically, the electrodes, (which reside on the inside of each cylinder and are electrically isolated by the SiO_2), will be charged so that they attract and repel mirror charges on the neighboring cylinder in a way that causes rotation. Currently, the system appears to be constrained to form 2D structures. The authors claim the completed system will have a yield strength similar to that of plastic and that the modules will be able to transfer power and communication signals capacitively from neighbor to neighbor.

The Claytronics project has proposed, but not yet demonstrated with hardware, the use of sub-millimeter intelligent particles as sensing and replication devices [88]. In particular, Pillai et al. present a theoretical 3D fax machine in which the object to be "faxed" is immersed in a container of intelligent particles that sense and encode the object's dimensions. At the receiving end, these same Claytronic particles decode the shape description sent by the transmitter and bond together to replicate the original object. Unlike our approach, Pillai's approach is completely centralized and relies on an external computer for computation.

White, Kopanski, and Lipson developed hardware and algorithms for several 2D stochastically-driven self-assembling systems [121]. To form specific shapes, each module is provided with a representation of the desired shape and decides, based on its location in the structure, whether to allow other modules to bond to its faces. Lipson et al. extended their 2D system to 3D [122, 108, 109] by using cubic modules suspended in turbulent fluid to achieve self-assembly and reconfiguration. As the free modules circulate in the fluid, they pass by a growing structure of assembled modules. When they come close enough, they are accreted onto the structure. The modules attract or repel each other with fluid suction or positive pressure. Early versions of the system used modules with interval values that could redirect these suction forces. More recently, Lipson's group has worked to move the intelligence and actuation capabilities from the modules to the tank in which

the modules circulate [113].

We developed the Miche system [34] consisting of 45mm cubic modules capable of mating with their neighbors using mechanically switchable permanent magnets. Each module contains three switchable magnets, each of which mated with a steel face on a neighboring module. Because the connectors were gendered, any collection of modules had to be assembled by hand so that the connectors were always oriented correctly, but the system was capable of self-disassembling to form 3D structures. The Robot Pebbles are based, at least in principle, on the Miche modules.

One of the newest lattice-type modular robotics is an aerial system composed of identical, hexagonal, single-rotor modules [82]. A group of modules may connect to form a flying platform with an arbitrary arrangement of multiple rotors. In addition to the ability to fly, each module contains wheels so that the system may self-reconfigure on the ground for the specific task at hand.

2.1.3 Truss Systems

Truss systems, as their name implies, are modular robotic systems in which the modules are nodes and edges in a truss structure. Both the trusses and connectors may be active in such systems. Unlike the lattice-based systems, truss-based systems do not need to operate on any regular lattice. Most truss-based systems under development employ struts that expand or contract to achieve structural deformation. One of the first such system to do so was Tetrobot [39]. The Odin system, conceived by Lyder, Garcia, and Stoy [67, 68] consists of three physically different types of modules: active strut modules capable of changing their length; passive strut modules of fixed length; and joint modules. The biologically inspired Morpho system [138] developed by Nagpal et al. is similar to Odin. It also uses active links, passive links, and connector cubes.

2.1.4 Free-Form Systems

Free-form systems are able to aggregate modules in at least semi-arbitrary positions. One such system is the Slimebot [101, 102]. The system consists of identical vertical cylindrical modules that move on a horizontal plane. The perimeter of each module is covered by six gender-less hook and loop patches used to bond with neighboring modules. These patches oscillate radially in and

out from the center of the body. By controlling the frequency and phase of the oscillations between neighbors, the system can achieve aggregate motion in a given direction.

Researchers are also developing algorithms for free-form systems. Funiak et al. developed a localization algorithm [29] that is capable of localizing tens-of-thousands of irregularly packed modules in 3D. Rubenstein, Shen, et al. developed a number of shape formation algorithms for collections of two-dimensional modules. These algorithms allow an arbitrary-sized collection of modules to form arbitrary scale-independent shapes [94, 95]. Once the shape is formed, modules can be added to or removed from the system, and the system will reconfigure itself to incorporate the new modules. The resulting shape will grow or shrink, but its basic form will remain unchanged. Recently, Rubenstein et al. developed a 1000-modules hardware platform on which to deploy these algorithms [93].

2.2 Other Programmable Matter Systems

So far we have focused on the most common programmable matter systems: those composed of robotics modules that rearrange themselves to form a variety of different shapes. Despite the popularity of these modular systems, there are many other creative approaches. One interesting approach to programmable matter uses the concept of jamming to create trusses with programmable stiffness [43]. By combining multiple trusses into a structure, the authors are able to change the structure's shape and material properties. At a larger scale, the authors propose creating larger "bags" of jammable material that can be formed into a specific shape and then made rigid.

Another alternative take on programmable matter is the paintable display [10]. The system demonstrates a type of virtual programmable matter by using hundreds of arbitrarily placed pixels to form text and images in a distributed manner. The system lacks actuation ability, but it demonstrates the realization of macro-scale "physical" properties from simple modules using local communication and limited computation power.

Whitesides et al. also developed a system with programmable optical properties [107]. In particular, the system employs a magnetic field to self-assemble a set of tiles whose surfaces are diffraction gratings. By applying different magnetic fields, the authors can create different config-

urations of the tiles and thus different optical patterns.

Researchers have also explored the use of folding to create a programmable matter systems [40, 4]. These systems use flexible wiring and shape memory alloy actuators embedded in composite sheets to programmatically create origami-inspired shapes. By controlling which actuators are energized, the system can form multiple different shapes.

2.3 Self-Assembling Systems

Self-assembling modular robotic systems are collection of modules that are capable of autonomously coalescing and bonding with their neighbors to form a greater structure. The result is often robotic, but it need not be. Whether a system is capable of self-assembling is independent of whether it is free-form, a chain, a lattice, or a truss-based system. Almost all of aforementioned modular robot systems rely on human intervention to assemble. In an attempt to automate the process of creating intricate modular robotic systems, researchers have attempted to mimic and improve upon natural self-assembling systems. Whitesides et al. investigated a wide variety of engineered self-assembling systems [124, 125, 30].

Miyashita et al. performed a more theoretical analysis of self-assembly using pie-shaped pieces to form complete circles [73] from pie-shaped pieces. In the process, they followed Hosokawa et al.'s lead [41] and modeled the system as a chemical reaction. Shimizu and Suzuki developed a system of passive modules capable of self-repair when placed on a vibrating table [103].

Computer scientists have also investigated theoretical aspects of self-assembly in the context of 2D tiles which selectively bond with their neighbors to form simple well-defined shapes like squares [92, 1, 2]. Each side of every tile in the system has an associated bonding strength. When two tiles collide, they remain attached only if their cumulative bond strength exceeds a globally defined system entropy. To form a specific shape, one must design a set of tiles with the appropriate bonding strengths.

Klavins et al. worked to develop intelligent self-assembling systems that employ triangular modules driven by oscillating fans on an air table to self-assemble different shapes [8]. The authors employ knowledge of the module's local topology and internal module state so that each

module decides, in a distributed fashion, when to maintain or break a connection with its immediate neighbors. Griffith et al., also worked with intelligent modules capable of selective bonding to show that self-assembling systems may self-replicate [38].

Matarić et al. [44] presented rule-based approach to self-assembly termed transition rule sets. In particular, they present a method that, given a goal structure, produces a set of rules shared among all modules that govern when and where new modules are allowed to attach to the growing structure. Zhang et al. [48] expanded on this work by optimizing the size of the rule sets used to form a specific shape. Werfel [120] also applied the idea of a transition rule set when studying the use of swarms to assemble complex structures from passive materials.

Other groups have attempted to make self-assembly more deterministic. The MEMS robots developed by Donald et al. [23, 24] consists of thin, (7–20μm), rectangular, (approximately $260\mu m \times 60\mu m$), scratch-drive devices capable of moving on an insulating substrate embedded with electrodes. The authors used four of these robots to build larger composite structures. The Sitti group has developed a similar system of micro-meter sized robots [85]. Instead of using a scratch drive for locomotion, the robots are manipulated by external magnetic fields. The authors can electrostatically clamp any number of robots to the stage on which they move. With all but one robot immobilized, the remaining robot may be moved independently. The system naturally self-assembles because the robots contain permanent magnets that attract their neighbors.

2.4 Simplifying Shape Formation by Self-Disassembly

The majority of existing self-assembly systems aim to form structures in one of two ways. Some systems such as [73, 103, 92, 1, 2] use a collection of application specific differentiated modules, that are only capable of assembling in a particular fashion to form a specific shape. In contrast, other systems such as [8, 78, 121, 122, 108, 44, 48, 120] use completely generic modules with more computation and communication ability embedded in each module. Both types of systems aim to form complex shapes in a direct manner: as these structures grow from a single module, new modules are only allowed to attach to the structure in specific locations.

We propose a new approach that eliminates many of the complexities of shape formation by

active assembly. Our Smart Pebble system employs a set of distributed algorithms to perform two discrete steps: 1) rely on stochastic forces to self-assemble a close-packed crystalline lattice of modules and 2) use the process of self-disassembly to remove the extra material from this block leaving behind the goal structure. By approaching shape formation in this manner, we hope to speed up the entire process, eliminate any global information that must be distributed throughout the system, and simplify the computing requirements of each module.

2.5 Simulators

In the process of developing our Smart Pebble system, we realized that we needed a way to test and debug the high-level algorithms which controlled the shape formation process. Using the hardware for development was too time consuming and difficult, so we created a simulator, named Sandbox, that is discussed in Chapter 4. Before deciding to create our own simulator, we explored existing alternatives. To better understand our reasons for creating Sandbox, we present details of many existing simulators here.

There a number of off the self robotic simulators designed for multi-robot system. Microsoft Robotics Developer Studio [72] and Webots [119] are two such systems that allow for high-level simulation of traditional mobile robot platforms. The Player Project [31, 89] is another popular system for high-level robotics simulations. The Player Project provides the Player network interface that allows the simulated robots to execute on any networked machine. The Stage and Gazebo components integrate with Player to simulate the world in which the robots exist, sense, and move. Chen et al. have extended Gazebo to simulate the OpenRTM-aist software framework [14]. While these systems are great for rapid simulation of traditional multi-robot systems, they lack the low-level configurability necessary to accurately simulate many modular robotic systems.

Many research groups have created their own mobile robot simulators. One of the first graphical simulators was the system created by Kurokawa et al. for the M-TRAN modular robot [62]. More recent simulators use a physics engine such as Open Dynamics Engine (ODE) [80] or PhysX [87] to simulate the interaction of the simulated modules with their surrounding environment. Tolley et al. have developed one of the newest physics based simulators to model the

stochastic self-assembly of their fluidic cubes [110].

Christensen et al. have developed the Simulator for Self-Reconfigurable Robots (USSR) [18] which they have used to simulate a number of common modular robotic systems. The USSR system is a physics-based simulator that aims to be easily extensible to any modular robotic system through the instantiation of a number of reusable components such as sensors and actuators. USSR is written in Java, but the interfaces with low level control code written in C. Fitch and Butler have also attempted to simulate a distributed robotic system containing millions of identical modules [27].

Carnegie Mellon and Intel Research have jointly developed the Dynamic Physical Rendering Simulator (DPRSim) [91, 25] to assist in simulating the Catom [88] modules. DPRSim simulates the physics of the Catoms, but it appears to largely ignore the challenges and unpredictability of low-level communication between neighboring modules. DPRSim, like the simulator we present in this paper, devotes a separate thread to each simulated Catom. It appears that the CMU/Intel team has improved the system so that it can be distributed across multiple host machines to allow the simulation of systems with millions of modules [5].

Our Sandbox simulator builds on the best developments of all these pre-existing systems. Sandbox is a low-overhead, easily reconfigurable, scalable simulator that allocates an independent process for each module allowing the simulated nodes to be executed on many separate workstations. Most important, the program used to simulate each module is compiled from the same source code that runs on the hardware. The only discrepancies exist to allow for differences in the low level hardware. A full discussion of the Sandbox simulator is contained in Chapter 4.

Chapter 3

Hardware

The unit modules developed in this thesis are $1cm^3$ cubes called the Smart Pebbles. The Smart Pebble was inspired by our previous work in self-disassembly demonstrated with the Miche hardware [34]. The Miche modules packaged power, communication, computation, and programmable connectors in an $85cm^3$ cube. The significant size reduction as we moved from the Miche to Pebble modules required hardware innovations that provide solutions to inter-module power, communication, and bonding challenges. Compared to the Miche modules, the Pebbles are 50 times smaller by volume, (as shown in Figure 3-1), about 5 times stronger when normalized by module weight, use gender-less connectors, and do not require recharging. Each module in the Pebble system is a 12mm cube capable of autonomously communicating with and latching to four neighboring modules in the same plane to form 2D structures. Each completed module weighs 4.0g and may be rotated any one of four ways on the assembly plane and still mate with its neighbors. The major functional components of each module are power regulation circuitry, a microprocessor, and four electropermanent (EP) magnets, which are responsible for latching, power transfer, and communication. We estimate that in quantity 50, each module costs $365.

Each Pebble module is formed by wrapping the flexible circuit labeled (a) in Figure 3-2 around the brass frame labeled (b) that is investment casted around a 3D-printed positive model. The flex circuit is a two layer design, and the entire stack-up including solder masks is 0.127mm (5mils) thick. The recommended minimum bend radius of the flex circuit is 10 times the material thickness,

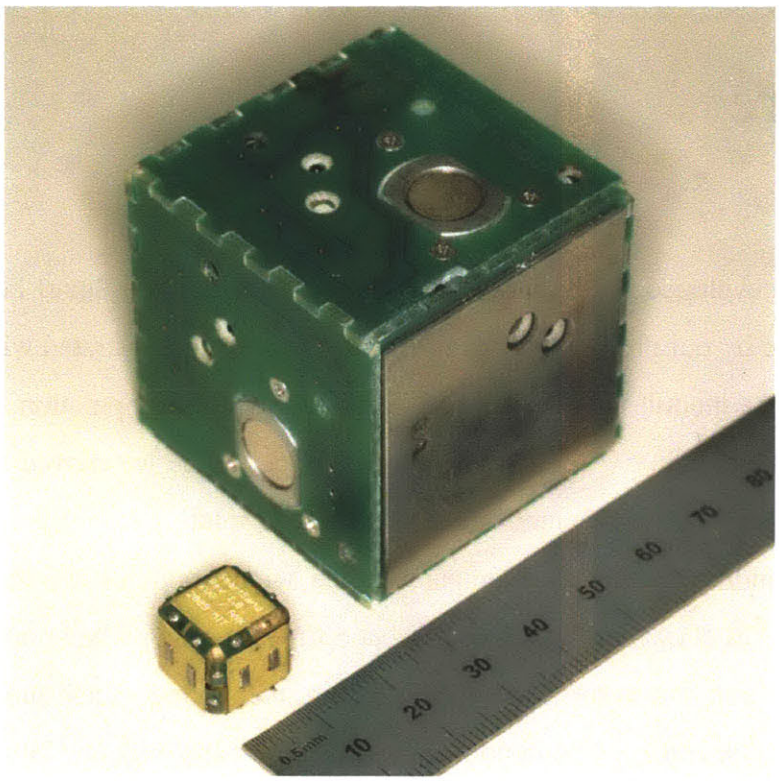


Figure 3-1: The Pebble modules are 50 times smaller by volume, (12mm vs. 45mm per side), and 5 times stronger by weight.

or 1.27mm. Therefore, we rounded the edges of the brass frame with 1.5mm radius fillets.

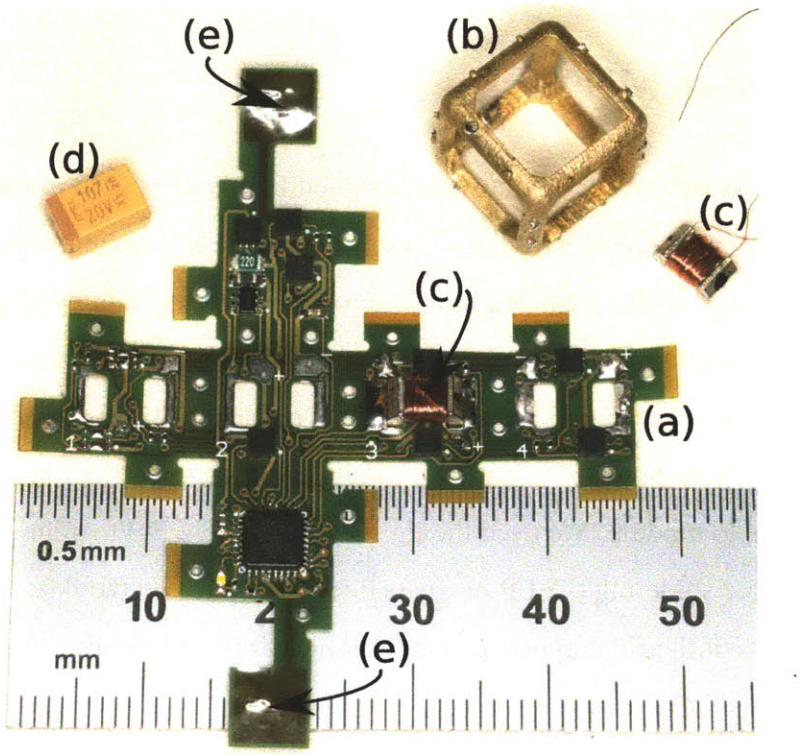


Figure 3-2: Each module is composed of a flex circuit (a), a brass frame (b), four electropermanent (EP) magnets (c), and an energy storage capacitor (d), which mounts to the bottom of tabs labeled (e).

Figure 3-4 shows how the flexible circuit is stiffened with 0.254mm (10mils) of Kapton in the six square areas corresponding to the six faces of the cube. In addition to helping flatten the faces of the cube, these stiffeners provide a rigid surface which prevents the solder bonding the components to their pads from breaking under stress.

The flex circuit is secured to the brass frame using a set of holes in the unstiffened portions of the flex circuit that mate with nubs on the frame. These holes and nubs align the flex circuit to the frame, and by soldering the flex circuit to the frame at these points, we form a secure bond between the circuit and the frame. This scheme allows for quick and easy disassembly of a module for service or debugging. Note, while a 3D system is theoretically possible, it would leave little room for electronics inside each module. Additionally, the pole arrangement of the EP magnets

would need to be made 8-way or axially symmetric.

3.1 Connection Mechanism

Figure 3-2 also shows two of the four custom designed EP magnets used in each module. At large scale, electropermanent magnets have been used for decades. Knaian miniaturized the technology to operate at millimeter-scales [51]. We have improved on the technology so that the EP magnets can deliver power and provide communication in addition programmed connections. The EP magnets are able to draw in other modules from a distance, mechanically hold modules together against outside forces (with zero power dissipation), communicate data between modules, and transport power from module to module. We used the jig shown in Figure 3-3 to align the EP magnets with the flex circuit before soldering the EP magnets directly to pads on the flex circuit. When assembled, the pole pieces of the EP magnets protrude slightly from the through the four sets of holes in the module faces.

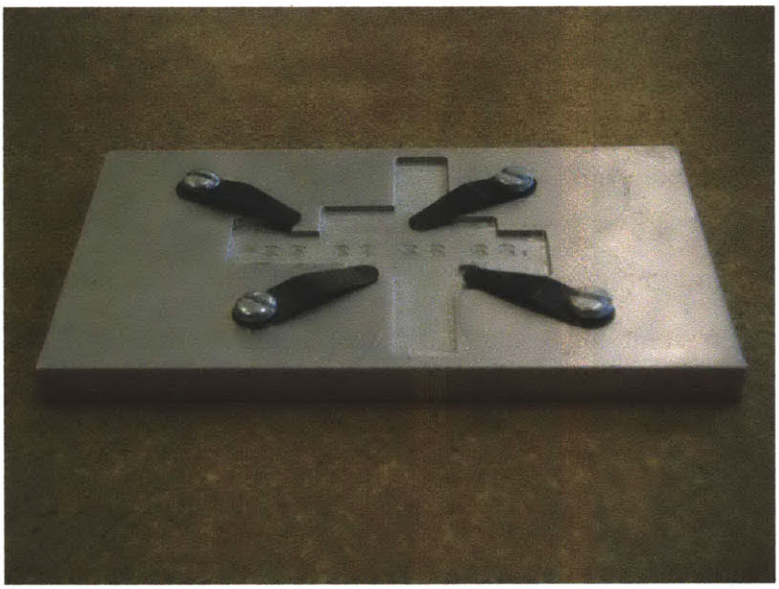


Figure 3-3: We used the jig shown here to align the EP magnets with flex circuit before they are soldered together. The jig attempts to ensure that the poles of the EP magnets are centered with the cutouts of each face and that the EP magnet pole pieces protrude slightly through the faces.

3.1.1 Electropermanent Magnet Theory

Figure 3-4 shows a partially assembled EP magnet. Each EP magnet consists of rods of two different types of permanent magnet materials, capped with soft-iron poles, and wrapped with a copper coil. One of the permanent magnets is Neodymium-Iron-Boron (NdFeB), and the other is Alnico V. Both of these materials have approximately the same remnant magnetization, about 1.2 Tesla, but very different coercivity; it takes about 100 times less applied magnetic field to switch the Alnico magnet than the neodymium magnet.

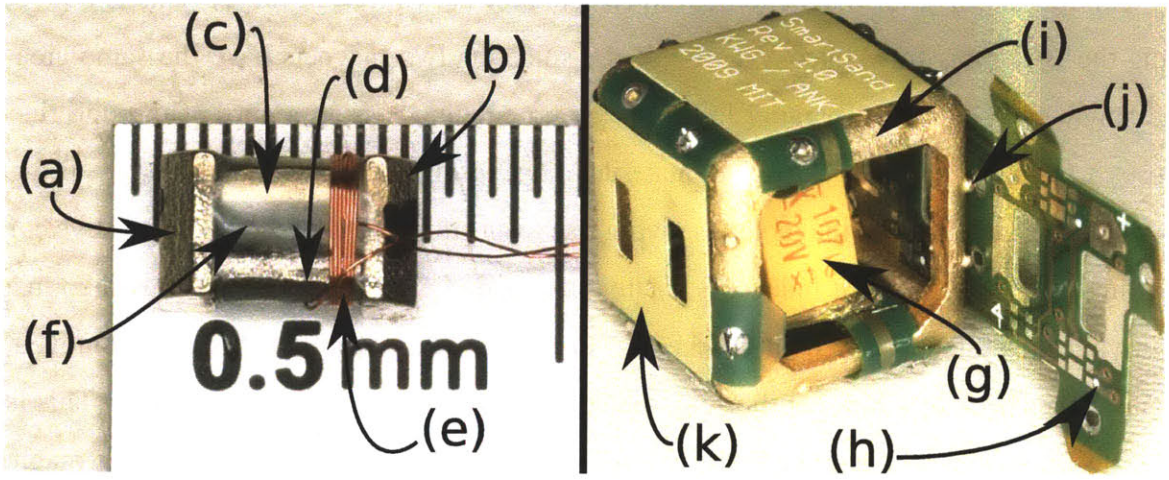


Figure 3-4: Each electropermanent (EP) magnet assembly is composed of two pole pieces (a,b) which sandwich cylindrical Alnico (c) and NdFeB (d) magnets. The entire assembly is wrapped with 80 turns of #40 AWG wire (e) and held together using epoxy (f) (which makes the Alnico magnet appear larger than its NdFeB counterpart). The reservoir capacitor (g) used to energize the EP magnet coils is soldered to the flex circuit (h) which wraps around and attaches to the brass frame (i) with a set of nubs (j). Once mounted, the EP magnets protrude 0.25mm through the stiffener (k).

The EP magnets are actuated as shown in Figure 3-5. A current pulse through the coil in the positive direction (Figure 3-5(b)) switches the polarization of the Alnico magnet so it is aligned with the polarization of the neodymium magnet. In this case, magnetic flux from both flows through the soft iron poles and to the other ferromagnetic object, attracting it. The attraction continues after the current in the coil is returned to zero (Figure 3-5(c)). We call this the "on" state of the connector. A current pulse through the coil in the negative direction (Figure 3-5(d))

switches the polarization of the Alnico magnet so it is opposite the polarization of the neodymium magnet. The polarization of the neodymium magnet is unchanged because it has a much larger coercivity. With the two magnets having opposite polarization, magnetic flux circulates inside the device but does not leave the poles, and thus does not exert force on the other connector or external ferromagnetic objects. Once again, this flux pattern continues after the current is returned to zero (Figure 3-5(e)). We call this the "off" state of the connector.

To understand the origin of bi-stability in an EP magnet, it is helpful to examine the B/H (magnetic flux density vs. magnetic field intensity) plot shown in Figure 3-6. This is derived by adding the B/H plots for Alnico V and NdFeB, since the two magnets have the same area and same length, and appear in parallel in the magnetic circuit. Passing a current through the coil imposes a magnetic field, H, across the materials. The resulting magnetic flux density, B, passes through the air gap between the modules giving rise to an attractive force. While a positive current is flowing through the coil, it induces a positive magnetic field, H, saturating the Alnico magnet and driving the system to the point marked (a) in Figure 3-6. When that current is removed, the system relaxes back to a new equilibrium, labeled (b), with positive flux but no field. This is the "on" state. Momentarily passing a negative current through the coil saturates the Alnico magnet to the negative field side driving the system out to point (c) in Figure 3-6. Once the negative current is removed, the system relaxes to the zero field, zero flux "off" state marked by point (d). If the magnets are pulled apart while on, a demagnetizing field appears, reducing the flux and resultant force.

The EP magnets used here are low average power but high peak power devices. Our system uses a 20V, 5A, 300μs pulse provided by a 150μF capacitor in each module to switch their state. The time-averaged power devoted to magnetic attraction is many orders of magnitude lower than would be required using equivalent electromagnets. We calculate that an equivalent electromagnet would consume 10W continuously. The EP magnet consumes 100W over 300μs when switching (on or off). Therefore, so long as the EP magnet is switched less than once every 3ms, the EP magnet will use lower time-averaged power. For more information about the EP magnets, including detailed design guidelines and a quantitative model, see [51].

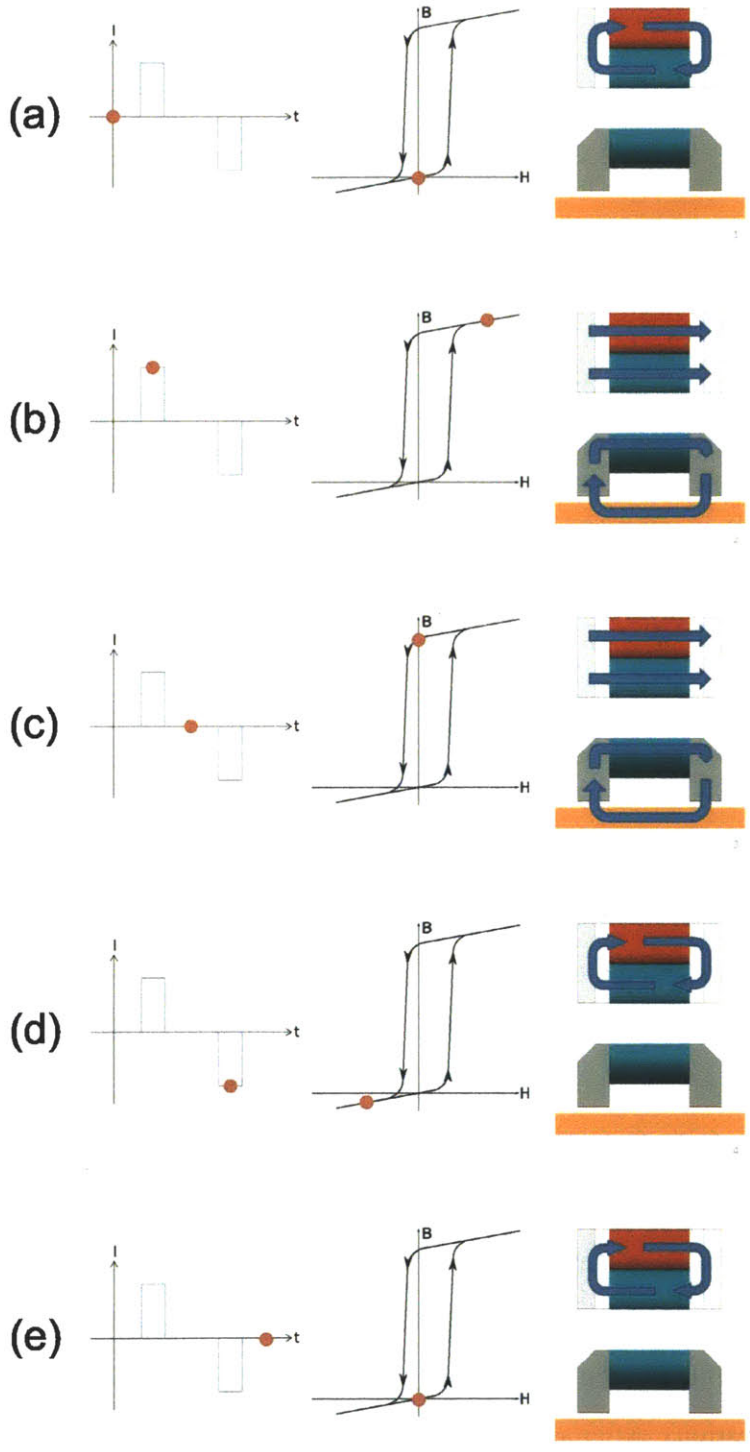


Figure 3-5: Here we show a series of snapshots that capture the process of activating and then deactivating an EP magnet. Each snapshot, labeled (a) through (e), captures the applied current, operating point on the magnetic flux-field curve, and flux flow path.

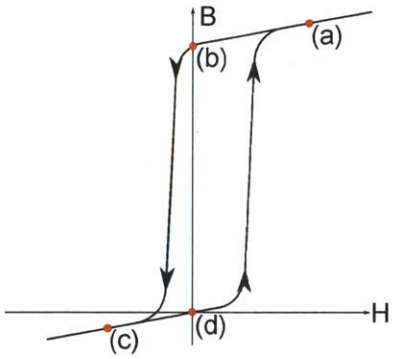


Figure 3-6: The hysteresis curve for the EP magnet assembly shows the origin of the magnetic bi-stability in the device. The current pulse which turns the EP magnet on drives the system to the point labeled (a). Once the current pulse is removed, the system settles to point (b) where the net remnant flux results in the assembly attracting nearby ferromagnetic materials. To turn the EP magnet off, a current pulse of opposite polarity drives the system to point (c). Once that current pulse is removed, the system settles to point (d) where there is no net remnant flux. As a result, the EP magnet does not attract nearby ferromagnetic materials and is off.

3.1.2 Electropermanent Magnet Construction

The magnetic rods and pole pieces were custom fabricated by BJA Magnetics Inc. The magnetic rods are grade N40SH NdFeB, and cast Alnico 5, both 1.587mm diameter and 3.175mm long, magnetized axially. The magnetic rods were fabricated by cylindrical grinding. The magnetic rods were coated with 5μm of Parylene by the Vitek Research Corporation. The pole pieces are 3.175mm by 2.54mm by 1.27mm blocks of grade ASTM-A848 soft magnetic iron, with a diagonal notch cut off to allow clearance when four are placed inside each module. The pole pieces were fabricated by wire EDM, and chromate coated to slow corrosion and facilitate solderability. We assembled the rods and pole pieces with tweezers under magnification, using the mounting plate shown in Figure 3-7 to hold the pole pieces and magnetic rods in position while we glued them together. The rods are glued to the pole pieces using Loctite Hysol E-60HP 60-minute work time epoxy (Henkel Corporation). After assembly, we ensured that the two pole faces were co-planar by rubbing the assembly against a 320 grit aluminum-oxide oil-filled abrasive file (McMaster-Carr). Then, we wound an 80-turn coil around the magnetic rods using #40 AWG magnet wire (MWS Wire Industries). For more details about the fabrication process, see Knaian's thesis [51].

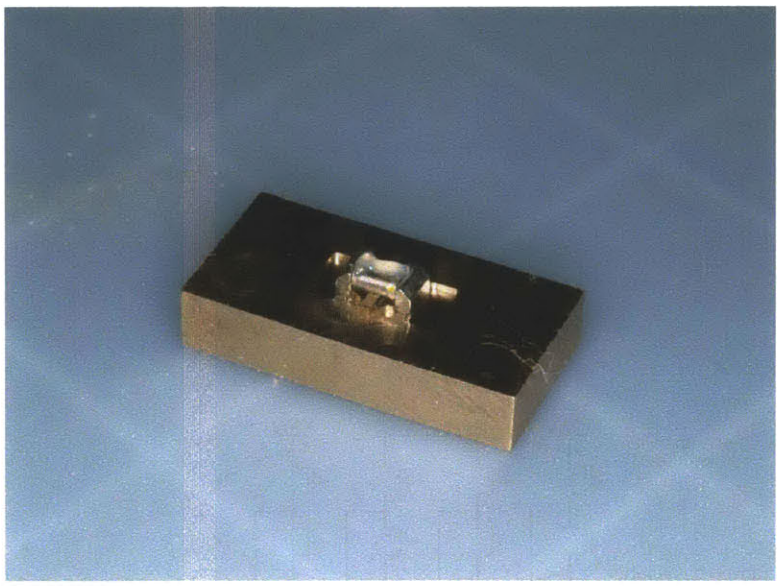


Figure 3-7: We used this jig to hold the magnetic rods in alignment with the pole pieces while applying epoxy to form the basic structure of the EP magnets.

3.2 Power Electronics

The four EP magnets in each module are driven by a set of 2mm square MOSFETs which are capable of handling the 5A required to switch the EP magnets (Fairchild Semiconductor FDMA2002NZ and FDMA1027P). In order to reduce the component count, we did not dedicate a full H-bridge to each EP magnet coil. There was just not enough space available inside each module to do so. Instead, each EP magnet has one dedicated half-bridge connected to one side of its coil. We call these the "face-specific" drivers. The other sides of the four coils are tied together and serviced by a single "common" half-bridge as shown in Figure 3-8. Using this configuration, we are able to pass current in both directions through each of the EP magnet coils, one coil at a time.

The two control lines for the common half-bridge are driven by the microcontroller's timer output compare pins. Using the output compare pins we can precisely control the duration and spacing of the current pulses flowing through the EP magnet coils.

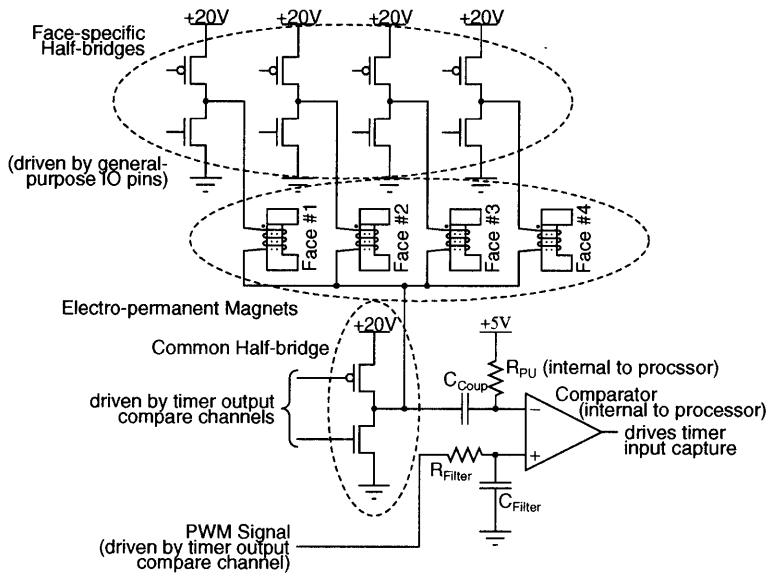


Figure 3-8: The four EP magnets are driven by a set of four face-specific half-bridges and one common half-bridge in order to reduce the modules' component count and circuit area. C_{Coup} allows the processor to detect communication pulses from neighboring modules. Except for four level shifters used to drive the PMOS devices, a voltage regulator, LED, and the processor, this is essentially the entirety of the electronics in each module.

3.3 Processors

Each module is controlled by an Atmel ATmega328 processor which offers 32KB of program memory and 2KB of RAM in a 5mm square lead-less package. To minimize the external component count, we employ the processor's internal 8MHz RC oscillator to clock the processor. We routed the processor's SPI and debugWire pins to pads on the outside of each module. We constructed a test fixture (see Figure 3-14) to contact these pads with spring-loaded pogo-pins allowing us to communicate with, reprogram, and debug the modules. When loaded with the shape duplication algorithm discussed in Section 7, we come close to completely filling both the processor's flash and RAM.

3.4 Bonding

In each module, the microprocessor sends control signals to the power electronics which allow the modules to mechanically bond with their neighbors. To characterize the strength of these bonds, we performed a number of pull-test experiments with two neighboring modules [33]. One module was mounted on a linear motion stage, and the other on an air bearing, with a load cell measuring the force along the air bearing's direction of motion. The experimental setup is shown in Figure 3-9. For each pull test trial, the module attached to the motion stage is connected to an external power source through an attached magnetic connector. The linear stage drives the modules together, and when they come into contact, the second module powers up. Once both modules have power, they exchange messages and energize their EP magnets. The duration and timing of these energizing pulses controls the strength of the resulting connection. Once the modules are bonded, we measure the strength of their connection by driving the motion stage so that it pulls the modules apart while recording the force exerted on the load cell. The force on the load cell grows until it reaches the connector's bonding strength at which point the two module snap apart.

The normal bonding force resulting from three different latching waveforms is shown in Figure 3-10. The average holding force, (over nine tests), for two asynchronous pulses, (one from each magnet), was 2.16N. When both magnets were pulsed synchronously, the resulting force was

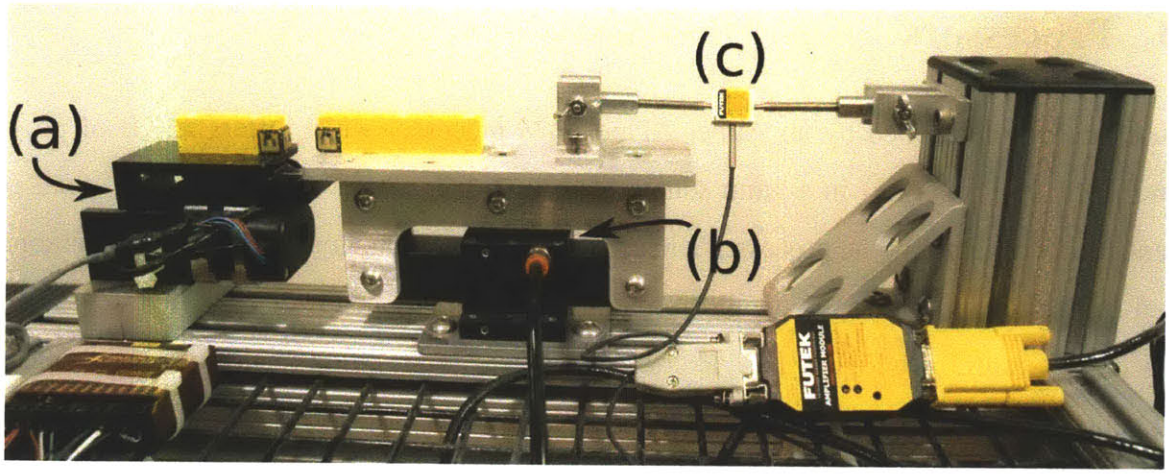


Figure 3-9: To characterize the bond strength between neighboring modules, we fixed one Pebble module to a linear motion stage (a) and another to a linear air bearing (b). The opposite side of the air bearing was attached to a load cell (c) which measured the force between the two modules. Once the two modules were bonded, the motion stage pulled its module away from the other while we recorded the force exerted on the load cell.

2.06N (averaged over 15 tests). When both magnets were pulsed synchronously twice, the average peak force was 3.18N (averaged over 4 tests). These results make physical sense. Synchronous pulses produce a stronger magnetic field, and repeated application of this field drives the EP magnet farther into the first quadrant along its B-H curve resulting in a larger remnant flux.

In addition to the normal force required to separate two modules, we measured the shear force between two modules using the same fixture. It was difficult to separate the effects of friction from the shear magnetic force. Five shear tests yielded forces of 0.22–0.83N with an average of 0.69N. Finally, we measured the remnant normal force after the magnets had been switched off to determine whether unused modules in an ensemble would easily separate from the goal shape. In ten trials, we were unable to measure any remnant force holding the modules together after their EP magnets had been deactivated. (The measurement noise of the force sensor is zero-mean with a standard deviation of 0.0068N.) We can use the fact that a magnetically suspended EP magnet naturally falls off of its mating surface when deactivated to upper-bound the remnant force by 0.002N (the force due to gravity on a single EP magnet).

Returning to the normal force pull test results in Figure 3-10, all three traces show an initial

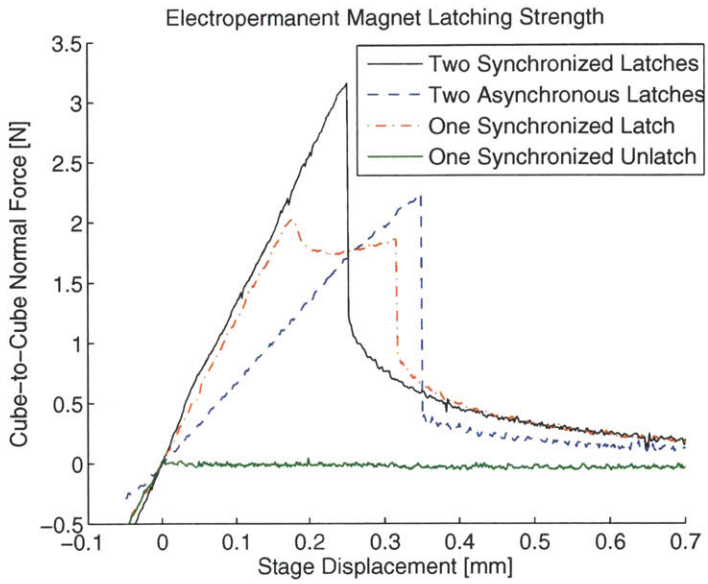


Figure 3-10: The latching force between modules is strongest when each module energizes its magnet assembly with two synchronized latching pulses spaced far enough apart that the 150μF reservoir capacitors have time to recharge [33].

linear rise in force with displacement, corresponding to the elastic deformation of the modules, (and the load cell spring), as they are pulled apart before the magnetic connectors separate. A peak is reached, and then the LED in the load-cell-side module extinguishes, corresponding to separation of at least one pole of the connectors, and the force decreases as the air gap distance between the magnets increases.

The distance over which the connectors remain in contact as the stage displacement increases, (the distance from 0 displacement until the peak force), provides a way to measure the tolerance to non-uniformity and misalignment in a large collection of modules. A large network of modules is mechanically over-constrained, so one might be concerned about the ability to get reliable power transmission between modules, which requires continuous contact. From the pull tests, one can see that a displacement between 0.25–0.35mm (2–3% of the total module size) is possible before separation, allowing a large network of modules to achieve precision connector alignment through elastic averaging.

In the single synchronized pulse experiments, (red dash-dot line in Figure 3-10), we observed

a plateau in force following the peak, before the rapid decrease. Observation was difficult, but it appeared that the plateau corresponds to a case where one pole of the connectors is still in contact while the other is separated. After separation, there is a non-continuous jump in the data down to a lower force. We suspect this is because, after the connectors are pulled apart, and the contact force has been removed from the system, the magnet pulls away and a new static equilibrium between the magnetic force and load cell stiffness is reached.

Figure 3-11 illustrates the coil current and voltage during a single synchronized pulse. Looking at the voltage and current data, we can see that the current reaches a momentary peak and then decreases during the pulse, indicating that the magnetic material is not saturating during the pulse, but that the peak current is instead limited by the discharge of the capacitor. This was the inspiration for the double synchronous pulse, (which energizes the coils a second time after waiting for the capacitor to recharge), and as Figure 3-10 shows, it does reach a higher force level. The force measured for the double synchronous pulse is 72% of the 4.4N figure measured in [51] for a single magnet being pulled away from an iron plate, in which a stiff power supply was used and full saturation of the magnetic material achieved.

In addition to testing the modules' ability to remain connected, we wanted to verify their ability to draw in and latch with other modules in close proximity. We performed two different experiments. In the first, one module had its magnets off while the other module had its magnets on. One module was fixed while the other was free to move on the non-sticky side of cellophane tape. The modules were aligned and their faces parallel. In 30 trials, the modules always successfully attracted and latched when their initial separation was 2.48mm. The second experiment was identical except that the magnets in both modules were energized. In 28 of 30 trials, the two modules latched from an initial separation of 4.31mm. These experiments encourage the idea that a collection of modules will be able to successfully self-assemble in the presence of stochastic environmental forces.

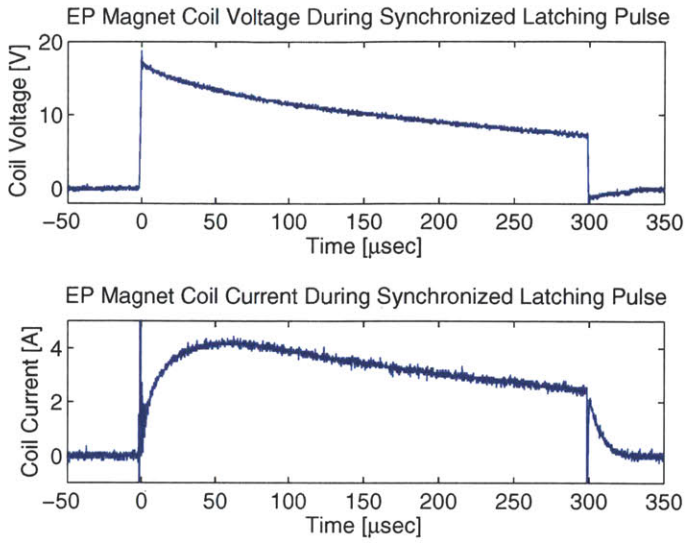


Figure 3-11: The EP magnet coil current peaks and then falls during a 300μs latching pulse indicating that the magnets are not fully saturated. The current does not reach a plateau because the capacitor discharges too quickly [33]. (Ignore the short switching transients in the current data.)

3.5 Communication

The EP magnets form an inductive communication channel between neighboring modules. In short, when two EP magnets are in contact, they behave just like a 1:1 isolation transformer. We utilize this fact to transfer data between modules without affecting their ability to latch together. All inter-module communication occurs at 9600bps using a series of 1μs magnetic pulses induced by the coil of one EP magnet and sensed by the coil of the neighboring EP magnet assembly. The presence of a single 1μs pulse during a bit period signifies a logical '1' while the lack of any pulse signals a '0'. Neighboring modules transfer data using pulses of the same polarity as the pulses used to latch the EP magnets. As a result, there is no risk of the latching strength decreasing over time during intensive communication.

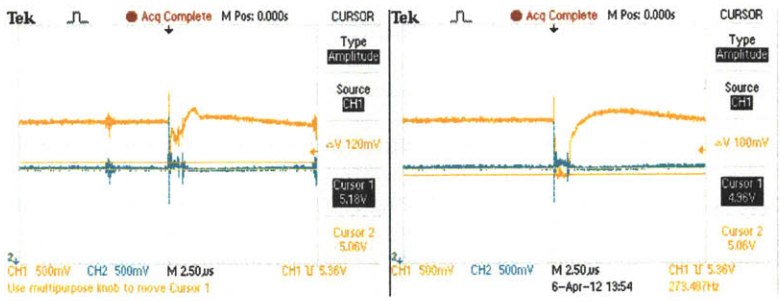
Because the four EP magnets share a common half-bridge (see Figure 3-8), a module is unable to discriminate between incoming messages if it is listening for messages on multiple faces. To select the face on which the module is listening, the face-specific high-side MOSFET of one face is turned on while the three others coils are left floating. Additionally, the common side of all four

EP magnet coils is left floating, but it is capacitively coupled by C_{Coup} to the processor's analog input. The internal pull-resistor (R_{PU}) on this analog input is enabled. Internally, the processor routes this signal to the inverting input of its internal analog comparator. Figure 3-8 shows the components used when receiving a message.

The non-inverting input of the processor's comparator is driven by a DC voltage that we generate by low-pass filtering the output of another of the processor's timer channels. Specifically, we employ one of the processor's output compare channels to generate a variable duty cycle square wave. Figure 3-8 shows hows this square wave is filtered by a passive first-order RC filter (R_{Filter} and C_{Filter}) to produce DC level which varies with duty cycle.

The module sending data to a neighbor does so by applying a $+20V$ pulse between the face-specific side and the common side of one of its EP magnet coils. This pulse will induce a current flow from the face-specific side to the common side. Because the EP magnets in the neighboring modules are oriented north-to-south, their coils are effectively wrapped in the opposite directions. Therefore, the current induced in the receiving module's coil will flow from the common side to face-specific side. The current drawn from the common side will be be sourced by the AC coupling capacitor. Figure 3-12 illustrates the negative-going voltage spike that is induced across the capacitor. Because the pull-up is enabled on analog input connected to this capacitor, the inverting input of the analog comparator will see a nominal voltage of VCC with short negative excursions corresponding to the magnetizing pulses sent by the neighboring module. When these pulses drop below the threshold voltage driving the non-inverting input, the comparator's output will transition from low to high. These edges drive a timer input capture channel so that they may be carefully measured and interpreted as inter-module messages.

The threshold voltage applied to the non-inverting must be selected carefully. Figure 3-12 illustrates that there is a significant amount of cross-talk between EP magnets on different faces. Even if a module is not explicitly listening for transmissions from a given neighbor, if that neighbor is transmitting, it will still induce negative-going pulses at the inverting input to the listening module's internal comparator. While these pulses are smaller in magnitude that the pulses that result from a neighbor attached to the face on which the receiving module is actively listening, they are



(a) (b)

Figure 3-12: The receiving EP magnet in a pair of Pebble modules sees negative-going pulses at the input to its internal comparator. The threshold voltage against which these pulses are compared is critical because there is significant cross-talk between all faces. In particular, (a) shows that a pulse resulting from a face to which the module is not currently listening barely remains above the threshold. In comparison, (b) shows a pulses from the face on which the module is actively listening. It does cross the threshold, but only by 100mV.

not trivial. We have found that a threshold voltage of 4.05V works well in practice to differentiate the two pulse magnitudes.

3.6 Power

The Pebble modules do not contain their own power sources. Instead, electrical power is distributed from one or more centralized sources and then transferred from one module to the next. In practice, we use the spring-loaded pogo pins of the test fixture shown in Figure 3-14 to supply 20V to what we term the root module. From the root, power is transferred between units via Ohmic conduction of DC power through the soft magnetic poles of the connectors. Each module contributes a resistance of 0.3Ω. Given that the quiescent current of each module is 15mA, each module in a chain results in a voltage drop of 4.5mV. In theory, a 20V source could power a chain of 3266 modules before the voltage supplied to the trailing module falls below the dropout voltage of the regulator used to power the microprocessor. In practice, the ability of the EP magnets to change state would be compromised after several hundred modules. Typical configurations will consist of more than a single chain of modules thereby providing many parallel electrical paths that would noticeably reduce the electrical resistance between any two points.

Within each module, the EP magnets are mounted to the flex circuit, which serves as an elastic mount, allowing slight bending as needed for the two magnetic connectors to achieve intimate contact. When one magnet is turned on, it attracts any nearby neighbor; contact is achieved; the adjacent module receives power, starts its program; and the two modules communicate to drive a series of synchronized pulses through their magnets to bond more strongly. All of the magnetic materials used in the connector are good conductors of electricity, so it was necessary to coat the rods of Alnico and NdFeB separating the two poles with Parylene to electrically isolate the two poles.

Each module contains a 150μF tantalum low equivalent series resistance, (low ESR), reservoir capacitor. These capacitors, one of which is labeled (d) in Figure 3-2, are responsible for sourcing the high-current demands of the EP magnets when they are switching on and off. These capacitors fill the interior of each module and can only be installed once the flex circuit is partially folded around the brass frame. In particular, the capacitor is soldered by its ends to the bottoms of two tabs labeled (e) in Figure 3-2 so that it floats, suspended, in the interior of the module.

The connectors on the four mating sides of the module are identical, and placed so that the magnetic north is always on the right (when viewing the face head-on), and the magnetic south is always on the left. Regardless of their rotations about a vector orthogonal to the assembly plane, when two modules are placed together, the magnets will align north-to-south. Internal to each module, all of the north poles of the EP magnets are tied together in one electrical net, and all of the south poles are connected to another. Therefore, in a chain of modules, the north pole net will alternate between serving as the electrical ground and the 20V rail. This is illustrated by Figure 3-13.

In a large network of modules, every circular path back to the same module passes through an even number of connector pairs, so there is no arrangement than can result in a short circuit. Internally, a bridge rectifier is used to produce a voltage with known polarity from the unknown polarity present on the north and south pole nets. As a result, the modules are four-way rotation symmetric.

Additionally, a bridge rectifier inside each module allows for a module to be flipped upside

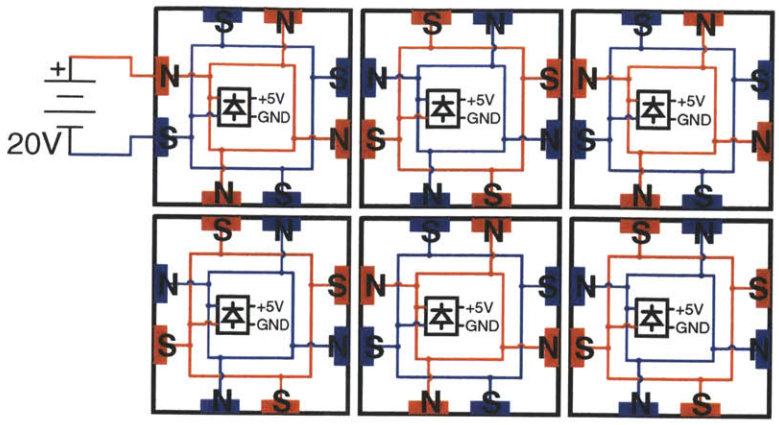


Figure 3-13: In a lattice of Pebble modules, neighboring modules alternate which of their magnetic poles (labeled N/S) serve as the 20V rail (red) or electrical ground (blue). The unknown polarity is converted to a known polarity using a bridge rectifier, and then a linear regulator is used to produce 5V to drive the module's processor.

down without affecting either the underlying electrical grid that it forms with its neighbors The main problem associated with inverting a module is that its EP magnets will begin to repel the neighboring modules instead of attract them. This is because the magnets' poles will align north-to-north and south-to-south instead of north-to-south.

There are many advantages to replacing the batteries typically found in robotic systems with capacitors. Primarily, we are able to decrease both the size and complexity of the modules. Not only are batteries significantly larger than capacitors, they require additional protection and charging circuitry in addition to a step-up converter to produce 20V for the EP magnets. Second, by eliminated batteries, which are short-lived with respect to other electronic components, we extend the potential lifetime of the modules. Additionally, the lithium-polymer batteries typically used to robotic systems are more toxic than other electronics, and can explode or catch fire if not handled with care. Finally, we eliminate the need to recharge the modules which can become a significant inconvenience as the number of modules grows into the millions.

One of the smallest lithium-polymer batteries available supplies 3.7V, has a capacity of 170mA, but occupies 2500 cubic millimeters[6]. To fit such a battery into our modules, the modules could be no smaller than 25mm per side. The 100μF capacitor in our modules is rated for 20V and only consumes 130 cubic millimeters. As a result, our modules are only 12mm per side. By using a

capacitor rated for 20V, the same voltage used to energize the EP magnets, we eliminate the need for a step-up voltage converter. Such a converter would be necessary if we were using any less than five lithium-polymer cells connected in series. By avoid lithium-polymer batteries, we also eliminate the need for the charging and protection ICs that typically accompany them. The only power conditioning IC present in the modules is a simple linear regulator that produces 5V to supply the microprocessor.

If, during the course of additional development, we find that the system must be untethered from all power sources, we could create passive battery modules that we mixed in with the active modules described here. These battery modules would be larger than the active modules but could be fabricated with the connector connector spacing.

3.7 Test Fixture

We have developed a test fixture which we use when running hardware experiments with the Pebbles system. It provides a method to supply what we term the root node with power, and it provides a communication link between the root and the user's personal computer. The test fixture and the mating pads on a Pebble module are shown in Figure 3-14. To communicate with and power the root, the test fixture employs seven spring-loaded pogo pins that protrude through an assembly platform constructed from laser-cut acrylic. Two of the seven connections provide 20V to the root module. Three provide standard the standard SPI bus signals (MOSI, MISO, and SCLK). One is connected to the processor's reset pin, which also serves as the debugWIRE interface, and the final connection is the signal ground for the two communication interfaces.

The test fixture itself contains a second Atmel processor that serves as a communication gateway. It communicates with the attached module using its SPI interface. The test fixture is the slave, and the Pebble is the master in this pairing. The test fixture takes whatever data it receives from the Pebble and translates it to a low voltage RS-232 serial data stream that it sends to an FTDI serial to USB converter. When attached to the user's PC, the USB interface appears as a serial port. The user can send data back to the root module through the same chain of interfaces.

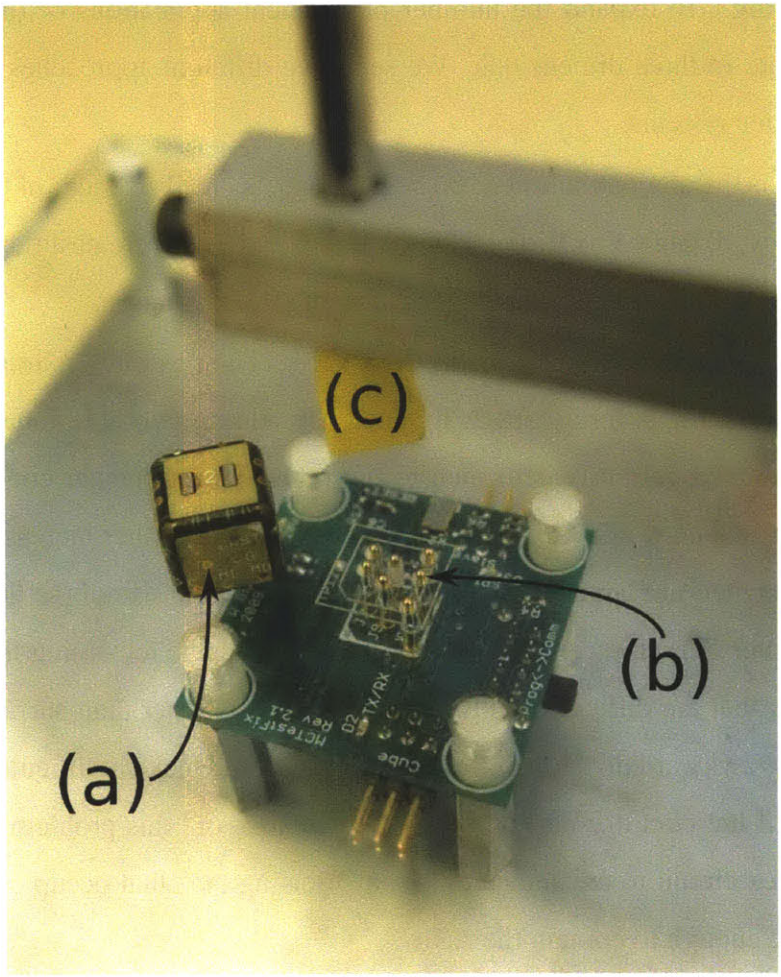


Figure 3-14: Each Pebble module has a set of seven electrical pads (a) on the outside of its bottom face. These pads mate with the spring-loaded pogo pins in a test fixture (b) which provides power and a communication link to the user's PC. The plunger (c) is used to hold the module in contact with the pogo pins.

3.8 3D Modules

The current generation of Smart Pebbles is only able to operate in the plane. Furthermore, the Pebbles cannot be flipped upside down. If they are, the EP magnets, when activated, change from attracting to repelling. To expand the number of practical applications of the system, it needs to be able to operate in three dimensions. We see three different approaches to achieving a 3D programmable matter systems.

The first option is the obvious solution: place EP magnets on all six faces of the Pebbles making them invariant to any 90 degree rotation. This solution provides the greatest flexibility and highest degree of redundancy when assembling the modules into a 3D structure.

The six-connector solution is not without drawbacks. The flex circuits in the current version of the Pebbles are already severely space limited. By adding two additional EP magnets, we would eliminate the area currently dedicated to the processor and power conditioning circuitry. (The additional EP magnet would also require additional drivers further increasing the component density.) The EP magnets are large components with respect to the size of the flex circuit and must be placed in the center of each face. As a result, they subdivide the remaining flex circuit area into many small parcels that are difficult to utilize for components other than surface mount resistors and capacitors. This awkward division of flex circuit area would make it difficult to utilize an ASIC that combined all of the circuitry into one IC. One way to avoid this problem may be to modify the design of the flex circuit to create an additional "floating tab" that occupies the middle of the module and is large enough to contain the ASIC.

The second problem with placing EP magnet connectors on all six faces is that the connectors would need to be redesigned. Currently, the connectors are only 2-way symmetric, but they would need to be 8-way or axially symmetric in the 3D system. Figure 3-15 shows a cross-section of one possible design of an axially symmetric EP magnet.

An alternative to employing six active faces in each Pebble is to create two or three distinct types of Pebbles, each capable of bonding with neighbors in separate planes as shown in Figure 3-16 One can think of this strategy as forming a structure as a stack of unbonded layers and then bonding the neighboring layers together with special "out of plane" Pebbles. We could continue

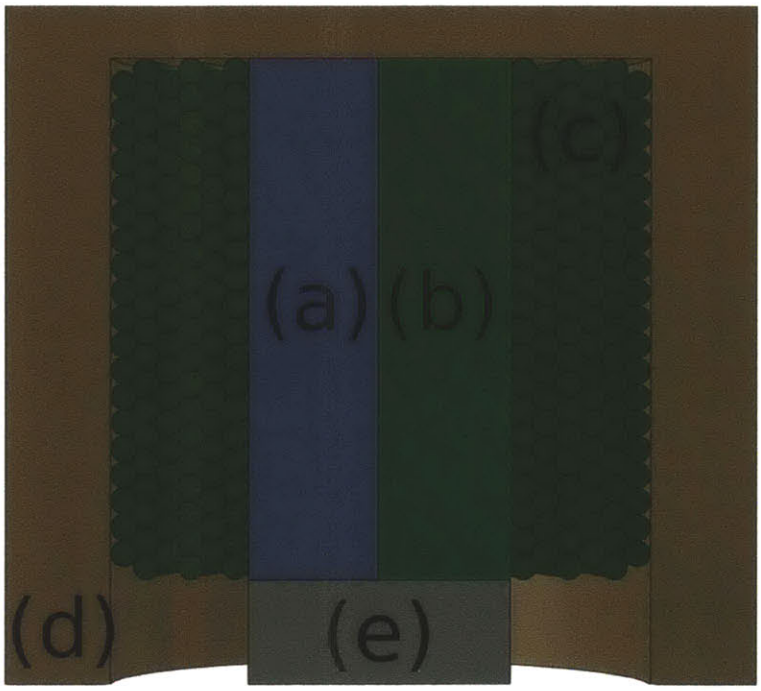


Figure 3-15: An axially symmetric EP magnet could be created by placing two half round magnets (a,b) next to each other to form a core than is then wound with a coil (c) and placed inside of a ferromagnetic cup (d). A small cap (e) is attached to the exposed end of the magnetic core to prevent fringing fields from giving rise to attractive forces when the EP magnet is deactivated.

using our current set of Pebbles for bonding in the X-Y plane, but we would then design two new types of Pebbles (still with just four connectors) capable of bonding in the X-Z and Y-Z planes. Starting with a sheet of X-Y type Pebbles, we could replace some of the modules with X-Z and Y-Z modules. On top of each of these new Pebbles we would place another X-Z or Y-Z module, respectively. Then, the remainder of the second layer could be filled with the standard X-Y Pebbles.

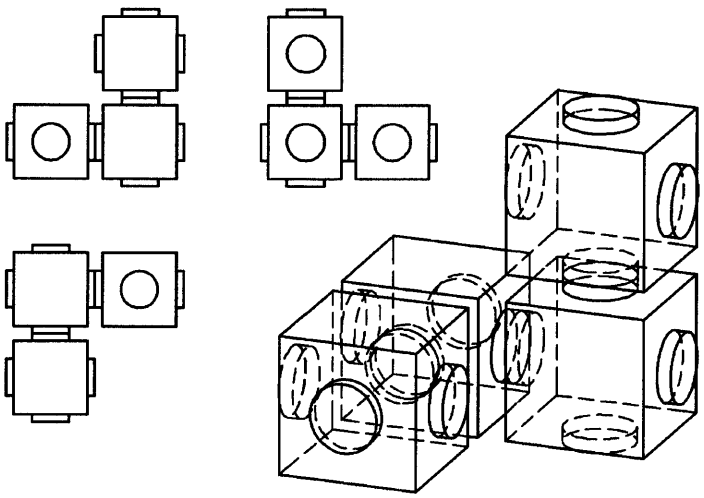


Figure 3-16: By using distinct types of Pebbles capable of bonding in either the X-Y, X-Z, or Y-Z planes, we can create 3D structures using only four connectors per module.

For a large structure containing an equal proportion of all three types of modules, there will, on average, be one third of neighboring faces which are not connected. In comparison to a system in which there are EP magnets on all six faces, this will weaken the structure and limit the communication pathways through it. As with the rotation invariant system, the connectors will need to include additional degrees of symmetry because stochastic forces will ensure that the modules touch in every possible orientation.

3.9 Miniaturization

If we want to continue to shrink the Smart Pebble modules so that they truly become Smart Sand, we face many engineering challenges. We must miniaturize both the inter-module connectors and

the modules themselves. We want our miniaturized modules to be three-dimensional particles that are able to form three-dimensional macro-scale objects. There are many microfabrication technologies under development with potential to help miniaturize the Smart Pebbles. Despite the variety of options, there are few, if any, automated processes that could fabricate Smart Sand without a large design effort. Many cutting edge microfabrication technologies are are focused on just a single aspect of what will need to be a larger multi-step integrated fabrication process. Integrating these individual technologies into a procedure that is able produce a million grains of Smart Sand will require significant effort.

3.9.1 Connector Technologies

In miniaturizing the connectors, there are good reasons for continuing to use EP magnets instead of competing options. Primarily, EP magnets consume minimal power, are simple to control, and exert forces that scales with their footprint area [51]. Wire-wound 0402 inductors are already fabricated in huge quantities. These surface mount devices are roughly 1mm long, 0.6mm wide, and 0.6mm tall. Smaller coils (0.5mm long and 0.2mm in diameter) are also possible [7].

In what follows, we compare EP magnets with several alternatives. Mechanical connectors are still the most popular bonding mechanism for modular robots, and they are one potential alternative to EP magnets. At scales larger than the Smart Pebbles, mechanical connectors provide favorable properties such as large bonding forces, fully constrained mates, and favorable strength to weight ratios. If we attempt to scale traditional mechanical latches down to the millimeter-scale, they will become increasingly difficult to fabricate and relatively fragile. Additionally, mechanical latches require precise alignment that is difficult to achieve at any scale.

Traditional electromagnets are another connector option. Compared to EP magnets, electomagnets are simpler to construct and control. Instead of several types of magnetic material, electromagnets can be built from a coil wrapped around a single piece of iron or magnetic steel. Such a device is on and attractive when the coil is energized and off otherwise. The most compelling reason to avoid electromagnets is their large power consumption. Consider an electromagnet with dimensions and holding force equivalent to EP magnets used in the Smart Pebbles. It is only a

matter of milliseconds before it is more efficient to use an EP magnet than to keep an electromagnet energized [51]. As we continue to shrink the Smart Pebbles, power dissipation will become increasingly important. In densely packed 3D structures in particular, the modules must dissipate a minimal amount of energy so that resultant heat does not destroy or incapacitate the system.

As a compromise between electromagnets and the EP magnet design that we present above, it is possible to completely remove the high coercivity NdFeB magnetic rod from our design. The remaining Alnico rod can be enlarged to fill the resulting void, or the whole structure could be miniaturized. The resulting device would still be an electropermanent magnet because once the Alnico is magnetized, no additional energy is needed to maintain the system's state. The main advantage to this approach is that it can be utilized to achieve a continuum of bonding forces [71]. This approach could also potentially simplify the EP magnet assembly process because in practice it may be easier to wrap the energizing coil around a single component than the two different materials that we currently use.

The disadvantage to the Alnico-only EP magnet is the control mechanism. Instead of driving the Alnico magnet to the two extremes of its B-H hysteresis curve, the controller would need to be capable of driving the magnet to its origin where the remnant magnetic flux is zero. Typically, this requires multiple pulses of alternating polarity that slowly step the magnet's flux closer to zero. Not only will this approach require more precise control over the amount of energy delivered with each pulse, it may require a feedback mechanism as well. Finally, it is worth noting that this process of driving the Alnico's flux to zero will likely require more energy than the current approach of simply reversing its polarity.

Electrostatic connectors also pose an alternative to EP magnets. Electrostatic connectors function by applying a voltage between two insulated conducting plates. The opposite polarities attract and draw the plates together closing the air gap in between. The advantage of electrostatic connectors is that they are simple to fabricate and, like EP magnets, dissipate no static power (except what is need to compensate for any leakage current). Karagozler et al. have developed a millimeter-scale modular robotic system that attempts to use electrostatics for locomotion by plating aluminum electrodes onto a SiO_2 cylinder [47]. Despite their potential, electrostatic connectors have one

important drawback: they require high potential voltages and small air gaps to exert forces comparable to EP magnets of the same size. The voltages required can often be several hundred volts [47]. Knaian performed an analysis comparing electrostatic and EP magnet connectors assuming that the footprint and air gap of the two connection mechanisms was the same [51]. He determined that, for connectors ranging in size from hundreds of micrometers to many centimeters, the voltage required for the electrostatic connector to rival the holding force of the equivalent EP magnet would always exceed the breakdown voltage of the air gap. Consequently, for a given footprint area and air gap, the EP magnet connector will always be stronger than a electrostatic connector. This result does not mean that electrostatic connectors should be abandoned. They are mechanically robust, easy to fabricate, weight less than their EP magnet counterparts, and, ignoring the high voltages required, are easy to control.

There are also other, often biologically inspired, connection mechanisms that may be viable replacements for EP magnets as well. Most of these dry adhesive approaches take their inspiration from geckos that that climb on vertical and inverted surfaces. The connectors operate using van der Waals forces and generally aim to create patches of high surface area, fiber-like structures with a large degree of nano-scale compliance. This compliance results in a maximal amount of contact between the connector and an opposing surface thereby maximizing the number of inter-molecular interactions and the resulting bonding force.

Before we can employ these dry adhesives as inter-module connectors, we need to ensure that they can be switched on and off. In one approach, researchers fabricated nickel paddles whose faces were coated with polymer nanorods and whose movements were controlled using a magnetic field [79]. By default, the connector was active and the faces of the paddles were oriented parallel to some opposing bonding surface. Because the nanorods were sandwiched between the paddles and the bonding surface, the van der Waals forces were strong. The resulting pressure was 14Pa. When a magnetic field was applied, the paddles twisted about their long axes, turning their nanorod coated faces away from the bonding surface. In this configuration, the bonding pressure exerted was only 0.37Pa.

In another approach, researchers molded flexible sheet of microfibers and attached them to

a backing plate with variable stiffness [57]. To control the bonding force of their material, the researchers formed a bond between the fibers and the rounded tip of a glass rod while the backing plate was soft. Then they stiffened the backing plate. This resulted in a pull pressure that was much greater than what resulted when the backing was left in its flexible state.

3.9.2 Unit Module Fabrication

The current approach to modular fabrication involves wrapping a flexible printed circuit board around an investment casted brass frame. The flexible PCB is then manually held in place while solder connections are formed to hold the flex circuit to the frame. This approach is cumbersome, time consuming, and unlikely to scale well to smaller dimensions for many reasons.

We are currently pushing the resolution limits of investment casting. Other technologies for fabricating metal frames do exist. Whitesides et al. have demonstrated the assembly of 3D millimeter-scale metal trusses produced by folding 2D electroplated nickel forms [9]. Two-photon photopolymerization is way to make even smaller 3D structures that have nano-scale resolution. The process uses a precisely focused laser to polymerize, and thereby solidify, liquid monomer. The polymerization only occurs in a small volume where the laser intensity surpasses a non-linear threshold. The newest implementations can create incredibly intricate 3D structures [20]. Structures created using 2-photon photopolymerization can also be used as scaffolding and coated with substances like Parylene [59].

Neither the metallic or polymer-based frames are electrically active. We will need to attach connectors, communication, and processing components. One possibility is to continue wrapping an active, intelligent skin around the frames. Instead of being an polyamide-based flexible printed circuit, the skin could be formed from SiO_2 [47], SU-8 [99], or SiN [117], for example. The flexures between the faces could be fabricated from gold [99] or SiN [117].

Once the frame and skin have been fabricated, they must be assembled. Automated pick and place machines already handle 01005 surface mount components which are only 0.4mm by 0.2mm. The same automation technology could be applied to align the frames onto a panel of unfolded skins. Many options exist to automate the folding of the skins around the frames. Researchers

have used internal stresses [47], electron beams [128], surface tension [127], capillary forces [117], Lorentz forces [99], and even biological cells [58] to fold flat sheets into three-dimensional structures. Simple mechanical latches, electroplating, or laser welding could be used to permanently anchor the skin to each frame.

There are other alternatives to folding active skins around passive frames. Wood et al. have developed a "pop-up book" technology that enables the fabrication of complex three-dimensional objects from two-dimensional sheets [126]. The process relies on the precise alignment and bonding of many intricately cut layers of carbon fiber, adhesive, and polyamide. After bonding, the part is then singulated from the larger composite sheet and unfolded. The resulting three-dimensional structures can display exceptionally high aspect ratios, and they may integrate active electronic components [104].

Future Smart Sand modules may also be construct in three dimensions using layered processes. EFAB (electrochemcial fabrication) is a commercialized process developed over ten years ago that prints complex three-dimensional structures [21]. The basic process deposits thin (several μm) layers composed of metal surrounded by support material. Once the layers are complete, the support material is removed leaving just the bonded metal layers behind.

Other researchers have formed micro-scale structures by stacking thin silicon components using a compliant probe fabricated from PDMS [49]. As an example of their technology, the researchers were able to create an object resembling a hemisphere, 200μm in diameter, by stacking seven silicon rings with different diameters, each 10-50 μm thick. Once stacked, the pieces can be bonded with a high temperature annealing process.

With respect to future Smart Sand modules, the advantage of using silicon to fabricate the structure of each module is that the required circuitry and intelligence could be built directly into the structure using traditional IC fabrication processes. To enable the distribution of the circuitry over several silicon layers, and to route control signals to the actuators, we will need to utilize through silicon vias (TSVs)—metal connections that add electrical connections to the backside of a silicon die [106, 26].

Finally, there are new approaches to microfabrication that attempt to create monolithic 3D

spheres directly [116, 13]. Because these approaches do not build the spheres from layers, they may be quicker and and more precise from those that do. We also believe that spheres, because of their uniformity and lack of sharp edges, may replace cubes as the basic shape of the Smart Sand modules. The disadvantage to spheres is that they share minimal surface area with their neighbors, so the associated connection mechanism must have a high strength to area ratio.

While this section has not addressed all of the potential challenges associated with miniaturizing programmable matter modules, it has attempt to address the major components and processes that will be necessary to further shrink the Pebble modules. Careful analysis and many trade-offs will need to be made in order to select the best approach to future miniaturization. Along the way, unforeseen challenges are certain to arise. Ultimately, the most difficult challenge will be integrating all of the fabrication steps into a single, streamlined batch process.

Chapter 4

The Sandbox Simulator

We have developed the Sandbox simulator to test our high-level shape formation algorithms on scales larger than those afforded by our hardware platform. While we produced fifty Smart Pebbles, we want to ensure that the algorithms we developed run on much larger two- and three-dimensional systems. Before we begin explaining these algorithms, it is best to understand the Sandbox simulator because it was used extensively while developing, testing, and debugging the algorithms. Sandbox allows engineers and scientists to simulate distributed robotic systems more accurately and quickly than existing alternatives. It uses a realistic communication model grounded in hardware experiments. Using a network of 8 workstations, we are able to simulate a collection of more than 2,000 Smart Pebbles communicating and executing a variety of algorithms.

Sandbox is focused on simulating algorithms and communication in distributed robotic systems, and a block diagram of the system is shown in Figure 4-1. The simulator has a number of advantages over existing turn-key systems. First, it is scalable while maintaining its compelling performance. Each module runs as an independent process and communicates with other modules and the simulator framework using UDP and TCP packets. Second, the simulator is built with a modular architecture that supports distributed execution. The virtual modules in the system can be executed on any number of networked computing nodes, and the simulator visualization environment can be executed on yet another.

Third, Sandbox is generic and can be used with any modular robot system. It enables the user

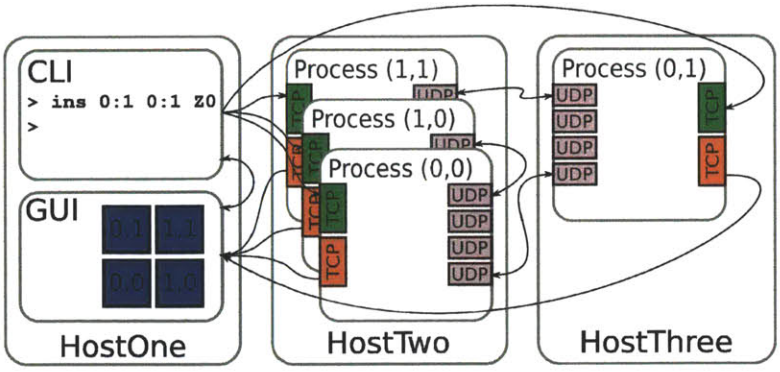


Figure 4-1: The Sandbox simulator system simulates each robotic module as a separate process. Instead of their native communication interfaces like IR LEDs/photodiodes, the modules communicate with their neighbors using UDP packets. As a result, the modules can be executed across multiple networked hosts allowing the simulation of huge ensembles of robots. There is a simple command line interface with which the user controls the system. A 3D GUI allows the user to view the topology of the robots in the system as well as observe the robot's internal state in real-time.

to reuse the vast majority of the code that comprises a robot's control software in both simulation and hardware. As shown in Figure 4-2, the process of adapting an existing modular robotic system to our simulator involves replacing the only the lowest-level physical communication layer with calls to Sandbox-specific functions that send UDP packets to a module's neighbors. In particular, Sandbox provides two basic communication functions: send a message to a neighbor and attempt to receive a message from a neighbor. All higher-level communication and application code can be reused regardless of language or complexity. This higher-level code is system-specific and includes the algorithms that give the modular robot system its unique abilities.

This light-weight approach of only replacing the physical hardware layer with simulator code guards against bugs that result in the normal porting process from simulation code to hardware code. This approach also ensures that it is easy to use Sandbox with almost any modular system. In addition to replacing the physical layer code with Sandbox-specific code, the user is free to instrument his higher-level code in order to convey internal state information from the robot to the simulator GUI. This instrumentation process is as simple as providing a list of variables to be monitored to an instrumentation thread that sends a TCP packet to the GUI whenever one of the variables changes state.

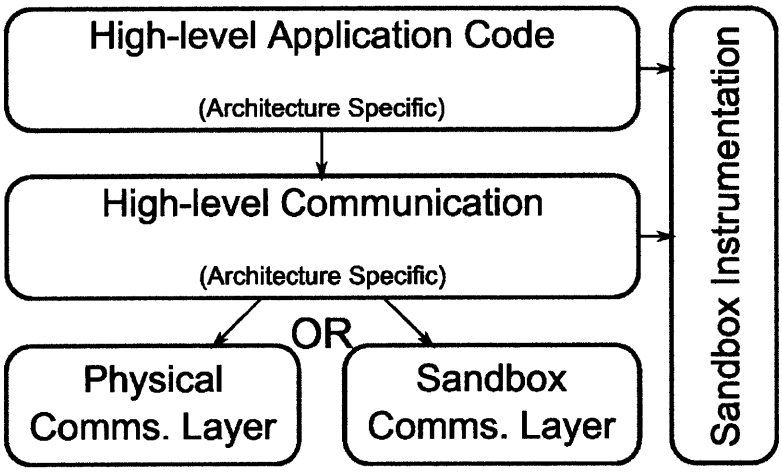


Figure 4-2: To use Sandbox with an arbitrary modular robot system, the user must replace the physical layer code with Sandbox-specific communication functions that allow a simulated module to communicate with its simulated neighbors. The user may also choose to instrument higher-level control code so that the module's internal state can be displayed in the simulator's GUI.

Sandbox excels over more established simulation systems such as Player/Stage/Gazebo or Webots when it comes to accurately simulating large collections of computationally impoverished modules with imperfect communication channels. These other systems are focused on simulating more traditional mobile robots in human environments. In comparison, Sandbox was designed for million-module ensembles of small robots that interact with the world in a radically different way than does a mobile robot. Sandbox models the unreliable nature of real-world communication links between neighboring robots. This allows Sandbox to better verify that theoretical algorithms will operate as expected in the presence of communication failures when implemented on tangible hardware that suffers from less than ideal neighbor-to-neighbor communication links.

Unlike many existing simulation systems, Sandbox system is not focused on simulating robot dynamics or the physics of the world in which the robot resides. We chose not to integrate this functionality into Sandbox for two reasons. Sandbox is primarily focused on simulating communication algorithms. Attempting to do more makes the simulator more complicated and raises the barrier to adoption. Additionally, many other simulators already do simulate module physics. These existing physics simulators could be layered on top of Sandbox to form a hybrid simulator. At the most basic level, the physics simulator would determine which modules are neighbors and

hence, which modules can communicate with each other. Sandbox could take this information and actually enable the low-level communication between the modules. Because the Sandbox system is more concerned with algorithms and communication, it can also be used to simulate sensor networks and other non-robotic distributed systems.

Modular and distributed robotic systems continue to grow in number and size. Advances in computation, fabrication, mass production technology accompanied by ever decreasing prices have enabled scientists and engineers to create distributed intelligent systems with ever growing number of nodes. In the Robot Pebbles system, in particular, we are interested in using hundreds of thousands or millions of modules to form objects through a process of self-disassembly. Before researchers commit to building larger and larger robotic systems, it is valuable to employ tools that allow for these systems to be thoroughly tested and debugged. Additionally, after deploying a large distributed system with anywhere from ten to millions of modules, researchers will benefit from tools that enable the evaluation of algorithmic changes while avoiding the need to modify the entire ensemble of robots to test each change. While the hardware is the ultimate testbed for our shape formation algorithms, we developed the simulator to make algorithm developing and testing faster, easier, and more reliable.

4.1 Simulator Design

The Pebbles have several unique features that we wish to capture in the simulator. First, the communication between Pebbles is probabilistic. Due to size constraints within the 12mm Pebbles, the circuitry controlling the EP magnets was designed such that only a single EP magnet can be active at a given time. While this does not impose noteworthy restrictions on a Pebble's ability to bond with its neighbors, it does affect how a Pebble communicates with its neighbors. A Pebble must divide its time between listening and transmitting messages on each of its faces. If a Pebble's neighbor is not listening at the same time that the Pebble is attempting to transmit, the message will not be received. Each Pebble executes a simple loop:

1. Listen on each face in random order for a fixed amount of time

2. Update state and queue messages for transmission

3. Attempt to transmit any pending messages

4. Repeat

The consequence of this loop is that a given Pebble is only listening for incoming messages on a given face less than a quarter of the time. In practice, this results in a message transmission success rate of approximately 25%. This mirrors the hardware behavior as illustrated by Table 5.3.

Second, the alignment between neighboring Pebbles is not perfect. The Pebbles are assembled manually by wrapping a flexible printed circuit around a brass frame. This results in slight non-uniformity between any two Pebbles. As a result, in a large ensemble, there are inevitably some neighboring Pebbles that cannot communicate. Additionally, it is possible for the EP magnets to break which also results in a Pebble that is unable to communicate with one of its neighbors. While the total percentage of broken links in a structure is small, it still important to simulate if we want to ensure the robustness of our algorithms.

Our goal in designing the Sandbox simulator is to capture these unique features of the Robot Pebbles system while ensuring that the simulator is extensible and easily adaptable to other distributed robotic systems. The simulator is controlled through an interactive command line interface with a basic set of commands:

- `ins` x y [z] *rotation* [*host*]
- `ins` $x_{min}:x_{max}$ $y_{min}:y_{max}$ [$z_{min}:z_{max}$] *rotation* [*host*]
- `rem` x y [z] *rotation* [*host*]
- `rem` $x_{min}:x_{max}$ $y_{min}:y_{max}$ [$z_{min}:z_{max}$]
- `rem all`
- `restart` $x_{min}:x_{max}$ $y_{min}:y_{max}$ [$z_{min}:z_{max}$]
- `restart all`

- `quit`

The `ins` commands insert new modules into the simulator, `rem` commands remove modules from the simulator, and `restart` commands restart a simulated module without changing its position, rotation or UID. The coordinates passed to the commands may specify a single point or a contiguous set of points. If the z-coordinates are omitted, the simulator assumes z-coordinate is 0. The *host* parameter may be omitted from the `ins` commands if the user wants the host on which the module is run to be assigned automatically.

The simulator's output is a topologically accurate, interactive visual representation of all modules created using OpenGL. The user can rotate, pan, and zoom his view of the system to get the necessary perspective. By hold the mouse cursor over any particular module, the user can see the module's internal state. More generally, the resulting simulator has a number important features: each robot runs as an independent process; the simulator front-end is separated from the robots being simulated; a single code-base can be used for both the hardware and simulated robots; and communication between robots is not idealized.

4.2 Process Distribution and Code Reuse

The simulator is designed to run the exact same robot control code as runs on the physical hardware. Unlike many other simulators, our approach does not depend on the user to maintain two distinct code bases: one for simulations (in Matlab for example) and another for the hardware (in C/C++ for example). Instead, we reuse the hardware code-base in the simulator by replacing the hardware abstraction layer with simulation-appropriate functions that replicate the hardware functionality in a realistic manner. The user switches between the simulation and hardware code by defining a single macro at compile time. We leave all of the high-level functionality untouched. With this virtual hardware abstraction layer in place, we recompile the code for the workstations used to perform the simulation. Then, to run the simulation, we use the workstation's underlying operating system to start a separate process for each module in the simulated system. Because each module runs as its own process, the simulator is a high-fidelity analogue for the hardware.

The operating system automatically manages task switching. Furthermore, in multi-core systems, modules can truly run in parallel with no additional effort from the designer.

In the Smart Pebbles system, the simulator-specific code used in place of the hardware abstraction layer handles three tasks: communication between modules, communication between a module and the user's PC, and flashing a LED. The inter-module communication scheme is explained in the next section. In short, it uses UDP packets to send messages between modules.

Simulating the communication between a module and the user's PC is accomplished with Linux pseudo-terminals. In the hardware system, each Pebble has a three-wire serial communication interface that it can uses to receive commands from the user or return state information. With the aid of a external microcontroller-based protocol converter, this interface appears as a serial port on the user's PC. Typically, the root module is the only module in an ensemble utilizing this interface. If the system is being used to form shapes through self-disassembly, the user sends an encoded description of the shape to be formed over a serial port, through the protocol converter, to the root module, where it is distributed to the remainder of the modules in the system. As already mentioned, to simulate this interface, the simulator employs a pseudo-terminal. In particular, the root module creates a new pseudo-terminal (e.g. `/dev/pts/0`) that can be opened for reading and writing just like a serial port (e.g. `/dev/ttyS0`).

Finally, the simulator code emulates the LED in each Pebble by replacing the LED with a binary state variable. Turning the LED on sets this variable, and turning it off resets it. The state of the LED, along with many other internal state variables, is transmitted to the simulator GUI over a TCP/IP connection whenever the state changes.

4.3 Communication

To accurately simulate communication between modules in a tangible system, the simulator uses UDP (user datagram protocol) packets sent over an IP connection. Both the source and destination of a UDP packet are specified by a address/port pair. An address is a typical IPv4 address such as `18.70.0.160` while a port is a 16-bit unsigned integer ranging from 0 to 65535. UDP, unlike TCP, is a connection-less protocol that does not involve handshaking or error correction. There is

no guarantee that a given UDP packet will be delivered to its destination, nor is there an automated method to determine if it has been. The sender must implement some higher level protocol to determine whether a UDP exchange is successful. Fortunately, this is a perfect model for many robot applications that use low-level communication devices like IR LED/photodiode pairs, simple radios, or in our particular case, electropermanent magnets. In such systems, the communication handshaking protocols are customized and built into the robot's application code. For example, the application code must modulate an IR LED and then wait for a specific response from its neighbor detected by photodiode to know that the message has been successfully received. UDP messages take the place of the LED and photodiode in the simulator.

In the Sandbox, each inter-module communication interface of each module is assigned a unique UDP port number. Through the use of command-line parameters and a online control port, each module knows the IP address and port number of the neighboring faces with which it is in contact. To successfully send a message to a neighbor, the transmitter must send a message and wait for a confirmation that it has been received. The receiver will only respond if it is actually listening to the port where the transmitter sent the message. If the receiver is not listening, the transmitted message will be lost.

The transmission algorithm in shown as Algorithm 1. First, and not shown in the listing, some higher-level function loads the message to be sent into a face-specific transmit buffer. Then, the transmitter performs a non-blocking write to its neighbor's address/port combination. The data written is a message sequence number prepended to the actual message in the transmit buffer. After the non-blocking write, the transmitter performs a number of non-blocking reads until all data has been flushed from the port's receive buffer. If any one of these reads contains an acknowledgment (ACK) message with a sequence number matching that of the original message, the message has been successfully received by the module's neighbor. The next time a message is transmitted, the sequence number will be incremented to differentiate the two messages if their contents happen to be identical. If a matching ACK message has not been received by the time the port's receive buffer has been emptied, it indicates that the neighboring module is not listening at the current time; the message to be sent will be left in the transmit buffer; and the sequence number will not

be incremented.

Algorithm 1 Inter-module Message Transmission

```
 1: txDatagram.sequenceNum = txSequenceNum
 2: txDatagram.payload = txBuffer
 3: setBlocking(socket, false)
 4: write(socket, txDatagram)
 5: repeat
 6:     rxDatagram = read(socket)
 7:     if rxDatagram.sequenceNum = txSequenceNum then
 8:         txSequenceNum++
 9:         txBuffer = ∅
10:         return true
11:     end if
12: until rxDatagram = ∅
13: return false
```

To receive a message from a neighbor, a module uses the approach shown in in Algorithm 2 which complements the transmit algorithm. To initiate the receive process, the module starts a timer and then performs a blocking read of the UDP port associated with a particular communication interface. If the timer expires before the read returns with a message, the read call will be interrupted and return without data. This indicates that the module's neighbor was not transmitting anything, and the read call is aborted. On the other hand, if the read returns data and the interface's receive buffer is empty, the received payload is moved into the receive buffer and a ACK message containing the sequence number of the received message is sent back to the transmitter. Realizing that there is no guarantee that the ACK message will arrive successfully, each incoming message with a sequence number that matches the sequence number of a message already in the receive buffer generates an additional ACK message.

Algorithm 2 Intermodule Message Reception

```
 1: success = false
 2: setBlocking(socket, true)
 3: startInterruptTimer()
 4: while interruptTimerNotExpired() do
 5:     rxDatagram = read(socket)
 6:     if rxBuffer = ∅ then
 7:         rxBuffer = rxDatagram.payload
 8:         rxBufSeqNum = rxDatagram.sequenceNum
 9:         success = true
10:     end if
11:     if rxDatagram.sequenceNum = rxBufSeqNum then
12:         write(socket,ACK)
13:     end if
14: end while
15: return success
```

The simulator also supports the ability to induce a given failure rate in communication links between neighboring modules in accordance with the *communication reliability* parameter. The user specifies this number as a percentage between 0 and 1. As each communication link is formed between a new process and its neighbors, the chance that is works is governed by the communication reliability. To disable a particular link, the simulator connects to the TCP/IP-based control port of the module with the newly "broken" communication link and informs the module that it may not communicate with a particular neighbor. This process is handled in the simulation-specific code so that the application code can treat the link normally.

4.4 Extensibility

The simulator is designed to be easy to distribute across multiple machines. Because each module is simulated as a stand-alone process that communicates with its neighbors over a standard IPv4 network, the simulated modules can be located on any machine that is connected to the network. If the user wants to simulate a large system, it is easy to leverage additional workstations. Our simulator has the additional advantage that it does not rely on any special libraries to function. Any standard Linux- or UNIX-based OS should be capable of acting as a node in the simulator network. This ensures that it is easy for non-privileged users to deploy our system.

The computation nodes used by the simulator to execute the module code are specified at runtime using a flat text file. This host file contains lines that specify a hostname, maximum number of simulated processes to be run on that host, and a nickname for the host. During initialization, the simulator reads this file once and caches the information. The maximum number of processes allowed to run on a given computation node is used when randomly assigning simulated modules to nodes.

4.5 Front-end and Simulated Robot Separation

The Sandbox system uses a central GUI running on a single host machine to display the topology of the devices being simulated, the internal state of each module, and the messages flowing between

modules. This GUI is written using the Qt application framework, and it uses OpenGL to display a 3D model of the topology. To receive internal state and message buffer information from each simulated module, the GUI opens a TCP/IP port that all of the simulated modules can connect to and communicate with. Each time the internal state of one of simulated modules changes, or each time a module receives a message, the module sends a TCP packet to the GUI over the underlying IP network. The GUI parses these messages and displays their contents in an informative manner as shown in Figure 4-3.

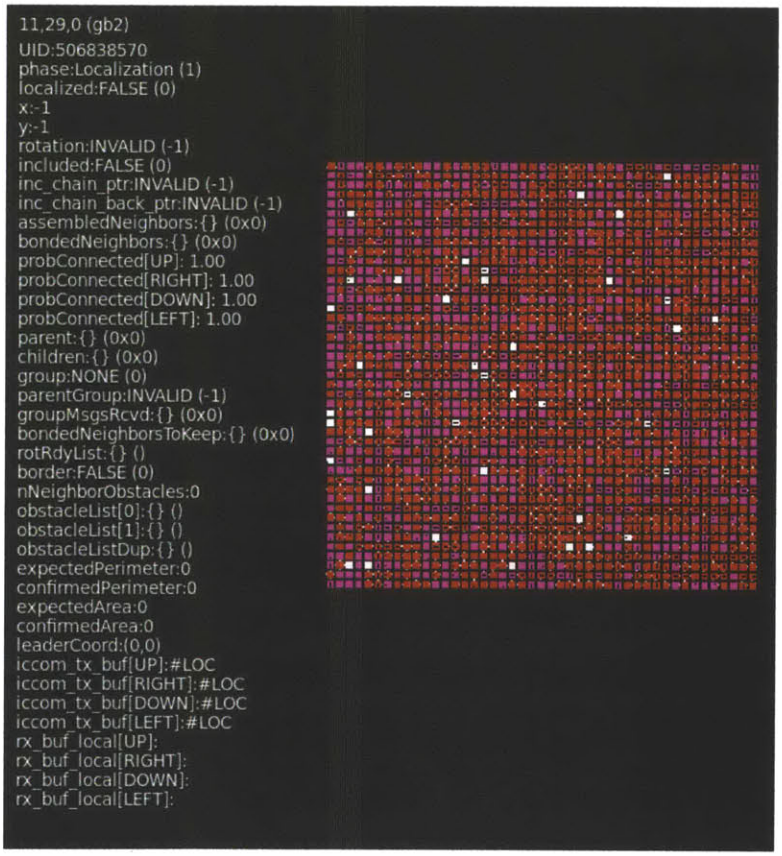


Figure 4-3: The Sandbox GUI helps the user visualize the topology of the robot modules being simulated, and it displays internal state information from each module as the mouse moves over it.

By separating the GUI from the actual modules being simulated, we achieve several benefits. First, it allows the aforementioned extensibility. If the GUI and all modules were a simulated using a single program, it would be nearly impossible to extend the simulator system across multiple

machines. Second, we can restart the GUI and the simulated modules separately. This feature is useful when one wants to modify the code running on the simulated modules without redoing the set-up of a particular experiment. One simply recompiles the Pebble code and then, within the simulator GUI, issues a `restart all` command. This restart commands kills all currently running processes, but keeps track of each process's physical parameters: location, rotation, and UID, for example. Using this information, Sandbox then reruns each process to recreate the same physical arrangement of simulator modules.

4.6 Experiments

To test the performance of the Sandbox system, we have completed a number of experiments on a set of six virtualized Linux machines spread across six distinct physical machines. Each physical machine was hosting 7 other unrelated virtual machines. Each of the virtual machines was given exclusive access to a single 64-bit processor running at 1.86GHz with 1MB of L2 cache. Additionally, each virtual machine was assigned 2GB of dedicated RAM and shared access (along with the 7 other virtual machines) to a 1Gb/sec Ethernet connection.

To test the speed of the Sandbox system, we performed experiments with the localization algorithm that we use as part of the self-disassembly process in the Smart Pebbles system. The goal of the algorithm is to inform every module in the ensemble of its location relative to a root module whose position we assign arbitrarily to be (0,0). The simplicity of the algorithm combined with the fact that it requires a small degree of local computation and a large amount of inter-module communication make it a good candidate for characterizing the Sandbox system.

The algorithm itself is as follows: once a module knows its position, it sends a position message on to its neighbors so that they can determine their positions. Each position message contains the transmitter's location and rotation. The receiving module, knowing on what face the position message was received, can determine its own position and orientation. Modules continue to send and receive position messages until all modules in the structure know their position.

We characterized the running time of the localization algorithm in both chains of modules and square blocks. Figure 4-4 shows the running time for the localization algorithm when the topology

of the Smart Pebbles is a n-unit chain. As the figure shows, we characterized two different ways to distributed the modules. The default was to run all modules on a single workstation. Alternatively, we randomly assigned modules to one of the six workstations. Figure 4-4 shows that there was a slight performance benefit when distributing modules across workstations. This makes sense given that each computational node needs to do less work. If communication between nodes were slower, this benefit might be lost.

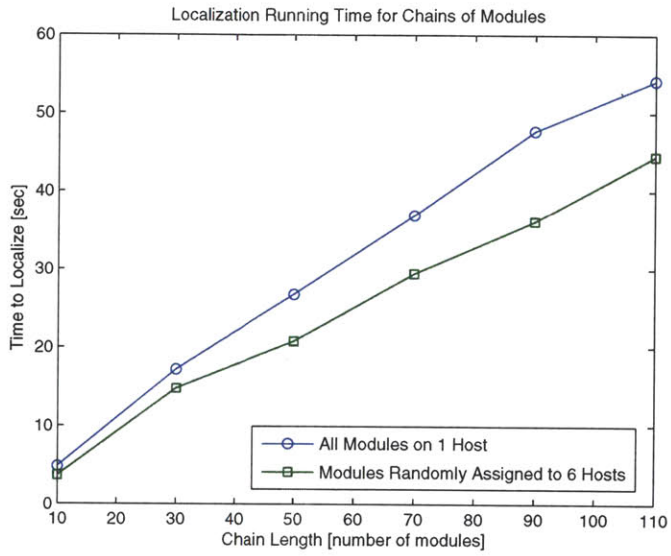


Figure 4-4: When executing the localization algorithm on chains of modules, the running time scales linearly with the length of the chain. Additionally, we see a benefit in using multiple computation nodes, instead of a single node, when running the simulator.

We also characterized the running time of the localization algorithm on square topologies. The results of these experiments are shown in Figure 4-5. We found that a single computational node became noticeably sluggish when attempting to run more than 400 modules. This is why we did not simulate a square with side length greater than 20 using only a single computational node. We observed this same latency when simulating squares with side length greater than 45 while using all six computational nodes. Interestingly, when running the localization algorithm on small, (less than 400 module), squares of simulated Smart Pebble modules, we see no difference in running time as we switch from running all modules on a single node to randomly distributing the modules

across six computational nodes.

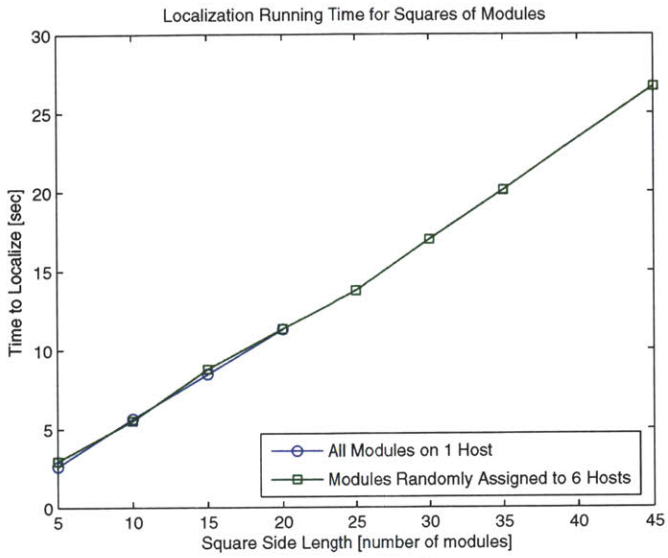


Figure 4-5: The time for the localization algorithm to complete on a square of modules scales linearly with the side length of the square. Surprisingly, the running time does not appear to depend on whether all modules are running on a single computational node or distributed across six.

We have also repeatedly run the localization algorithm on the physical Smart Pebble and Miche hardware. In hardware, we see a similar linear running times, so we conclude that the simulator is operating correctly. Consult Chapter 6 for many additional plots which compare running times in hardware and simulation.

Chapter 5

Low Level Communication

Nearest-neighbor communication between Robot Pebble nodes forms the basis of all the higher-level algorithms that drive the Robot Pebbles system. Using nearest-neighbor communication, the nodes are capable of localizing, routing messages from one arbitrary point to another, and duplicating passive objects that are surrounded by active modules. We have developed a robust low-level communication system that ensures that messages are delivered reliably and correctly.

There are two major challenges associated with inter-module communication in the Pebbles system. First, due to space constraints, we have a minimalist hardware approach. There is not enough area available on the flexible PCBs that form the modules for more than a few communication-specific electronic components. Second, the software resources available for communication are severely limited. The processors run at 8MHz, have 32KB of program memory, and have 2KB of RAM.

As explained in Section 3.5, each node is only capable of receiving or transmitting on a given face at any time. As a result, the modules must divide their time between listening for incoming messages and transmitting outgoing messages on all of their faces. If a module is attempting to transmit a message to its neighbor, there is no guarantee that its neighbor will be listening for an incoming message on the corresponding face. Furthermore, the receiving module may begin listening for an incoming message on the given face part way through the transmitter's attempt. Alternatively, a module actively transmitting a message may experience interference from its in-

tended recipient if that recipient begins to transmit its own, unrelated message back to the original module.

The communication channel is noisy. As seen in Figure 3-12, the noise margin between high and low bits is roughly 800mV. If there is misalignment between neighboring modules, the noise margin will decrease. In addition, there is significant cross talk between modules. Bits from a neighboring module to which the receiving module is not listening come within 100mV of the bit detection threshold.

To handle these challenges we have created a packet-based communication protocol that minimizes data corruption while ensuring acceptable communication through-put. Packets are protected with a sequence number, length field, and checksum. Each module randomly divides its time between listening for incoming messages on all four faces. A module does not repeat listening on a given face until it has also listened for incoming messages on all other faces. The modules are also capable of detecting severed communication channels and channels that have been re-established. Finally, to manage the messages flowing between neighboring modules, we have created a set of transmit and receive buffers. We have performed numerous experiments to demonstrate the results of our approach.

5.1 Message Buffers

The high-level application code interfaces to the low-level EP magnet communication routines with a set of transmit, receive, and payload buffers. There is one set of buffers for each face, an additional set for each module's SPI-based interface to the external world, and a final set that is used internally. These buffers are illustrated in Figure 5-1.

To send an application-level message to a neighbor, the module checks whether the transmit buffer is currently empty, and if it is, loads the new message into the buffer using the `fillTxBuf` function shown as Algorithm 7. Once the transmit buffers are loaded, the high-level application code must call the `transmitMessage` function (Algorithm 6) to prompt the low-level communication interface to attempt to send the queued messages.

Whenever the low-level communication interface is not sending messages, it is listening for

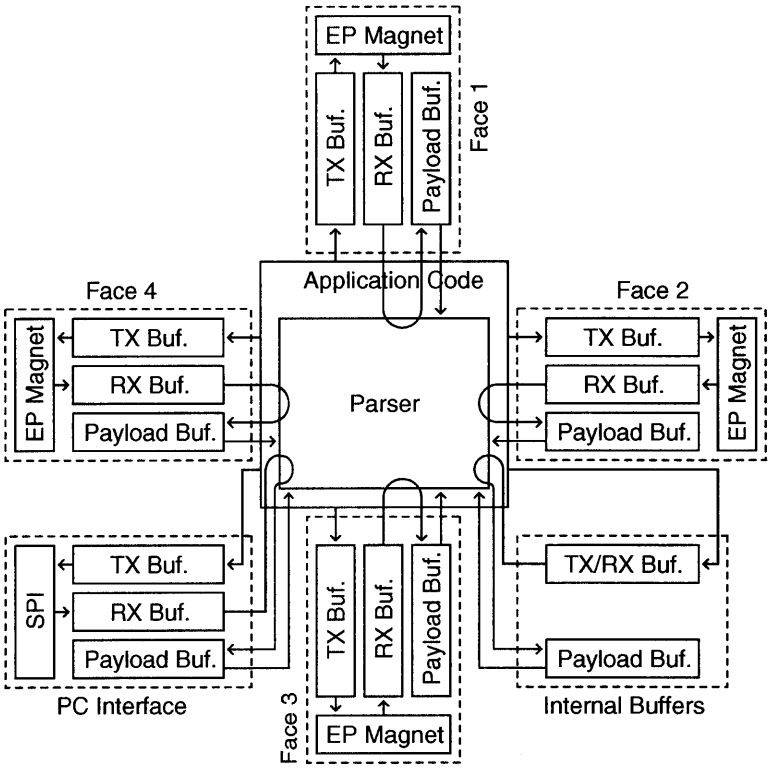


Figure 5-1: The Robot Pebbles use transmit and receive buffers as an interface between the high-level application code and the low-level communication interface. The arrows indicate the direction in which data flows.

incoming messages. When the module successfully receives a new message, (which it is only allowed to do if the face-specific receive buffer is empty), it places the new message into the receive buffer. From there, the high-level application code can access and parse the message.

In Section 5.4 we explain how the system employs routing messages as an abstraction layer that facilitates the delivery of messages to nodes at arbitrary coordinates, not just a node's nearest neighbors. Routing messages carry a payload message that the receiving module needs to parse after determining that it is the intended recipient of the routing message. To facilitate this process, each face also has a payload message buffer. The payload of a routing message is transferred to this buffer when the routing message reaches its destination.

The reason the payload of a routing message is copied to the payload buffer for parsing is that some routing messages are marked *public*. This means that as they propagate to their destination, every module through which they pass parses the routing message's payload. The payload buffer provides a location from which to parse a public payload without overwriting the message in the receive buffer. If instead we wrote a routing message's payload back to the receive buffer, over the routing message itself, we would effectively halt the routing message's journey to its specified destination. Algorithm 3 demonstrates this process.

Algorithm 3 routeParse(*rxFace*, *msg*)—high-level parsing function for received routing (ROT) messages showing when a ROT message's payload is moved to the receiving face's payload buffer

```
Require: 1 ≤ rxFace ≤ 4
Require: msg: message in receive buffer being parsed
 1: if msg.destination = getThisLocation() then
 2:     if isPayloadBufEmpty(rxFace) then
 3:         moveToPayloadBuf(rxFace, msg.payload)
 4:         emptyRxBuf(rxFace);
 5:     end if
 6:     return
 7: else if msg.public = true then
 8:     if isPayloadBufEmpty(rxFace) then
 9:         moveToPayloadBuf(rxFace, msg.payload)
10:     else
11:         return
12:     end if
13: end if
14: forwardROT(msg)
15: return
```

In addition to the buffers associated with each face of a module, there is an internal set of buffers. The internal buffers are used when a module wishes to initiate a routing message. So that

the high-level application does not need to decide on which face to start propagating the routing message, it can write the routing message to the module's internal transmit/receive buffer. The first time the message is parsed, the routing message handler will move the routing message to the correct transmit buffer so that it departs the module in the correct direction.

Finally, there is a set of buffers dedicated to communication with the external world using the module's serial peripheral interface (SPI). This interface is connected to a set of contacts on the bottom face of each module. (See Section 3.7.) These contacts interface with a set of pogo pins in a test fixture. In most usage scenarios, only a single module in an ensemble is connected to the test fixture, so the SPI transmit and receive buffers are typically unused.

5.2 Packet Format

Inter-node communication is governed by the simple two-way protocol based on 1μs pulses inductively coupled between the EP magnet coils of neighboring modules. (See Figure 3-12 for the waveforms.) To send a *space* (low-level), nothing is sent during the bit period. As a result, the output of the processor's internal analog comparator (see Figure 3-8) remains low. To send a *mark* (high-level) bit, a 1μs pulse is sent during the bit period which pulls the inverting input of the comparator low for a short period. As a result, the comparator's output goes high momentarily.

Bits are sent using on-off keying at 9600bps. We chose 9600bps because it is a standard baud rate, and it allows the 8MHz processor enough time between bits to perform other tasks. While the ATMega328 inside of each module does have a hardware UART, the fact that the mark pulses only occupy 1μs of the entire 104μs bit period, prevents us from using it. Instead, all UART functionality is implemented in software using interrupts.

Figure 5-2 illustrates the structure of each byte. In particular, each 8-bit data byte is preceded by a start bit (always a mark) and followed by a parity bit (mark or space) and two stop bits (always space bits). Each byte is sent least significant bit (LSB) first. All inter-node data is constrained to be ASCII strings with character values ranging from 0—127. This leaves one bit of each byte unused for data. The unused bit, when a mark, signifies that the byte is a synchronization byte. For all other bytes, the bit is a space. Because we want the receiver to be able to identify a synchronization

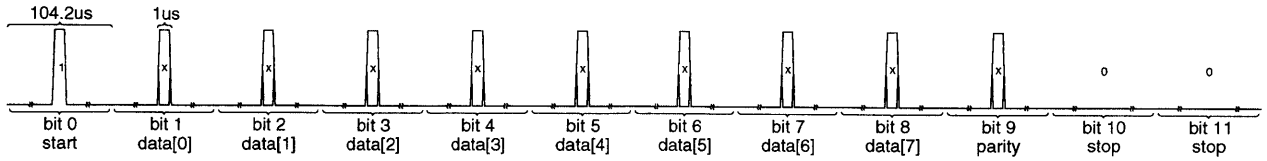


Figure 5-2: Each byte exchanged by neighboring Pebbles begins with a start bit that is a "1" thereby driving current through EP magnet coils. The start bit is followed by eight data bits, a parity bit, and two stop bits.

byte as early in its reception as possible, we left-shift the seven ASCII character data bits and use the LSB as the synchronization identifier.

The exchange process is a bidirectional protocol. We term the module sending the application-level message to its neighbor the *master* and the module receiving the application-level message the *slave*. The slave still transmits some bytes back to the master to confirm that it is listening and that it has received the message successfully. This exchange is illustrated in Figure 5-3. Algorithm 4 illustrates the transmitter's algorithm and Algorithm 5 the receiver's.

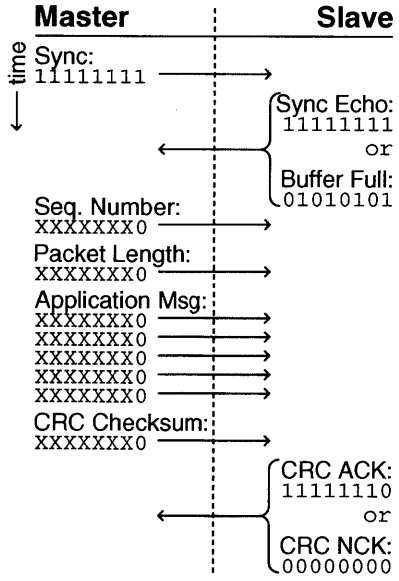


Figure 5-3: The master module transmits an application-level message to its neighbor using a bidirectional exchange that attempts to ensure robust communication in spite of a noisy, unreliable communication channel.

The master initiates the data exchange process in Line 6 of Algorithm 4 by sending a synchro-

Algorithm 4 epMagTransmit(*txFace*, *msg*)—called to transmit an inter-module message to a particular neighbor

Require: $1 \leq txFace \leq 4$
Require: *msg*: message to be sent
1: **if** $txBuf[txFace].full =$ **false then**
2: **return** NULL-STRING
3: **end if**
4: $txPacketLength \leftarrow$ stringLength($txBuf[txFace].msg$)+3
5: $txChecksum \leftarrow$ computeCRC($txSeqNum, txPacketLength, txBuf[txFace].msg$)
{Send the synchronization byte and wait for the slave to echo it}
6: transmitByte($0xFF$)
7: $byteCnt \leftarrow 1$
8: startNextByteTimer()
9: **while** isNextByteTimerExpired() = **false do**
10: $rxByte \leftarrow$ receiveByte()
11: **if** $rxByte = \emptyset$ **then**
12: **continue**
13: **else if** $rxByte = 0xFF$ **then**
14: transmitByte($txBuf[txFace].seqNum$)
15: $byteCnt \leftarrow byteCnt + 1$
16: **break**
17: **else if** $rxByte = 0x55$ **then**
18: **return** RECEIVER-FULL
19: **else**
20: **return** INCORRECT-RESPONSE
21: **end if**
22: **end while**
23: **if** isNexByteTimerExpired() = **true then**
24: **return** NO-RESPONSE
25: **end if**
{Send the body of the packet}
26: **while** $byteCnt \leq txPacketLength$ **do**
27: **if** $byteCnt = 2$ **then**
28: transmitByte($txPacketLength$)
29: **else if** $3 \leq byteCnt < txPacketLength$ **then**
30: transmitByte($txBuf[txFace].msg[byteCnt]$)
31: **else if** $byteCnt = txPacketLength$ **then**
32: transmitByte($txChecksum$)
33: **end if**
34: $byteCnt \leftarrow byteCnt + 1$
35: **end while**
{Wait for the slave to acknowledge that the checksum matches }
36: startNextByteTimer()
37: **while** isNextByteTimerExpired() = **false do**
38: $rxByte \leftarrow$ receiveByte()
39: **if** $rxByte = \emptyset$ **then**
40: **continue**
41: **else if** $rxByte = 0xF7$ **then**
42: $txBuf[txFace].full \leftarrow$ **false**
43: **return** SUCCESS
44: **else**
45: **return** CHECKSUM-MISMATCH
46: **end if**
47: **end while**
48: **return** CHECKSUM-TIMEOUT

Algorithm 5 epMagReceive(*rxFace*)—called to receive an inter-module message from a particular neighbor

Require: $1 \leq rxFace \leq 4$

```
 1: startListenTimer()
 2: while isListenTimerExpired() = false do
 3:     rxByte ← receiveByte()
 4:     if rxByte = ∅ then
 5:         continue
 6:     else if rxByte = 0xFF then
 7:         if rxBuf[rxFace].full = false then
 8:             transmitByte(0xFF)
 9:             break
10:         else
11:             transmitByte(0x55)
12:             return
13:         end if
14:     end if
15: end while
16: byteCnt ← 0
17: startNextByteTimer()
18: while isNextByteTimerExpired() = false do
19:     rxByte ← receiveByte()
20:     if rxByte = ∅ then
21:         continue
22:     end if
23:     byteCnt ← byteCnt + 1
24:     if byteCnt = 1 then
25:         rxSeqNum ← rxByte
26:     else if byteCnt = 2 then
27:         rxPacketLength ← rxByte
28:     else if 3 ≤ byteCnt < rxPacketLength then
29:         rxBuf[rxFace].msg[byteCnt] ← rxByte
30:     else if byteCnt = rxPacketLength then
31:         rxChecksum ← rxByte
32:         if computeCRC(rxSeqNum, rxPacketLength, rxBuf[rxFace].msg) = rxChecksum then
33:             transmitByte(0x7F)
34:             if rxSeqNum ≠ rxBuf[rxFace].seqNum then
35:                 rxBuf[rxFace].full ← true
36:                 rxBuf[rxFace].seqNum ← rxSeqNum
37:             end if
38:             return
39:         else
40:             transmitByte(0x00)
41:             return
42:         end if
43:     end if
44:     restartNextByteTimer()
45: end while
46: return
```

nization byte of all ones (marks). When the slave senses that the LSB is a 1, it starts a timer that counts the bit period of each of the 7 remaining bits. It then averages the seven bit periods to determine the rate at which the transmitter is operating. It adjusts its receiving and transmitting frequencies accordingly. This synchronization is necessary because each node is clocked by an RC oscillator whose frequency differs slightly from node to node. The slave node will continue to use the sensed frequency until it begins to listen for incoming data on a different face.

After sending the synchronization byte, the master switches to listening for a confirmation from the slave in Lines 8–25 of Algorithm 4. Assuming that the slave's receive buffer is empty, it echos the synchronization byte back to the master in Line 8 of Algorithm 5. If instead, the slave's buffer is full, it echos 0x55 back to the master so that the master knows that the slave is alive, but unable to accept packets (Line 11 of Algorithm 5). Alternatively, the slave may not be listening for the master's transmission. In this case, the master receives nothing from the slave. After a timeout period, the master decides that the slave is not listening, declares its transmission attempt a failure, and returns to an idle state (Lines 23–25 of Algorithm 4).

Having received the echoed synchronization byte, the master begins to send data to the slave. The first byte sent is a sequence number (Line 14 of Algorithm 4). The sequence number ensures that the slave does not receive duplicate copies of a single message. In particular, it guards against the case in which the master thinks that a transmission has failed when, in fact, the slave thinks that it was successful. This scenario arises when the slave verifies the message's checksum, but the CRC acknowledgment byte does not reach the master. If the master does not see the CRC acknowledgment from the slave, it will attempt to re-transmit the message. Without the sequence number, the master would think that the message had only been sent once, while the slave would think that the master had intentionally sent two copies of the same message.

There is a unique sequence number associated with both the transmit and receive buffers of every face. It is used as shown in Table 5.1. Each time the master wishes to send a new message on a specific face, it increments the sequence number associated with that face (Events 2 and 10 in the table). When the slave processes an incoming message, it compares that message's sequence number with the sequence number of the last received message. If the sequence number of the

incoming message matches the sequence number of the last received message (Events 6 and 8), the slave sends a confirmation back to the transmitter, but internally, it treats the message as a duplicate and does not move it to its receive buffer. In contrast, if the incoming sequence number is different from the sequence number of the last received message (Events 3 and 11), the slave still sends a confirmation, but it also treats the message as a new, unique message. This approach ensures that the slave only processes a single instance of a message with a given sequence number.

Following the sequence number, the master sends a byte containing the length of the packet (Lines 27–28 of Algorithm 4), including the sequence number, length byte itself, application-level payload message, and CRC checksum byte. The length byte allows the slave to determine whether the correct number of bytes was received. It also allows the slave to set a time-out so that if the correct number of bytes is not received within a fixed time period, the slave can assume that there was an error in the communication process. In the event of a timeout, the slave returns to the idle state.

Following the length byte, the master sends the application-level message payload (Lines 29–30 of Algorithm 4). The payload is a null-terminated ASCII string, but the null terminator is not sent nor included in the previous length calculation. This message payload is what is seen by the higher-level algorithms that control the system's shape formation properties. The format of these payload messages is described in Section 5.4.

After the master has sent the entirety of the application-level message payload, it sends a CRC checksum [52] (Lines 31–32 of Algorithm 4). Because the system reserves the LSB of every byte as the synchronization identifier, the checksum is only seven bits long. The CRC polynomial used to compute the checksum is 0x5B. The CRC is computed over all bytes, including the sequence number and length byte, sent, not just the ASCII message.

Upon receiving the CRC byte, (which it recognizes based on the already transmitted message length), the slave compares the received byte with its own computation of the CRC based on previous bytes in the packet. If the received CRC matches the slave's locally computed CRC, the slave decides that the message is valid (Lines 32–38 of Algorithm 5). In response, the slave sends a CRC acknowledgment (0xF7) to the master. It also stores the received sequence number for

Table 5.1: Every time the master loads a new packet into its transmit buffer, it increments the associated sequence number. The slave sends a checksum acknowledgment back to the master each time the slave receives a packet, but the slave only moves the received packet into its receive buffer for high-level parsing when the incoming sequence number differs from the sequence number already attached to its receive buffer.

Event #	*Event Description*	*Master Tx. Buf.*	*Master Seq. Num.*	*Slave Rx. Buf.*	*Slave Seq. Num.*
1	Master and slave idle	Empty	17	Empty	17
2	Master loads new packet into transmit buffer	Full	18	Empty	17
3	Master transmits, and slave receives, packet	Full	18	Full	18
4	Slave transmits, but master does not receive, checksum acknowledgment	Full	18	Full	18
5	Slave parses message in its receive buffer	Full	18	Empty	18
6	Master re-transmits, and slave receives, packet	Full	18	Empty	18
7	Slave transmits, but master does not receive, checksum acknowledgment	Full	18	Empty	18
8	Master re-transmits, and slave receives, packet	Full	18	Empty	18
9	Slave transmits, and master receives, checksum acknowledgment	Empty	18	Empty	18
10	Master loads new packet into transmit buffer	Full	19	Empty	18
11	Master transmits, and slave receives, packet	Full	19	Full	19
12	Slave transmits, and master receives, checksum acknowledgment	Empty	19	Full	19
13	Slave parses message in its receive buffer	Empty	19	Empty	19

comparison to future messages. If the received sequence number was different than the previously stored number, it moves the received message to a face-specific receive buffer and sets the buffer full flag. At this point, the module's application-level algorithms may parse the message.

If the received and computed CRC bytes do not match, the slave sends a not acknowledgment (0x00) to the master instead (Line 40 of Algorithm 5). The slave does not update its sequence number or move the message to the face-specific receive buffer. The application-level algorithms never know about the failure.

Meanwhile, when the master receives a CRC acknowledgment from the slave, it knows that the message has been successfully transmitted (Lines 41–43 of Algorithm 4). As a result, it marks the face-specific buffer from which the message was transmitted as empty so that another message can be loaded by the application code for transmission. If the master receives a CRC not acknowledge byte, or nothing at all after transmitting its CRC checksum, it assumes the transmission has failed and leaves the transmit buffer unmodified (Lines 45 and 48 of Algorithm 4).

5.3 Packet-Level Experiments

To ensure that packet-level communication algorithms work correctly, we logged the exchange of over 30,000 inter-module messages. We aimed to determine how quickly and reliably a group of modules is able to communicate. In each case, we ran a series of four related experiments. In each experiment, one, two, three, or four transmitting modules were mated to a central receiving module. Each transmitting module attempted to send a string of messages consisting of increasing numbers: "1", "2", "3", etc. The transmitting modules were attempting send these messages on all of their faces (i.e. "3" was transmitted from all faces before transmitting "4"). If the receiving module did not respond to a transmitting module's attempt to transmit, the transmitting module progressed to the next number. The receiving module was connected to a power source and also shared a serial communication link with a desktop PC running a terminal program. The receiving module's only task was to listen for incoming messages on each of its four faces and relay these messages to the desktop computer. Table 5.2 summarizes the results of our communication speed test. In each case, we measured how many messages were received in the first 60 seconds after all

modules were energized.

Table 5.2: The inter-module message exchange rate is roughly linearly related to the number of neighboring modules transmitting messages.

# Transmitters	*Rate [msg/sec]*	*Rate per Face [msg/sec]*
1	10.4	10.4
2	20.5	10.3
3	39.3	13.1
4	50.9	12.7

The communication speed test shows that the message reception rate is, in the worst case, 10 messages per second, but grows in proportion to the number of transmitters. This is not surprising given that the receiver listens for incoming messages on each face for a set amount of time before proceeding to listen on the next face. In the event that the receiver does receive a message while listening to a specific face, it immediately advances to listening on the next face. In the experiments summarized in Table 5.2, the receiver was programmed to linger and listen on each face for 25ms, but the messages being transmitted were roughly half this length. (Given our experience with the Miche system [34], we expect the average message employed the the disassembly algorithms to be 15 characters in length and therefore require 12.5ms to transmit.) If the receiver receives a message each time it listens to each face, it will be able to progress through its tour of all four faces more quickly. This explains why the per-face message reception rate was greatest when the receiver had three of four neighbors.

To test how reliably neighboring modules were able to communicate, we performed two experiments. The first was designed to test the reliability of the communication channel; the receiver listened for incoming messages on only one face. We allowed the single transmitter to send over 10,000 messages. Not a single message was lost or received incorrectly. We conclude that the inter-module communication channel is quite robust when a module is only communicating with a single neighbor. In the second experiment, the receiver divided its time by listening for incoming messages on all four faces. We measured both the fraction of messages received as well as the number of attempts each transmitting module made before it was successful. Table 5.3 shows what percentage of transmitted messages were received and passed to the PC.

Table 5.3: The percentage of messages received by a module with multiple transmitting neighbors increases with the number of neighbors.

# Transmitters	*% Messages Received*
1	25.0
2	25.0
3	26.4
4	30.2

The results for the second experiment show that percentage of messages received never exceeds 30%. This is due to the fact that the receiving module is only listening for incoming messages on any given face 25% of the time. For every time slot during which the transmitter and receiver synchronize and exchange a message, there are three other periods when the transmitter fails to send its message because the receiver is not listening. The transmitter records each of these failures and does not attempt to re-transmit any message. (This approach is only used when characterizing the communication algorithms.) When running application code, the transmitter will make multiple attempt to re-transmit any message that is not sent successfully on the first attempt. This is detailed below.

Also note that as the number transmitters is increased, the percentage of messages that are received increases. This trend is due to the fact that the receiver is able to cycle through listening for incoming messages on all faces more quickly when it is actually receiving messages. Once it receives a message on a given face, the receiver immediately moves to listening for an incoming message on the next face. If the time to exchange a message is shorter than the duration the receiver normally lingers listening on each face, the receiver will be able to cycle through its faces more quickly when it receives a message on each face.

Finally, we tabulated the number of unsuccessful transmission attempt made by each of the transmitting modules before a successful transmission. We allow each transmitter to send messages for 60 seconds. During this 60 seconds, each transmitter attempted to send between 8,000 and 12,000 messages. The results are displayed in Figure 5-4 which shows, once again, that four neighboring transmitting modules leads to fewer dropped messages than just one or two transmitting modules. Regardless of the number of transmitting neighbors, the percentage of the time a

transmitter unsuccessfully attempted to communicate with the receiver before success was rarely more than three attempts. If the transmitters were programmed to retry sending each message until successful, a transmitter would, on average, succeed within 4 attempts 98.5% of the time. By the time a transmitter has made 7 attempts, it is virtually guaranteed to have sent its message successfully.

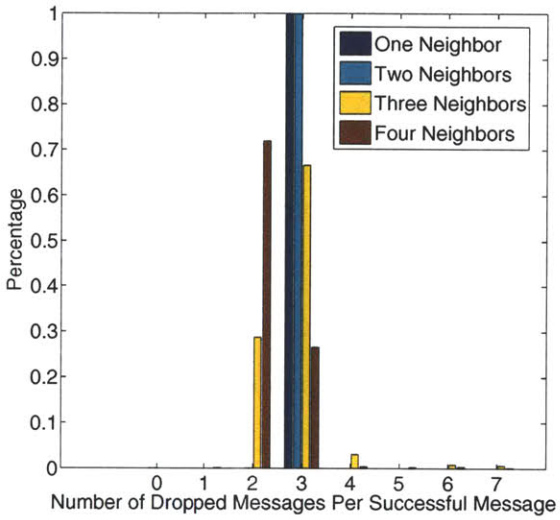


Figure 5-4: This plot shows the number of unsuccessful attempts a transmitter typically makes before it successfully sends a message. Regardless of the number of transmitting neighbors a receiver is communicating with, 98.5% of messages are send successfully within four attempts.

5.4 Application Message Format

The application-level message payload contained in each packet follows a simple format. In particular, messages are composed of printable ASCII characters from 0—127. By choosing to use text-based messages, we greatly simplify the debugging process. It is easy to quickly interpret a list of messages exchanged between modules, and it is possible to construct messages that the user wants to inject into the system through a particular module. The downside to ASCII messages is that they consume unnecessary space. An integer that would only require one byte to represent in binary, may require up to three ASCII bytes. The other disadvantage to ASCII messages is their

variable length. That same one-byte integer may only need a single ASCII byte to represent it. We cannot simply assume that a given byte position within an ASCII message always holds the same field. As a result, using ASCII messages dictates that we have some method to differentiate between neighboring data fields within a message.

Each message begins with a type field. The type field can be any length and contain almost any printable character. In our code, most messages used three or four character type fields. We chose the character combinations to reflect the purpose of the message. For example, we picked "LOC" for a localization message and "DIS" for a disassembly message. When a module receives a messages from its neighbor, the first thing it does it compare the message's type field against a list of known message types. When it finds a match, it sends the message to the parsing function that is specific to that particular type of message.

To signify the end of the type field, we use a field separator. We have chosen a comma as the universal field separator. After this first comma, is the first data field. Messages can have an arbitrary number of data fields, or none at all. All data fields are also separated by commas. The number and contents of the data fields are message specific. For example, a localization (LOC) message has no data fields. It is sent by a module wishing to localize itself. The receiver, if it can help the sender localize, responds with a position (POS) message that contains its coordinates, rotation, and transmitting face. A sample position message:

POS,1,0,0,3

The one exception to using the comma as a field separator arises with routing messages to arbitrary destinations. Routing messages transport another message to a given destination coordinate that is not necessarily the sender's immediate neighbor. For example, a routing message could be used to send a disassembly (DIS) message from coordinate $(2,1)$ to $(4,4)$. The message embedded in and being transported by a routing message comprises the last field of the routing message. That is, it is appended to the end of the routing message. To differentiate the embedded message being routed from the routing message itself, we use a semicolon. This is the only instance where semicolons appear with messages, so it easy to separate a routing message's payload from the routing

information that the message carries instructing it how to reach the destination. A sample routing message:

ROT,2,1,4,4,0,-1,0,0,0,-1,-1;INC,0,0,1,1

As a convenience, we terminate all messages with a new line character, (ASCII code 10). The Pebble modules ignore the new line character, but the new line characters ensure that each message appears on its own line when viewing a list of messages in a standard text editor.

There are many different types of messages used by the Robot Pebbles system. We list them here alphabetically with short explanations. Table 5.4 illustrates the format of each message type in more detail.

Bounding Box (BBOX) describes a rectangular bounding box by its minimum and maximum coordinates. Used to describe the approximate shape of the initial configuration of modules or the obstacle being duplicated.

Border (BOR) used to notify modules on the border of the duplicate shape of their special status.

Child (CLD) defines a parent/child relationship between neighbor modules so that each understands which is dependent on the other for power.

Disassemble (DIS) instructs the module to start the self-disassembly process.

Duplication (DUP) instructs modules on the border of the original passive shape being duplicated to send border (BOR) messages to their conjugate border modules that will form the border of the duplicate shape.

Fill (FIL) notifies all modules inside the duplicate border that they are part of the duplicate object and should not self-disassemble along with all other modules.

Group (GRP) used by neighboring modules to determine whether they are part of the same shape being formed or different shapes. If part of the same shape, the modules maintain their mechanical bond during disassembly. Otherwise, they break it.

Inclusion (INC) notifies a module that it is included in the structure being formed and that it should not self-disassemble. Not used during the duplication process.

Localization (LOC) request that a neighboring module send it position information to the localization message's sender because the module needs to localize itself.

Position (POS) specifies the position and rotation of the sender. Sent in response to a localization message.

Ready (RDY) indicates that a module is ready to process incoming routing messages. Until a module has received a ready message from a neighbor, it will not forward routing messages to that neighbor.

Reflection (REF) sent by modules to inform an external graphical user interface of their existence. Helpful to the user when debugging the system.

Routing (ROT) routes an embedded payload message to a specific destination in a given plane. ROT messages only support two-dimensional routing.

Sense (SEN) used to sense the shape of the obstacle that will be duplicated. Is always sent as the payload of a routing message.

Undeliverable (UND) sent back to the sender of the routing message when routing message cannot reach its specified destination. Sent as the payload of a routing message.

There are two additional messages types that are used for synchronous latching and unlatching of neighboring modules. To request that its neighbor synchronously latch with it, a module sends an ASCII ACK character (code 6) to its neighbor. Similarly, a module sends a ASCII NAK character (code 21) if it wishes to synchronously unbond from its neighbor. Once the two modules successfully exchange the message, they simultaneously energize the their electropermanent magnets to either latch or unlatch. Synchronously latching produces stronger bonds than if each module independently activated its EP magnet (see Section 3.4). Likewise, synchronously unlatching produces bonds with undetectable remnant force.

Table 5.4: Each inter-module message follows a predefined format that begins with a message type identifier and is followed by some number of data fields separated by commas.

Message Type	*Format*
Bounding Box	BBOX,<min x>,<min y>,<max x>,<max y>
Border	BOR,<leader x>,<leader y>;<border dir 1>,...,<border dir n>
Child	CLD,<sibling/child/parent>
Disassemble	DIS,<all/structure>
Duplication	DUP,<leader x>,<leader y>,<offset x>,<offset y>
Fill	FIL,<tangible src x>,<tangible src y>,<virtual src x>,<vitual src y>, <dest x>,<dest y>,<src UID>,<inside>,<leader x>,<leader y>, <offset dist x>,<offset dist y>
Group	GRP,<group number>
Inclusion	INC,<hop count>,<branch dir.>,<ignore>,<group number>
Localization	LOC
Position	POS,<tx face>,<position x>,<position y>,<rotation>, <min duplication area>
Ready	RDY,<notify/query>
Reflection	REF,<position x>,<position y>,<rotation>,<parent>,<neighbor 1 present>, ...,<neighbor 4 present>
Routing	ROT,<src x>,<src y>,<dest x>,<dest y>,<public>,<ideal dir>, <closest approach>,<departure x>,<departure y>, <departure dir>,<departure dir old>;<payload>
Sense	SEN,<src x>,<src y>,<dest x>,<dest y>,<src UID>,<perimeter>,<area>, <min x>,<max x>,<min y>,<max x>
Undeliverable	UND,<original dest x>,<original dest y>

5.5 Monitoring Link State

A module is only physically capable of transmitting or receiving messages on a single face at any give time. As a result, when module attempts to transmit a message to its neighbor, there is no guarantee that it will be successful on its first attempt. More likely, it will require several attempts before the transmission is successful. When a module fails to send a message to its neighbor, the module should not necessarily assume that the communication link is broken or that the neighbor is absent.

Still, for several reasons, a module does need a way to determine when a neighbor is unreachable. First, many high-level algorithms contain loops and `if` statements that depend on a message being sent to the module's neighbor. If the module never abandons its attempt to send a message to an unreachable neighbor, the algorithm will be stuck in an infinite loop. Second, if a module is parsing a routing message, we want it to find an alternate route to the message's destination instead of continuing its futile dedication to the forwarding the message along a static path. For both these

reasons, it is important that a module eventually identify broken communication links.

There are several reasons why communication links break. First, a module may be removed from the system, either by the the user, or by the execution of a shape formation control sequence. Second, a module may shift in the block so that it is no longer in contact with one or more of its neighbors. Because all of the modules are slightly different sizes, they do not pack perfectly into a grid. It is not completely uncommon for a module to have four neighbors, but only be able to communicate with three of them. Exactly which of its neighbors a module can communicate with can change as the topology of the system evolves. As some modules detach from the system, they relieve internal stresses resulting in slight mechanical realignments and consequently other communication links being formed or broken. Finally, a communication link can break if a module enters a fault state. The modules can experience both hardware and software faults that result in them entering a non-responsive state until power is removed. Between these three causes, it is not uncommon for several communication links between neighboring modules to change while the system is running.

Algorithm 6 shows how the link state monitoring process operates. To identify broken communication links, each module tracks how many unsuccessful attempts it has made to transmit a message to each neighboring module. Once the module successfully sends a message to one of its neighbors, (or the neighbor at least indicates that its receive buffer is full), the transmitting module resets its failed transmissions count (Lines 8–12 of Algorithm 6). If the count is not reset and passes a hard coded threshold, (Lines 19–21 of Algorithm 6), the module marks the given face/neighbor as non-responsive. Once a face is marked non-responsive, the low-level communication routines report failure after a single attempt when asked by the application code to transmit a message. As a result, the application code can follow a contingency plan instead of waiting indefinitely.

Once a face is marked non-responsive, the module attempts to re-establish the link to its neighbor. Whenever the low-level communication code is not attempting to send an application-level message to a neighbor, it attempts to send a ping (PNG) message to the neighbor (Line 4 of Algorithm 6). As soon as the module successfully sends a ping message to its neighbor, it marks the communication link active. When the neighbor receives a ping message, it simply discards it after

Algorithm 6 transmitMessage(*txFace*)—manages message re-transmission and link state monitoring

Require: $1 \leq txFace \leq 4$
1: **if** $txBuf[txFace].full =$ **true then**
2: $txStatus \leftarrow$ epMagTransmit($txFace, txBuf[txFace]$)
3: **else if** $txFace \notin unbondedNeighbors$ **then**
4: $txStatus \leftarrow$ epMagTransmit($txFace$, "PNG")
5: **else**
6: $txStatus \leftarrow$ epMagTransmit($txFace$, UNLATCH)
7: **end if**
8: **if** $txStatus =$ SENT **then**
9: $unsuccessfulTransmissions[txFace] \leftarrow 0$
10: $communicatingNeighbors \leftarrow communicatingNeighbors \cup \{txFace\}$
11: $txBuf[txFace].full \leftarrow$ **false**
12: value($txBuf[txFace].statusPtr$) $\leftarrow$ SUCCESS
13: **else if** ($txStatus =$ CHECKSUM-MISMATCH) **or**
($txStatus =$ CHECKSUM-TIMEOUT) **or**
($txStatus =$ RECEIVER-FULL) **then**
14: $unsuccessfulTransmissions[txFace] \leftarrow 0$
15: value($txBuf[txFace].statusPtr$) $\leftarrow$ AGAIN
16: **else**
17: $txBuf[txFace].remainingAttempts \leftarrow txBuf[txFace].remainingAttempts - 1$
18: $unsuccessfulTransmissions[txFace] \leftarrow unsuccessfulTransmission[txFace] + 1$
19: **if** $unsuccessfulTransmissions[txFace] > THRESHOLD$ **then**
20: $txBuf[txFace].remainingAttempts \leftarrow 0$
21: $communicatingNeighbors \leftarrow communicatingNeighbors \setminus \{txFace\}$
22: **end if**
23: **if** $txBuf[txFace].remainingAttempts = 0$ **then**
24: $txBuf[txFace].full \leftarrow$ **false**
25: value($txBuf[txFace].statusPtr$) $\leftarrow$ FAILURE
26: **else**
27: value($txBuf[txFace].statusPtr$) $\leftarrow$ AGAIN
28: **end if**
29: **end if**

sending the CRC acknowledgment byte. The ping messages only serve to test whether the link is active; they carry no information.

There is one exception to when ping messages are sent. Each module keeps a list of neighbors from which it has explicitly unbonded. Typically, this list is populated during the self-disassembly phase of shape formation. Once a neighbor has been added to this list, the module sends synchronous unlatch messages instead of ping messages (Line 6 of Algorithm 6). This serves as insurance that helps guarantee that the magnetic bond between the modules is actually broken.

5.6 Robustness: Responding to Broken Links

By detecting and attempting to gracefully handle broken communication links, we complicate the parsing of many messages. Often a message received from a neighbor prompts a module to transmit one or more more messages in response. For example, when a module receives a routing message, (unless it is the message's specified destination), the module needs to forward the message to one of its neighbors. More generally, once a module parses an incoming message from one of its receive buffers and determines what outgoing messages it needs to transmit in response, the module proceeds to load the outgoing messages into the appropriate transmit buffers. If we could guarantee that these outgoing messages would be successfully transmitted, the module could then purge the incoming message that it just finished parsing thereby freeing the receive buffer.

Because we cannot guarantee that an outgoing message loaded into a transmit buffer will actually be delivered, we must wait to purge the incoming message from the receive buffer. If we purge the incoming message too soon, the module may lose information crucial to reprocessing the incoming message when the module fails to transmit the outgoing message on first attempted face. For example, as routing messages are forwarded through the network of modules, the data fields in each message are constantly updated. If a module fails to forward a routing message along the ideal path as the result of a communication failure, but has already purged the incoming routing message, it will be impossible for the module to recover all the information necessary to reroute the message. Therefore, we must not purge incoming messages from their receive buffers until we have verified that any messages generated in response to the incoming message have been sent

successfully.

To explain the details of this verification process, we next describe the high-level loop driving each module. Each module does the following:

1. Attempt to receive messages from neighbors and update the face receive buffers (i.e. call `epMagReceive` once for each face).

2. Parse the messages in face receive buffers. This typically results in the `fillTxBuf` function (explained below) to be called.

3. Update internal state variables. This step, along with how the parsing function operates, determines the high-level behavior of each module.

4. Attempt to the send messages in the face transmit buffers to our neighbors (i.e. call `transmitMessage` once for each face).

5. Repeat

The framework that we use to monitor whether a message has been successfully sent is illustrated by Algorithm 7. In particular, the `fillTxBuf` function expects the caller, (typically the parsing function from step 2 above), to provide a pointer to a status field that may take on one of three values: SUCCESS, AGAIN, or FAILURE. To move a message into one of the transmit buffers, the caller must set the value of the status pointer to AGAIN when calling the `fillTxBuf` function. Then, assuming that the transmit buffer is empty, the `else` clause will be exercised, and the `fillTxBuf` function will move the message into the buffer. Note that the function call does not change the value of the status pointer, but it does copy the pointer itself to a field of the same name associated with the transmit buffer.

After the call, the parsing function knows that the message has not yet been transmitted, only loaded into the transmit buffer. Also note that if the transmit buffer were already full, the caller would never know the difference. The value of the status pointer would still be AGAIN, so the caller would know to call the `fillTxBuf` function again with the same parameters.

A simplified parsing function is shown by Algorithm 8. This parsing function simply attempts to transmit any incoming message back to its source, but it demonstrates how the `fillTxBuf` function is called and how its results are checked. In particular, note that the function expects to be told when the message that it is parsing is new in the sense that it is the first attempt made to parse it. When a message is new, the function sets the persistent *status* variable to AGAIN so that the call to `fillTxBuf` will copy the message to the buffer (assuming that it is empty).

Algorithm 7 fillTxBuf($txFace, overwrite, msg, repeat, statusPtr$)—moves a message into a particular face's transmit buffer

Require: msg: message that the caller wishes to transmit
Require: $overwrite \in \{\textbf{true}, \textbf{false}\}$: whether to overwrite the contents of the buffer
Require: $repeat > 0$: number of times to attempt to send the message
Require: value($statusPtr$) $\in$ {SUCCESS, AGAIN, FAILURE}: pointer to transmission status variable
Require: $1 \leq txFace \leq 4$
1: **if** $msg = txBuf[txFace].msg$ **and** value($statusPtr$) $\in$ {SUCCESS, FAILURE} **then**
2: **return**
3: **else if** $txbuf[txFace].full =$ **true and** $overwrite =$ **false then**
4: **return**
5: **else**
6: $txBuf[txFace].seqNum \leftarrow txBuf[txFace].seqNum + 1$
7: $txBuf[txFace].msg \leftarrow msg$
8: $txBuf[txFace].full \leftarrow$ **true**
9: $txBuf[txFace].remainingAttempts \leftarrow repeat$
10: $txBuf[txFace].statusPtr \leftarrow statusPtr$
11: value($statusPtr$) $\leftarrow$ AGAIN
12: **end if**

Algorithm 8 parseMsg($rxFace, msg, new$)—example message parsing function that demonstrates the proper use of the `fillTxBuf` function by echoing a received message back to the neighbor that sent it

Require: $1 \leq rxFace \leq 4$
Require: msg: message to be parsed
Require: $new \in \{\textbf{true}, \textbf{false}\}$: whether this is the first attempt at parsing this particular message
Require: $status \in$ {SUCCESS, AGAIN, FAILURE}: local, persistent status variable
1: **if** $new =$ **true then**
2: $status \leftarrow$ AGAIN
3: **end if**
4: fillTxBuf($rxFace$, **false**, msg, ∞, address($status$))
5: **if** $status =$ SUCCESS **then**
6: purgeRxBuf($rxFace$)
7: **else if** $status =$ FAILURE **then**
8: purgeRxBuf($rxFace$)
9: **end if**

Because the `fillTxBuf` function does not update the value of the status pointer, some other function must so that the parsing function eventually learns whether the transmission was successful. It is the `transmitMessage` function shown in Algorithm 6 called during step 4 of the high-

level loop that performs this update. When the `transmitMessage` function successfully sends the message, it updates the value of the buffer's status pointer which, (due to line 10 of Algorithm 7), points to the same status field that the parsing function provided as an argument to the `fillTxBuf` function.

To illustrate how this approach works, consider this scenario of iterations through the 5-step high-level control loop. Assume that the first call to `transmitMessage` after the initial call to `fillTxBuf` is unsuccessful. Consequently, the value of the status pointer will still be AGAIN (Line 27 of Algorithm 6). The second time the parsing function calls `fillTxBuf`, the `elseif` clause on Line 3 will be exercised because the transmit buffer already contains the message that the calling parsing function wishes to transmit. Now, after the second call to `fillTxBuf`, assume that the second call to `transmitMessage` is successful. As a result, the `transmitMessage` function will update the value of the status pointer to SUCCESS (Line 12 of Algorithm 6).

At this point, the parsing function will be called a third time, but because the *new* parameter is now false, it will not modify the value of the status pointer (Line 2 of Algorithm 8). Even though the value of the status pointer is already SUCCESS when the parsing function is called, the function does not check for this condition until after it has again called `fillTxBuf` (Lines 4–5 of Algorithm 8). This is acceptable because the third time the parsing function calls `fillTxBuf`, the `fillTxBuf` function will exercise the `if` clause on Line 1 of Algorithm 7 because the passed message will match the message already in the face transmit buffer, and the value of the status pointer will be SUCCESS. The `fillTxBuf` function will not take any action, but when it returns, the parsing function will proceed to check the status pointer's value. In doing so, it will learn that the message has been sent successfully. As a result, the parsing function purges the message it had been processing from the receive buffer (Lines 5–6 of Algorithm 8). In this example, the parsing function also purges the message from the receive buffer if it cannot be transmitted back to the neighbor from which it was received, but the parsing function could do something else instead.

The important fact to note is that even though the message had already been transmitted before the third call to `fillTxBuf`, that call did not reset the transmit buffer's `full` flag to true. If it had, the module would have attempted to send the same message twice. If the parsing function really

does want to send the same message twice, it must first wait until the value of the status pointer is SUCCESS. Then it must set the value back to AGAIN and call the `fillTxBuf` function a second time.

5.7 Link State Experiments

To verify that the modules correctly identify and respond to broken communication links, we performed 102 experiments in which we modified the topology of a network consisting of 16 modules. In all trials, we used the system to route a message between points A to B in the loop of modules shown in Figure 5-5. While the routing algorithm will be explained later, it suffices to say that by default, the messages always follow the shorter of the two paths taking four hops to reach their destination.

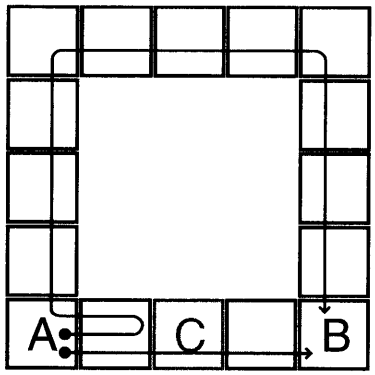


Figure 5-5: We routed messages from module A to B with and without module C present. The arrows show the two possible paths taken by the messages.

In 25 trials, the average time for a message to traverse from A to B was 5.13s with a standard deviation of 2.33s. To test the system's ability to reroute around a broken communication link we removed the module labeled C from the network in order to force the message to traverse the longer, 14-hop route to its destination. In 25 trials during which module C is removed just before the message departs from module A, the average delivery time is 22.06s with a standard deviation of 2.13s. In all 25 of these trials, the message reached its destination by taking the only

available, longer route. After each of these 25 trials, we also allowed the system to stabilize so that it knew, before a second routing message was ever sent, that module C was missing. In these trials, the average delivery time was 11.66s in 25 trials, and the standard deviation was 2.19s. So, by subtracting the average delivery time when the system did not know that module C is missing from the average delivery time when the system did know that module C was missing, we find that the system requires, on average, 10.40s to detect the broken communication link.

Finally, we re-inserted module C, and after it was localized and ready to process routing message, measured the time for a message to travel from point A to B. If module C's neighbors were slow to detect that it had been reintroduced, we would expect that the messages would take the longer route. This was not the case. In 25 trials, the average routing time was 4.60s and the standard deviation was 2.07s. From this, it is apparent that the system realizes that the shorter path is again available and begins using it with minimal delay. The results of all trials are summarized in Table 5.5.

Table 5.5: The low-level communication algorithms are capable of detecting and routing messages around dynamically broken links. Additionally, when those links are restored, the system again uses them to deliver messages along the shortest path.

Experimental Setup	*# Trials*	*# Successes*	*Avg. Time [s]*	*Std. Dev. [s]*
A $\rightarrow$ B, C Present	26	25	5.13	2.33
A $\rightarrow$ B, C Recently Removed	26	25	22.06	2.13
A $\rightarrow$ B, C Removed, System Stabilized	26	25	11.66	2.19
A $\rightarrow$ B, C Re-inserted	26	25	4.60	2.07

In the 102 experiments summarized in Table 5.5, we saw 2 cases where a routing message was not delivered. Any single module-to-module message exchange error would have been enough to cause either of these failures. Given that the 102 experiments corresponded to 918 message hops, the single hop failure rate is less than 0.22%. We would like to see a 0.0% failure rate, but the routing algorithms will require additional refinements to achieve this.

5.8 Two-Dimensional Routing

This section explores how we can transfer messages between modules that are not immediate neighbors. We have developed a routing algorithm that allows a module to specify an arbitrary destination for any message. The modules in the system then ensure that this message is delivered or automatically determine that the destination is unreachable. The ability to route message to any module in an ensemble is essential to the shape duplication algorithms presented in later chapters.

There are many possible strategies that we could employ in our routing algorithm. We need to ensure that whatever algorithm we choose is capable of handling non-convex topologies and missing communication links. Consequently, a simple gradient descent routing algorithm is not sufficient. The limited processing power and storage available to each module further constrain our choice of routing algorithms, For instance, it is not possible, especially as the number of modules in the system grows, to maintain routing tables. Instead, we choose to use the traditional bug algorithm [66] to route messages through the system. Instead of the bug being a robot, the message is the bug, and the modules are the environment through which the message must navigate from its source to destination.

5.8.1 Routing Algorithm

In particular, we use the Bug2 algorithm. This algorithm is provably correct [66] and ensures that, if it is possible for a message to reach its destination, it eventually will; and if it is not, the system will eventually be notified. The Bug2 algorithm is a natural choice for our system because it assumes that the bug has no access to global information. The bug only needs to determine its position and whether it is in contact with an obstacle, (in our case a void not occupied by a module), facts readily available from the modules themselves. The Bug2 algorithm is also advantageous because the bug only needs to maintain a constant amount of state information, and all this information can easily be stored in the message.

The messages moves from its source to destination by following a direct path vector from its source directly to its destination until it hits an obstacle. It then follows the obstacle until it re-encounters the direct path at which point it leaves the obstacle and continues along the vector to

the destination. This repeats until the message reaches its destination, or in the process of following an obstacle, re-encounters the position where it left the direct path vector to follow the obstacle. Should the message loop back on itself like this, it then knows that it cannot reach its destination.

5.8.2 Experimental Results

We have characterized the routing algorithm's speed in over 900 trials. First, we measured how quickly the system could deliver routing messages. We assembled a 5-by-5 grid of Pebble modules and then started the localization process. Once the modules were localized, we proceeded to measure the time required to route a message from the root module at $(0,0)$ to every other module in the system. Specifically, we routed at least 25 inclusion (INC) messages to each of the 24 destinations (we did not route messages to the root module). The inclusion messages that we sent alternated between informing their recipients that they were included or not included in the final structure. When a module received an inclusion message indicating that it was included, it turned its internal LED on. When it received a message indicating that it was not included, it turned it LED off. We used a stopwatch to measure the time between when we pressed the enter key on our PC's keyboard, thereby sending the inclusion message, and the moment when the LED toggled. After sending 25 including messages to one Pebble, we sent 25 to the next module without restarting or relocalizating the modules. Figure 5-6 illustrates the average time required to route messages from the root to any other module in the 5-by-5 grid. Figure 5-7 presents the same data but aggregated to show the message delivery time as a function of Manhattan distance between the source and destination. Both plots show that the delivery times increase as the message's destination moves farther from the source. Figure 5-7, in particular, illustrates the linear relationship between the delivery time and the Manhattan distance separating the source and destination.

We performed a total of 622 trials routing inclusion messages to the modules in the 5-by-5 grid. In the course of those trials, there were only three instances where an inclusion message was not successfully routed to its destination. This a 0.48% routing failure rate. If we are interested in the *single hop failure rate*, that is, how often low-level communication between modules failed, we need to know the total number of inter-module hops taken by all messages in the experiment.

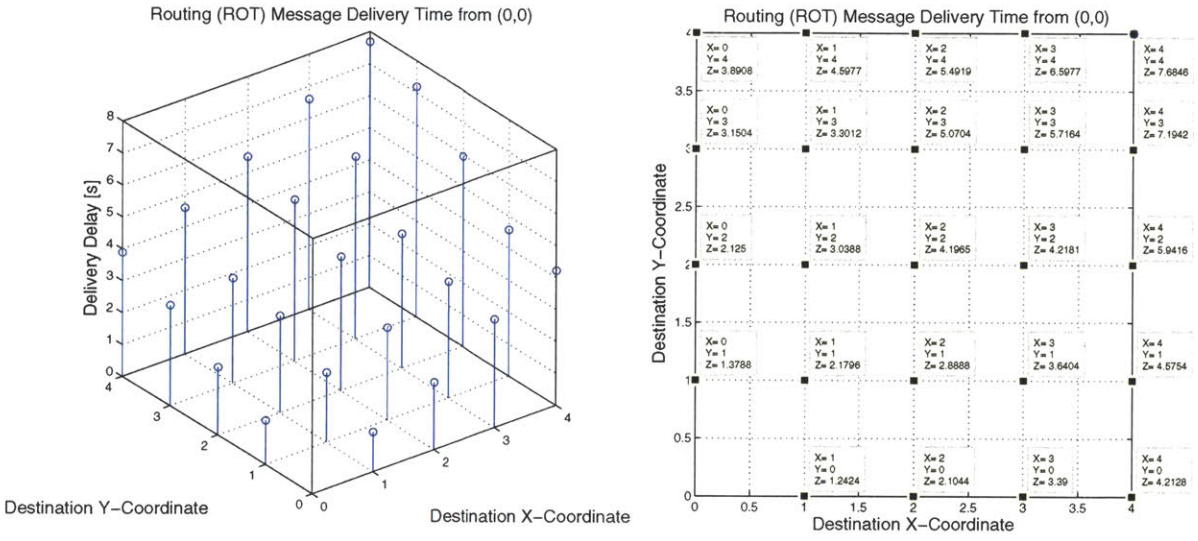


Figure 5-6: The time required for the system to route an inclusion message from the root module at $(0,0)$ to any other module is a linear function of the Manhattan distance that the message must travel.

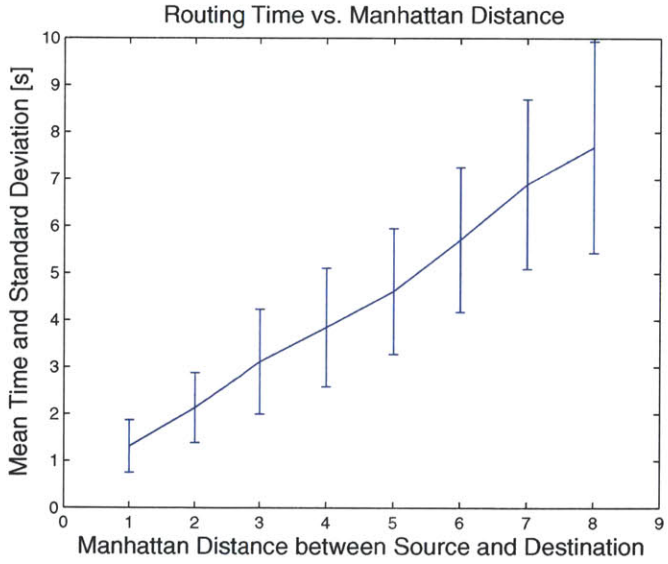


Figure 5-7: The time required by the system to route a message is a linear function of the Manhattan distance between the source and destination. This data was collected from over 600 trials.

Summing the number of hops required to reach each of the 24 destinations, we determine that the total number of hops across all trials was 2600. Given that there were still just 3 errors, the single hop failure rate is 0.12%.

We also characterized the time required for the system to determine that a routing message was undeliverable. To do so, we assembled and localized another five-by-five grid of modules. As before, we sent routing messages from the root module at $(0,0)$. Now, instead of sending the messages to modules that were in the system, we attempted to send them to non-existent modules just past the perimeter of the block. In particular, we attempted to route messages to the 13 modules below and to the right of this block (the row of modules $(-1,-1)$ to $(5,-1)$ and the column $(5,5)$ to $(5,-1)$). We attempted to send 10 messages to each of the 13 destinations. When the system discovered that the destinations did not exist, it routed an undeliverable (UND) message back to the root. We used a stopwatch to measure the time between when we pressed the enter key on our PC's keyboard, thereby sending the routing message, and the moment when the undeliverable (UND) message returned to the PC (via the root module).

Figure 5-8 shows the time required for the system to determine that a message's destination is unreachable, and Table 5.6 summarizes the results. In general, the system requires more time to determine that a message's destination is unreachable when the destination is further away from the source. The reason for this is illustrated by Figure 5-9. In its attempt to reach its nonexistent destination, each routing message first traverses the interior of the 5-by-5 array before colliding with, and then following the array's perimeter. In contract, for destinations that are nearby, the routing message starts to follow the perimeter almost immediately. Both messages completely circumnavigate the perimeter before returning to their initial collision points. Then each propagates back to the root module. Once again, the message originally destined for a distant module must take a longer path to reach the root.

In total, we performed 138 experiments to characterize the time it took the system to inform the sender of a routing message that the message was undeliverable. In these 138 experiments, we only saw 1 trial in which the root module was not informed that the routing message was undeliverable—a 0.72% failure rate. If one considers that each trial required that a message travel

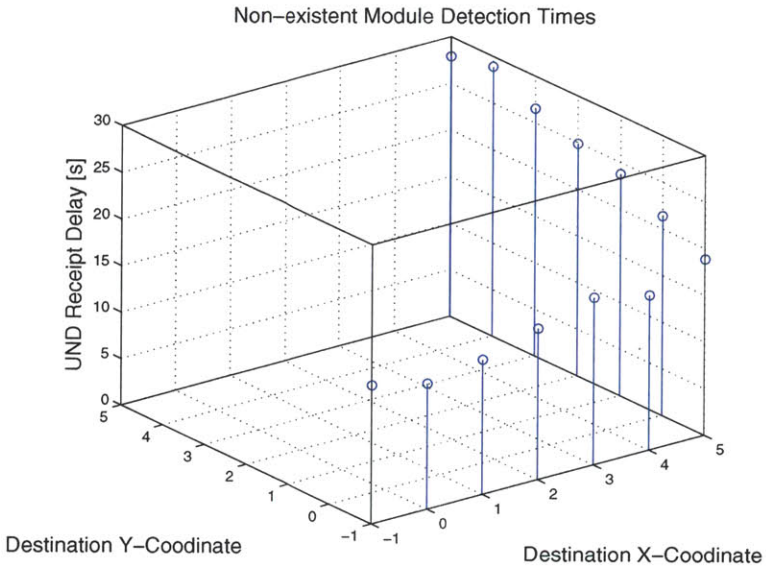


Figure 5-8: Using a 5-by-5 block of Pebbles, we measured the time it took for the module at $(0,0)$ to be informed that a routing message it had sent was undeliverable. The routing messages were intentionally sent to the 13 non-existent modules below and to the right of the of the 5-by-5 block.

Table 5.6: The system requires additional time to discover that routing message bound for destinations far from their source are undeliverable.

Destination	*# Trials*	*# Successes*	*Mean Time [s]*	*Std. Dev. [s]*
$(-1,-1)$	10	10	14.9	3.2
$(0,-1)$	10	10	13.5	2.1
$(1,-1)$	10	10	14.5	2.6
$(2,-1)$	12	12	16.2	2.7
$(3,-1)$	11	10	17.9	3.4
$(4,-1)$	10	10	16.6	1.6
$(5,-1)$	10	10	18.9	4.0
$(5,0)$	15	15	21.4	4.1
$(5,1)$	10	10	23.8	3.5
$(5,2)$	10	10	24.9	2.6
$(5,3)$	10	10	26.5	2.7
$(5,4)$	10	10	28.9	3.4
$(5,5)$	10	10	27.9	4.6

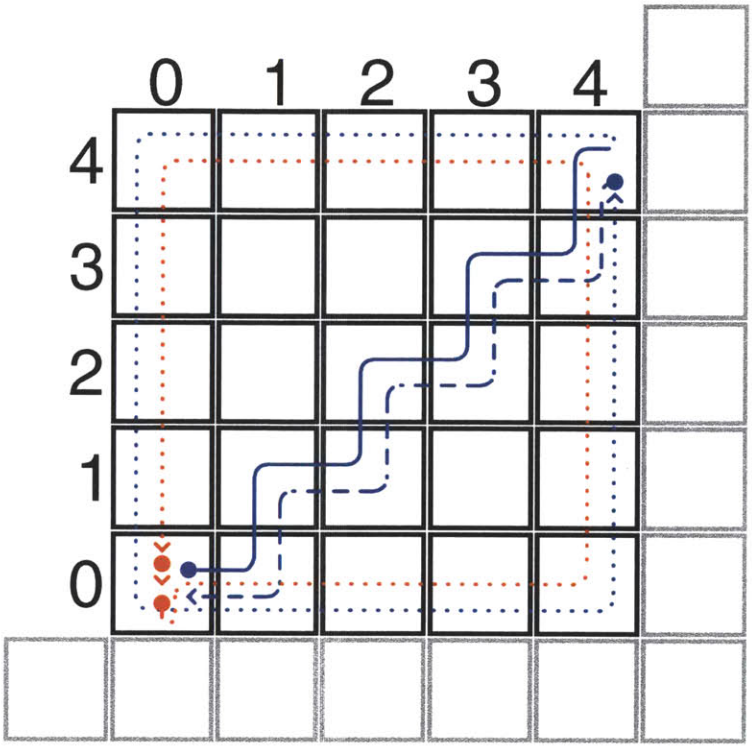


Figure 5-9: The solid blue trace is a routing message destined for the non-existent (grey) module at $(5,4)$. Once the message diverts from the direct path between its source and destination it becomes a dotted line and circumnavigates the perimeter of the 5-by-5 block looking for an alternative path to $(5,4)$. When the routing message returns to $(4,4)$ it realizes its destination is unreachable, and it send an undeliverable (UND) message, represented by the blue dashed trace, back to the root. The red trace represents the path that routing message destined for $(0,-1)$ follows. Immediately, the trace is dotted as the routing message searches for a alternative path to $(0,-1)$. Additionally, red the undeliverable (UND) message that results need not travel anywhere because it is sent by the modules at $(0,0)$ to itself. As a result, it takes much less time for the system to determine that there is no path to $(0,-1)$ than it takes to determine that there is no path to $(5,4)$.

at least 16 inter-module hops, the single hop failure rate is less than 0.045%.

Chapter 6

Shape Formation Basics

Chapters 3—5 presented the hardware and communication substrate on which the Smart Pebbles are built. In this chapter, we introduce the self-assembly and self-disassembly algorithms that we have developed for the Smart Pebbles, and we evaluate them on a 50-module hardware platform. Self-disassembling systems require two high-level capabilities: (1) the ability to aggregate the initial block autonomously and (2) the ability to remove modules from this block to form a particular shape by subtraction. In this chapter, we present solutions for both of these challenges. The two capabilities are inter-related. When the utility of an object built from Smart Pebbles is exhausted, the component modules are returned to the collective system. The self-assembly operation will create a new block, which, in turn, will be transformed into the next object by self-disassembly. To enable the creation of the widest range of objects, it is important that the result of self-assembly process be a solid block.

By aiming to form a close-packed lattice during the self-assembly phase, we eliminate the need to transmit a description of the goal shape to every module in the structure. Our approach avoids transmitting the complete shape description to all of the modules by notifying, with a single bit carried in an inclusion message, only those modules that are part of the goal shape. Any module that does not receive an inclusion message assumes, by default, that it is not part of the goal structure. When the self-disassembly process begins, these unincluded modules break their mechanical bonds with their neighbors while the modules that did receive inclusion messages

remain bonded.

Our two-step approach to shape formation attempts to minimize the amount of information transmitted to modules in the system because it does not transmit a complete blueprint of the structure to all modules. The alternative, in systems that perform self-assembly in a one-step process, is to transmit information to every single module indicating on which faces that module should allow neighbors to bond. This blueprint for the goal object has unacceptable communication and storage costs.

If the system distributes the complete blueprint to all modules, the communication cost scales as $O(n^2)$: the blueprint is size n, and one copy must be sent for each of the n modules. Additionally, because each module would need to hold, at least temporarily, a complete copy of the blueprint, the storage requirements for each module would scale as $O(n)$.

Alternatively, an external controller could send a local section of the blueprint to each module as the module joins the system. With this approach, the message size and memory requirements could be reduced to $O(1)$, but the overall communication cost still scales as $O(n^2)$. To understand why, consider a single chain of n modules. The local piece of the blueprint sent to the last module in the chain would take n hops to arrive. Likewise, the blueprint message sent to the second to last module would take $n-1$ hops, etc. Summing all of hops taken by the individual messages (1, 2, 3, $\ldots, n-2, n-1, n$), the total is still $O(n^2)$. This approach also necessitates that the newly attached modules send messages back to the controller requesting the relevant sections of the blueprint.

Distributing the unique local sections of the blueprint to each module also requires that the system efficiently route these local blueprint messages to their destinations. If the routing information is contained in the message, it adds another factor of n to the communication cost, making it $O(n^3)$. This is due to the fact that it could require $O(n)$ information to describe the route to a module that is n hops distant from the message's source. More sophisticated routing algorithms, like the bug algorithms presented in Chapter 5 may reduce the cost, but routing is never free. One can always construct test cases that elicit the worst case performance from any routing algorithm.

In addition to simplifying communication, the other major advantage of self-assembling a close-packed lattice before self-disassembling into the desired shape is that the modules in the

lattice form a supportive scaffolding. The scaffolding adds mechanical rigidity while the goal shape is being formed, and it also helps to better constrain and align modules as they attach to the system. The scaffolding provides more potential routing paths for messages in the system. Likewise, the scaffolding provides more current paths and thereby reduces the electrical impedance between any two modules in the system.

Once the system has formed a solid, close-packed block of material, we have two ways to convey the desired shape to the system: sculpting and distributed duplication. In this chapter, we focus on the sculpting process. It uses inclusion messages to inform individual modules that they are part of the goal shape and should maintain their mechanical bonds with other included neighbors when all other modules in the system self-disassemble. Consequently, it avoids transmitting the entire shape description to the structure as a whole.

The sculpting process minimizes the cost of routing by constructing the routes for the inclusion messages using an external controller. When distributing the inclusion messages, the system reuses most routing information from the previous message when delivering the next message. As a result, each message carries only $O(1)$ routing information and the total communication cost is $O(n^2)$. For more information, about the route construction process, consult our MEng thesis [32].

The distributed duplication approach is covered by Chapters 7 and 8. It enables a computer-free user interface for shape specification by providing algorithms for autonomous shape sensing and duplication using a miniature physical model of the desired shape. Using the duplication algorithms, the system is able to autonomously sense the shape of, and duplicate, a passive object surrounded by Smart Pebble modules. The advantage of duplication is that we completely eliminate the need for an external controller. This makes the system more practical, and it eliminates the extraordinary communication burden from whatever module had been serving as the communication link between the system and the external controller.

6.1 Sculpting

Shape formation by sculpting is a six-step process:

1) Self-assembly/Neighbor Discovery The self-assembly process attempts to construct an initial block of programmable matter modules that are aligned in a close-packed lattice. Alternatively, if the user wishes to skip the self-assembly process, he can manually assemble the modules into an initial block. During both processes, the modules discover and begin to communicate with their neighbors. The self-assembly processes is discussed in Section 6.2.

2) Localization During localization, each module learns its relative position within the initial block of material. If the system is self-assembling, localization is integrated into the self-assembly phase. If, instead, the user is manually assembling the initial block of material, the user initiates the localization process by transmitting a position (POS) message to the root module. Position messages then propagate throughout the entire structure so that each module learns its position.

3) Reflection After each module has learned its position, it sends a reflection (REF) message to the root module. The root modules passes these REF message to a GUI running on the user's computer. Each REF message contains the position and orientation of the module that sent it. Both localization and reflection are described in more detail in Section 6.3.

4) Virtual Sculpting The GUI constructs a virtual model of the physical system using the incoming REF messages. The user then employs the GUI to select which modules should remain bonded with their neighbors to become the goal shape and which should self-disassemble. This result of this virtual sculpting process is a series of inclusion (INC) messages that will convey the desired shape to the structure during the shape distribution phase. For details on how this series of inclusion messages is generated, consult [34].

5) Shape Distribution After this sculpting process is complete, the program generates a sequence of inclusion messages. During the shape distribution stage, the GUI transmits these inclusion messages to a the root module. The structure then propagates these inclusion messages to their proper destinations. As with the localization process, the messages only contain local information. Shape distribution is described in Section 6.4.

6) Self-Disassembly During the disassembly phase, the modules not designated to be in the final structure disconnect from their neighbors to reveal the shape the user previously virtually sculpted. Self-Disassembly is described in Section 6.5.

6.2 Self-Assembly

The goal of self-assembly is to aggregate a solid block from all of the free modules available in the system. During the self-assembly process, we want to ensure that no voids are formed in the growing structure. Voids restrict the set of shapes that can be sculpted from the initial block; weaken the structure; and reduce the available communication and current flow paths. If we allow new modules to be accreted at any location on the growing structure, it is easy to create gaps in the structure that are theoretically difficult and practically impossible to fill; a loose module will never fill a lattice position that is already surrounded on three sides. Therefore, our goal is prevent the creation gaps surrounded by neighboring modules on more than two sides. Doing this also guarantees that we do not create voids in the structure.

6.2.1 Self-Assembly Algorithm

To avoid holes in the self-assembled structure, we propose a simple distributed algorithm that only requires local information. Based on this information, each free module coming into contact with a potential bonding site on the solidified structure must decide whether to permanently bond with the structure or move on and look for another bonding site. The algorithm we describe is similar to the self-assembly rule set generated by Matarić et al. in [44] for forming a rectangular structure. Matarić et al.'s work focuses on the broader question of how to generate a set of rules to assemble arbitrary structures, and as a result, generates a larger, more complex set of rules that depends on each module knowing in which of eight potential sectors it resides. In contrast, our work focuses on developing a minimal complexity, easy to implement algorithm that guarantees the assembly of a close-packed lattice. By following the self-assembly process with self-disassembly, we eliminate the need for complex sets of rules which govern when and where modules may attach

to the growing structure.

Our self-assembly algorithm makes two assumptions. First, all modules correctly assume the location of the root module. This is easy to hard-code into each module's processor as location $(0,0)$. Second, once each module is added to the structure, it can determine it's (x,y) position. This requirement is also easy to meet. The user informs the one module anchored to the assembly platform that it is the root and therefore at location (0,0) and un-rotated. Using this information, the root can inform the module added to its right that the new module's location is (1,0). Likewise, the module added below the root is at location (0,-1), etc. Based on which of its faces the new module receives this message, it can determine its orientation. Now that the root's neighbors know their locations and orientations, they, in turn, inform their newest neighbors of their locations. More details, and a proof that this algorithm is correct are proved in [34]. Note that the algorithm only requires neighbor-to-neighbor communication, and it does not rely on any global information being communicated within the structure. All modules in the structure are able to to determine their coordinates without any concept of the structure as a whole.

The entire self-assembly algorithm, shown as pseudocode in Algorithm 9 begins as the free module receives power when it comes into contact with a module already a part of the crystallized structure. Immediately, the module queries its neighbor to determine its location. Based on this location, the module then constructs a *root vector* pointing back to the root module. The vector may have x- and y-components. The new module permanently bonds with the structure—by calling the `latchAllFaces()` function—if it detects that it has neighbors in both the x- and y-directions of the root vector, (if they exist). For example, consider a new module that determines its location is (10,2). As shown in Figure 6-1, the root vector is then (-10,-2) which has both x- and y-components. As a result, the module only bonds with the structure if it has neighbors at (9,2) and (10,1). Instead, if the new module were located at (0,-5) and the root vector was (0,5), the module in question would only bond if it detected a neighbor at (0,4).

If the new module does not detect neighbors along both components of its root vector, it informs whatever neighbors it is contacting, and they deactivate their connectors releasing the module. The module will lose power, so when it next contacts the structure, its self-assembly algorithm will

restart.

Once a module decides that it should permanently bond to the growing structure, it enters a loop in which it simply listens for disconnect request messages on its faces. When a new module decides that it cannot connect to the structure, it sends one of these disconnect request messages—using the `unlatchAllFaces()` function—to all of its neighbors. When the previously a solidified module receives one of these messages on a particular face, the previously solidified module keeps the connector on that face deactivated for a fixed period of time to allow the rejected module to move out of range of its attractive force. This is the purpose of the `disableFace()` function in the pseudocode (line 20). Eventually, the connector is reactivated in hopes that the bonding site will have become valid.

Algorithm 9 selfAssemble()—algorithm uses the existence or absence of two of a module's neighbors to determine whether it is allowed to bond with its neighbors and become a part of the growing structure.

Require: $\overrightarrow{myPos}$: module's location as determined by localization process
1: $\overrightarrow{root} \leftarrow (0,0) - \overrightarrow{myPos}$
2: **if** $\overrightarrow{root}.x \neq 0$ **then**
3: $\quad \overrightarrow{neighborPos} \leftarrow (\overrightarrow{myPos}.x + \text{sign}(\overrightarrow{root}.x), \overrightarrow{myPos}.y)$
4: $\quad$ **if** neighborExists($\overrightarrow{neighborPos}$) = **false then**
5: $\quad\quad$ unlatchAllFaces()
6: $\quad\quad$ **return**
7: $\quad$ **end if**
8: **end if**
9: **if** $\overrightarrow{root}.y \neq 0$ **then**
10: $\quad \overrightarrow{neighborPos} \leftarrow (\overrightarrow{myPos}.x, \overrightarrow{myPos}.y + \text{sign}(\overrightarrow{root}.y))$
11: $\quad$ **if** neighborExists($\overrightarrow{neighborPos}$) = **false then**
12: $\quad\quad$ unlatchAllFaces()
13: $\quad\quad$ **return**
14: $\quad$ **end if**
15: **end if**
16: latchAllFaces()
17: **loop**
18: $\quad$ **for** $face \leftarrow 1$ to 4 **do**
19: $\quad\quad$ **if** disconnectRequested($face$) = **true then**
20: $\quad\quad\quad$ disableFace($face$, LOCKOUT-TIME)
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: **end loop**

Theorem 1. *The self-assembly algorithm (Algorithm 9) prevents the formation of gaps in the lattice structure which are surrounded by more than two neighbors.*

Proof. Guaranteeing that the algorithm never creates a gap that is surrounded on more than two

sides is equivalent to ensuring that, on any vertical or horizontal line of the lattice, an unpopulated gap between two distant modules is not formed. Consider, for illustrative purposes, an any unoccupied position on the lattice and the horizontal, (or vertical), line extending to positive and negative infinity from this point. If this line intersects solidified modules, (arbitrarily far away), in both the positive and negative directions, one could imagine working from the solidified modules inward to fill this gap. Eventually, enough modules will be attached so that the initial unoccupied position has immediate neighbors to its left and right. Once this occurs, the empty position, will be impossible to fill. As a result, if the algorithm avoids creating a gap, no matter how wide, along any horizontal or vertical transect of the lattice, it will avoid creating gaps in the lattice which are surrounded by more than two immediate neighbors.

The self-assembly algorithm, if it does not detect immediate neighbors along both the x- and y-components of a vector pointing from the potential bonding site to the root module, assumes that other, more distance modules may exist along those transects. As a result, by not connecting a module to the structure, the algorithm does not risk creating gaps along these transects.

Finally, the algorithm is guaranteed not to create gaps along the x- and y-vectors originating at the potential bonding site but pointing away from the root module. For this type of gap to be created, a solidified module would have exist farther away from the root in either than x- or y-direction than the bonding site in question. Conveniently, this is impossible. As explained in the preceding paragraph, a module will never bond if there is any potential for an empty position in the lattice along either component of the module's root vector, which, in this scenario, there would have been.

□

Theorem 2. *The self-assembly algorithm prevents the formation of holes in the lattice.*

Proof. By Theorem 1, the self-assembly algorithm never creates gaps with more than two neighbors, so the algorithm can never create a gap with four neighbors–the definition of a hole. □

While the algorithm presented here has pertained to a two dimensional system, the extension to 3D is straightforward. Instead of a 2D vector pointing back to the root module, each module will have a 3D vector and may need to check for neighbors along the three potential components of the

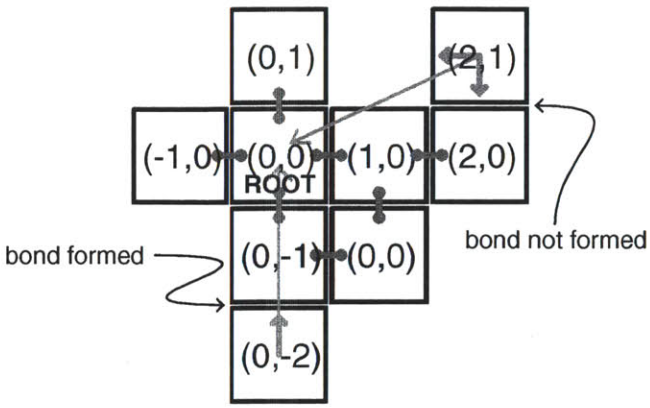


Figure 6-1: During self-assembly, modules only permanently attach to the already assembled structure if they detect immediate neighbors along a vector that points back to the root module. The module at (2,1) does not attach because, while it has a neighbor along the y-component of its root vector at (2,0), it does not detect a neighbor at (1,1) along the x-component of the vector. The module at (0,-2) does attach to the crystallized structure because it detects a neighbor at (0,-1), along the y-component of its root vector. The root vector does not have an x-component, so the module does not attempt to detect neighbors at (-1,-2) or (1,-2).

root vector. Likewise, the 3D algorithm guarantees that a gap with more than three neighbors will never be created which implies than holes will never be created.

6.2.2 Self-Assembly Experiments

We experimentally tested the self-assembly algorithm using a collection of 17 Smart Pebble modules. In three dimensions, we imagine shaking a bag full of modules to drive the self-assembly process. The 2D analog is an inclined vibration table. We built a custom vibration table that provides the stochastic forces necessary to move and align the modules (see Figure 6-2). The amplitude of the vibration can be controlled with a variac and we can also change the tilt of the table. The perimeter of the table is surrounded by a low barrier that prevents modules from falling off.

In our experiments, we anchored one module, the root, in a corner of the vibration table at coordinates $(0,0)$. The root module provides the power and communication link between the system and the user. Then we tilted the table 4 degrees in both the x- and y-directions to bias the movement of all free modules toward the root module located at $(0,0)$.

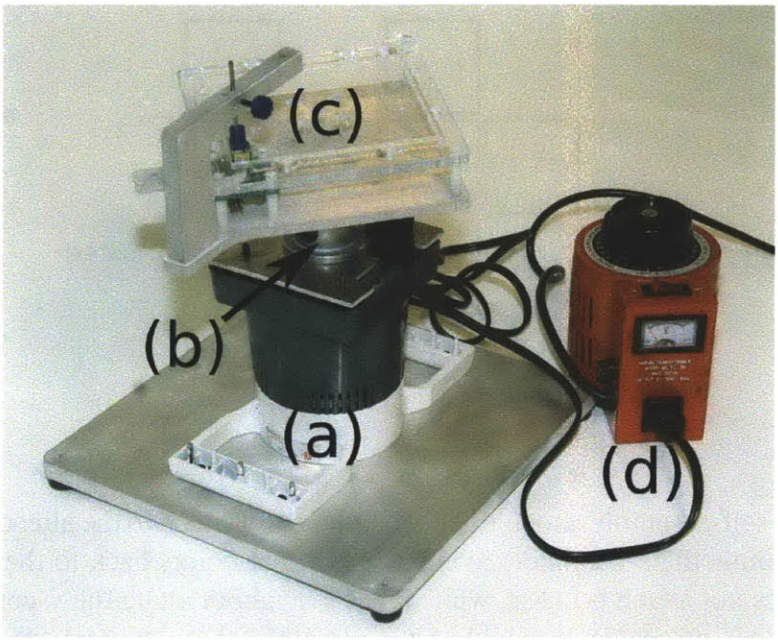


Figure 6-2: A vibration table is used to drive the self-assembly process. It consists of a vibrating base (a), a universal joint to control tilt (b), the assembly surface (c), and a variac (d) to control the vibration frequency.

Using 16 randomly arranged modules, (in addition to the fixed root module), we first tested the self-assembly algorithms. A progression of still images from one of these trials is shown in Figure 6-3. As shown in the last frame of the figure, after the modules have coalesced and have been given sufficient time to latch with their neighbors, the solidified structure can be removed from the test fixture without falling apart.

Figure 6-4 shows how the 17 modules tended to be distributed after all modules had settled into discrete grid positions. The data was collected from a set of 13 trials. Not surprisingly, the experiments show that the modules tend to form an isosceles right-triangular configuration. In addition to determining the most likely distribution of initial modules, we wanted to ensure that all modules were able to bond with their neighbors and communicate with the system's PC-based user interface. In a series of 15 trials, each using 17 modules, we observed a total of only 22 instances in which a module failed to localize and send a message back to the user interface through the root module—a failure rate of 8.2%. In most of these cases, one or more modules was clearly

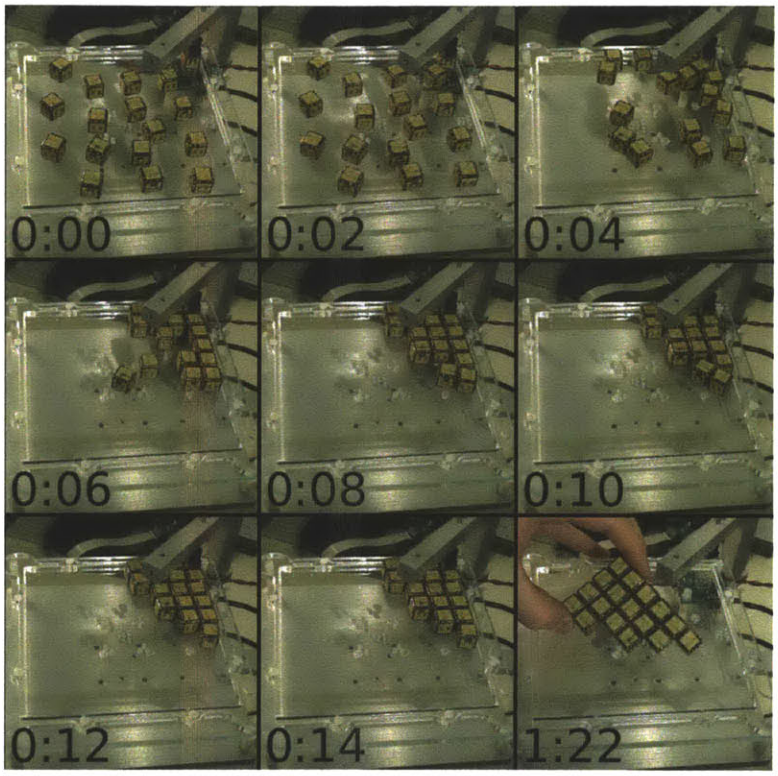


Figure 6-3: A collection of 16 randomly distributed Smart Pebble modules, (each a 12mm cube), and one fixed root module, (back right of each video frame), self-assemble when placed on an inclined vibration table. Initially, the connectors on each module are deactivated, and they are only turned on when a module successfully communicates with the growing structure. The last frame shows that all modules bond together to form a solid shape that can then be used for self-disassembly.

not in contact with one more of its neighbors. In one particularly bad trial, one of the modules adjacent to the root was about 45 degrees out of alignment resulting in 13 of the 17 modules not localizing. This was the only trial of the 15 in which the vibration table was unable to align all of the modules. The average time taken to self-assemble the 17 modules was 1min, 47sec. The self-assembly process worked most efficiently when the table vibration was swept up and down several times through varying amplitudes.

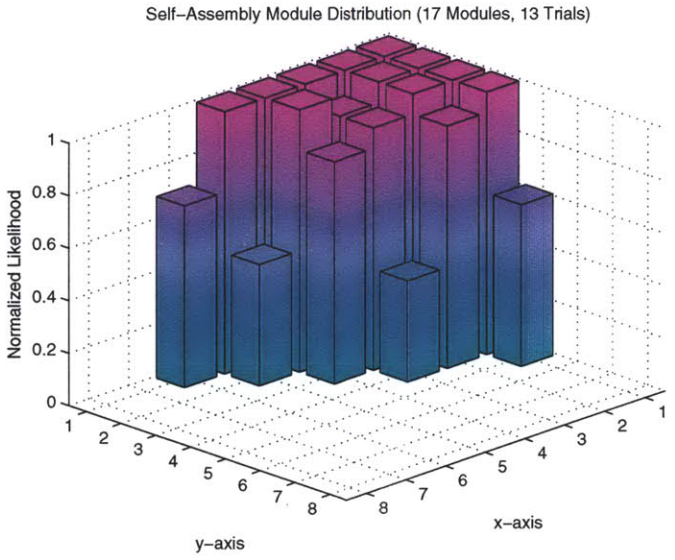


Figure 6-4: When 17 modules are placed on a vibration table included so that the $(0,0)$ location is the table's low point, the modules self-assembly into a close-packed lattice. The likelihood that a particular position in the lattice is filled is shown in this plot.

6.3 Localization and Reflection Algorithms

Before the Smart Pebbles system can be used to form shapes, either by sculpting or duplication, each of the modules must learn its location within the initial block of modules. Additionally, if the system is self-assembling, each module must learn its location and orientation so that it can determine whether it is bonding to the growing structure in a valid location. When the user is forming shapes by sculpting, he needs to know which modules exist so that he can determine how to place

and orient his goal shape within the initial block of material from which the shape will be sculpted. While the user could approximate the placement by eye in smaller two-dimensional structures, it will become increasingly difficult to do in 3D configurations with more than a hundred modules. To inform the user how the initial block of material is configured, each module sends a reflection (REF) message to a GUI running on the user's computer. Using a virtual model constructed from these reflection messages, the user can easily determine where within the initial block of material to place the goal shape.

6.3.1 Localization Algorithm

When a module first receives power from one of its neighbors, it immediately begins sending localization (LOC) messages to its neighbors. Each localization message is a request for the receiver to reply with its position, if it knows it. The localization begins when the user sends one module, the root, a position (POS) message assigning that module an arbitrary set of coordinates and rotation. In practice, we always tell the root that it is located at the origin, $(0,0,0)$, with a rotation of 0 degrees. With its position known, the root module can begin responding to the incoming LOC messages. The root's neighbor's will learn their positions and then begin sending POS messages to their unlocalized neighbors. Eventually, all modules in the system will learn their location relative to the root.

Each module must receive a single POS message in order to localize, so the number of messages that need to be exchanged is $O(n)$, where n is the number of modules in the system. The worst case running of the localization process occurs when n modules are arranged in a line. Each module in the line cannot localize until the previous module knows its own position. Each module requires a constant amount to time to process an incoming POS message from its newly localized neighbor, so the total running time is $O(n)$. In other structures, the *average* running time is $O(m)$ where m is the longest dimension of the structure. In particular, the average running time is proportional to the Manhattan distance between the root and the most distant module.

Each POS message contains the transmitter's rotation and transmitting face number in addition to the transmitter's coordinates. By combining the transmitter's rotation and transmitting face

number with its receiving face number, the receiver can determine its own rotation using a look-up table. Combining this with the transmitter's coordinates, the receiver can compute its own location. Figure 6-5 illustrates how a module's face numbers are assigned. Note that we do not specify a module's rotation as a number of degrees because in three dimensions we would need to specify a rotation axis. Instead, we specify rotations by the module's face numbers than align with the principal axes.

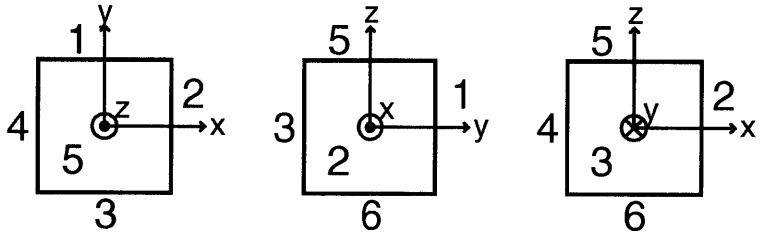


Figure 6-5: The six faces of each Pebble are numbered 1–6. Faces 1–4 correspond to the four faces containing EP magnets. Face 5 is the top face, and 6 is the bottom. Rotations are specified by which faces align with the principle axes. The figure shows three different views of a Pebble whose orientation is X2Y1Z5. That is, face 2 aligns with the positive x-axis, face 1 aligns with the positive y-axis, and face 5 aligns with the positive z-axis.

6.3.2 Three-Dimensional Localization

Localization in three dimensions is more complex than localization in two dimensions. The reason for this is that receiving a POS message from a neighbor only partially constrains the rotation of the receiver. This is because two neighboring modules cannot sense their relative orientation about an axis that passes through the center of the two modules. To fully localize in three dimensions, a module must receive two POS messages from two orthogonal directions. One consequence of this constraint is that a module with neighbors along only one axis will never localize in 3D. For example, a single chain of modules will never localize in 3D. In three dimensions, each modules must receive two POS messages, so localization still requires $O(n)$ time.

6.3.3 Localization Experiments

To verify that the localization algorithms operate correctly, and to measure their running times, we performed experiments in hardware and simulation. In hardware, we performed a total of 256 trails. In all trials, we arranged the modules on the test fixture shown in Figure 3-14 that provides power to the root module, (the module clamped to the fixture), and a communication link between it and an external computer. The root module was always situated at the lower-left of the arrangement of modules. We started the localization process by sending a position (POS) message to the root module. As the position messages propagated, the other modules learned their positions. When they did, their internal LEDs stopped flashing and stayed solid for several seconds. We measured the localization time with a stopwatch that we started as soon as we sent the first POS message and stopped when all LEDs were illuminated solid.

Of the 256 hardware trials, 155 characterized the localization time in m-by-1 lines of modules, where m varied from 3 to 12. We performed at least 15 trials for each value of m. The other 101 trials characterized the running time of the localization algorithm in m-by-m squares of modules ranging in size from $m = 2$ to 5. We performed at least 25 trials for each square. All trials were successful and resulted in all modules learning their positions. Figure 6-6 shows the average time required for all modules in a line to localize. Figure 6-7 show the average localization time for squares. The plots show that the localization algorithm's communication cost obeys the expected $O(n)$ limit though it is only tight for lines of modules.

To show that the localization algorithm's $O(n)$ time scaling continues for larger groups of modules, we used the simulator presented in Chapter 4. The simulated localization times for lines and squares are shown in Figures 6-6 and 6-7 alongside the hardware results. We performed 192 trials with m-by-1 lines of modules in which m varied over 15 different values between 2 and 50. The minimum number of trials for a given value of m was 10. We performed 240 trials with m-by-m squares of modules in which m varied from from 2 to 10. The minimum number of trials for a given value of m was 15. Using the simulator, we also simulated localization in m-by-m-by-m cubes of modules. In our simulations, m varied from 1 to 7. When simulating cubes, we performed 151 trials, and the minimum number of trials for any given value of m was 16. The results of these

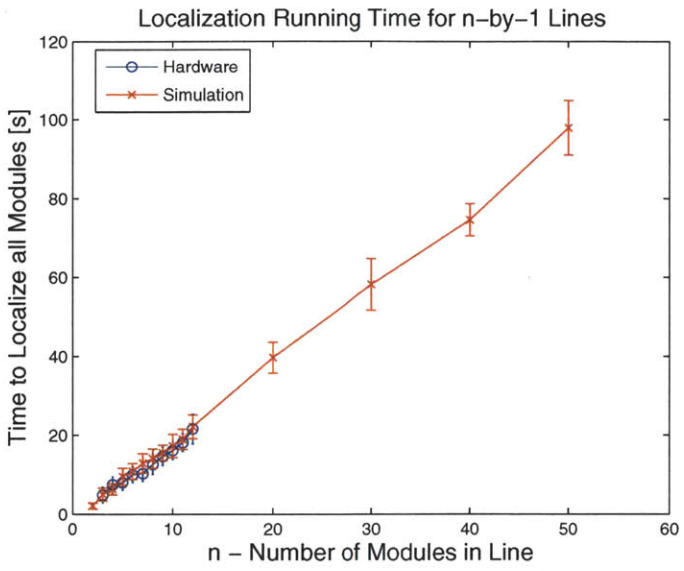


Figure 6-6: As the localization algorithm indicates should be the case, the time required for a line of modules to localize is $O(n)$, where n is the length of the line. The bars on each data point indicate one standard deviation.

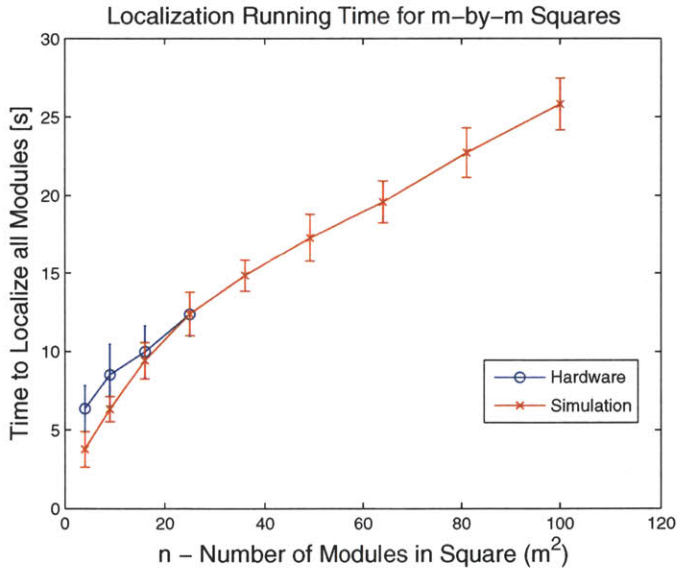


Figure 6-7: The average time required for a square sheet of modules to localize scales as the Manhattan distance between the root and the most distant module. For squares of modules with the root in a corner, the Manhattan distance is proportional to the square root of the number of the total number of modules. The bars on each data point indicate one standard deviation.

trials are shown in Figure 6-8.

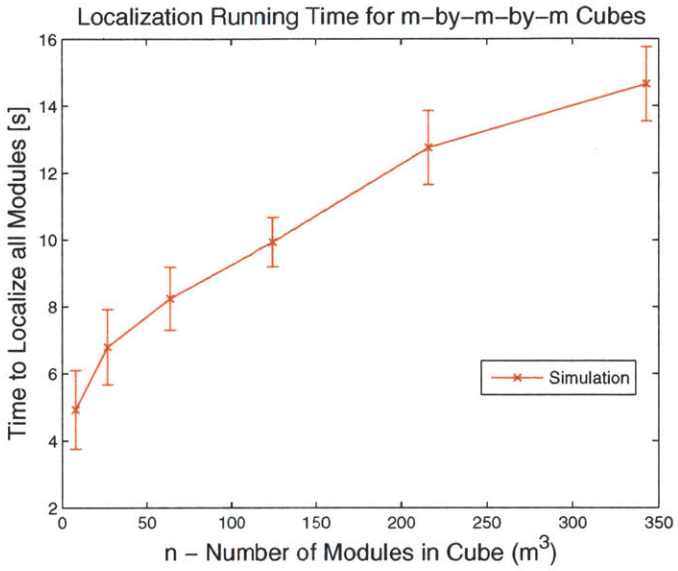


Figure 6-8: The average time required for a cubic block of modules to localize scales as the Manhattan distance between the root and the most distant module. For cubes of modules with the root in a corner, the Manhattan distance is proportional to the cube root of the number of the total number of modules. The bars on each data point indicate one standard deviation.

As Figures 6-6, 6-7, and 6-8 illustrate, the $O(n)$ running time bound, (where n is the total number of modules), is not tight for all shapes. As discussed above, the localization time is actually linearly proportional to the Manhattan distance between the root and the most distant module. That is, the running time scales as $O(m)$ where m is the largest dimension of the collection of modules. Figure 6-9 shows the running time of the simulated localization algorithm as a function of m, which we label the object's diameter. It confirms our assertion that the localization time obeys an $O(m)$ limit.

Figure 6-10 show the running time of the localization process in hardware as we vary the aspect ratio of a rectangle composed of twelve Smart Pebble modules. The left-most data point is the localization time of a 4-by-3 module rectangle. The middle data point corresponds to a 6-by-2 module rectangle, and the right-most data point corresponds to a 12-by-1 module rectangle. The plot demonstrates that even though the number of modules remains fixed, their arrangement plays a large role in the localization time.

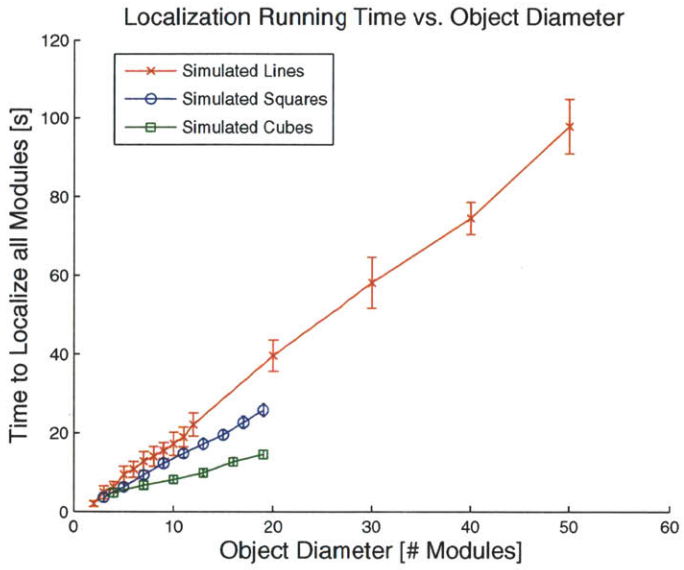


Figure 6-9: The average time required for any arrangement of modules to localize scales as the Manhattan distance between the root and the most distant module. When the root is in a corner of the arrangement, (as it is in all of our experiments), this Manhattan distance is the object's *diameter*. The bars on each data point indicate one standard deviation.

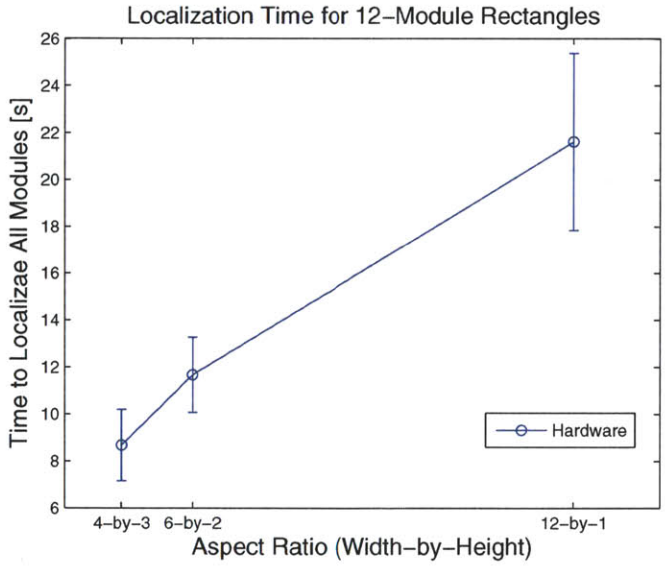


Figure 6-10: As a rectangular arrangement of modules approaches a square, the localization running time is minimized. This is due to the fact that a square minimize the Manhattan distance between the root and the most distant module. The bars on each data point indicate one standard deviation. Each data point is averaged from 15 trials.

The simulator allows us to record the number of messages exchanged during the localization process. Figures 6-11, 6-12, and 6-13 illustrate how the number of messages exchanged during localization scales with object size of lines, squares, and cubes, respectively. Note that the scaling is not linear as predicted above. The observed quadratic behavior is due to the fact the simulator counts all message types, not just POS messages, exchanged during the localization process. Each module, until it is localized, continuously broadcasts localization (LOC) messages to all of its neighbors. These LOC messages are included in the message counts shown in Figures 6-11, 6-12, and 6-13. Each of the n modules sends LOC messages at a fixed rate until it is localized. Because the localization process runs in $O(n)$ time, the number of LOC messages sent will therefore scale as $O(n^2)$.

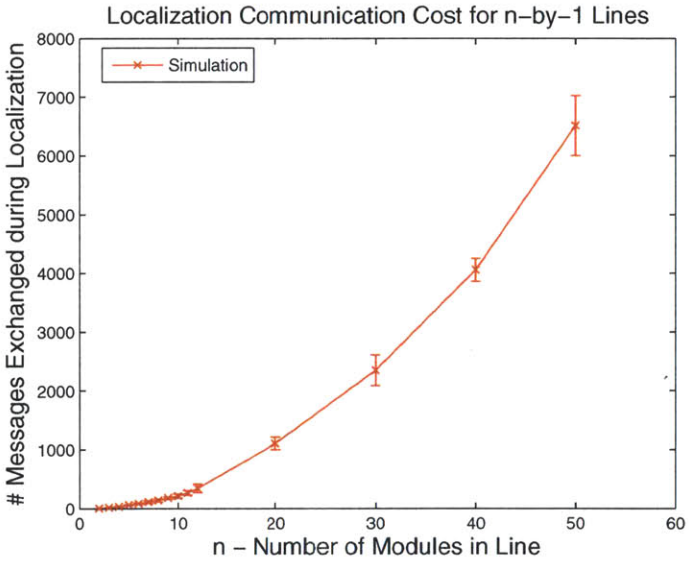


Figure 6-11: Despite the fact that only $O(n)$ POS messages are needed to localize n modules (regardless of configuration), the total number of inter-module messages exchanged during localization of a line scales as $O(n^2)$ because we also count the LOC messages that each module sends continuously until it is localized. The bars on each data point indicate one standard deviation.

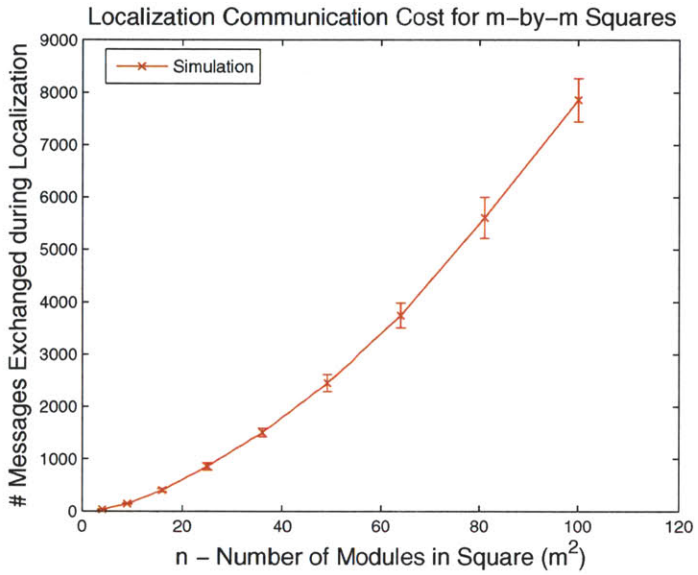


Figure 6-12: Despite the fact that only $O(n)$ POS messages are needed to localize n modules (regardless of configuration), the total number of inter-module messages exchanged during localization of a square sheet scales as $O(n^2)$ because we also count the LOC messages that each module sends continuously until it is localized. The bars on each data point indicate one standard deviation.

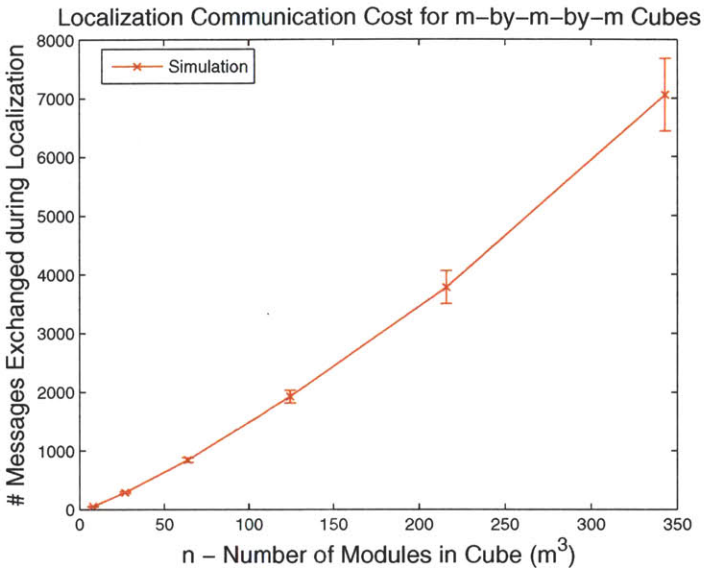


Figure 6-13: Despite the fact that only $O(n)$ POS messages are needed to localize n modules (regardless of configuration), the total number of inter-module messages exchanged during localization of a cubic block scales as $O(n^2)$ because we also count the LOC messages that each module sends continuously until it is localized. The bars on each data point indicate one standard deviation.

6.3.4 Reflection Algorithm

During the reflection step, modules transmit their location and orientation to the GUI running on the user's external computer. Reflection is only necessary when virtually sculpting the initial block of material. (In fact, the significantly slow down the duplication process.) It allows the GUI running on the user's PC to visualize the physical configuration of the hardware modules. By pointing and clicking on the modules in the GUI, the user can select which modules should become part of the goal structure and which should self-disassemble.

Each module sends a reflection (REF) message immediately after it is localized. The REF messages propagate back to the root module by following a set of *parent pointers*. A module's parent pointer is assigned during the localization process. In particular, a module's parent pointer indicates the face on which the module first received a position (POS) message from a neighbor. Because modules only reply to incoming LOC messages with POS messages after they are localized, any module sending a POS message to its neighbor is guaranteed to have its own valid parent

pointer. By following this chain of parent pointers, all REF messages eventually propagate back to the root module. For more details about parent pointers, consult [34].

In a collection of n modules, there are n unique REF messages that must propagate back to the root module. The total communication cost of the reflection process scales as $O(n^2)$. The worst case scenario arises in a line of modules. The REF message sent by the m-th modules must traverse through $m-1$ other modules to reach the root. Summing the number of hops traversed by all REF messages, the total communication cost is $O(n^2)$.

The time required for all REF messages to reach the root is dependent on several variables. Given a line of modules, the REF messages can move in lock-step. As a module transmits its message to its neighbor closer to the root, it can immediately accept the next incoming REF message from its opposite neighbor. This implies a linear relationship between between the reflection time and the distance between the root and the most distant module. In squares and cubes of modules, this relationship is not so exact.

The root module, and other modules in its proximity, become choke points through which all REF messages must pass. With these modules near the root receiving REF messages from all directions simultaneously, they cannot forward the messages to their parent modules as quickly as they arrive. A traffic jam is created, and the running time degrades from the ideal linear relationship. The other factor affecting the running time of the reflection process is the fact that not all modules send REF messages simultaneously. Each module sends a REF message as soon as it is localized, but modules nearer the root localize before those that are far away. In total, these factors lead us to expect a running that that is roughly linear, but may be worse.

6.3.5 Reflection Experiments

As part of the localization experiments in Section 6.3.3, we allowed each trial to continue to run after the modules had been localized. Each module, after localizing, sent a REF message that propagated back to the root module and from there to a terminal emulator running on our desktop computer. In software, all 583 trials were successful. In hardware, we saw 21 failures in the 256 trials. A failure is defined by a single REF message that does not arrive at the root. In total there

should have been 2533 REF messages transmitted back to the root. The 21 lost messages represent a 0.83% loss rate.

Figures 6-14, 6-15, and 6-16, characterize the number of individual inter-module messages exchanged in the simulated system as the REF messages propagate back to the root module. All plots show the expected $O(n^2)$ dependency on the total number of modules in the system. Note that quadratic nature of the three plots decreases from the line (Figure 6-14) to the square (Figure 6-15) to the cube (Figure 6-16). This behavior is explained by the fact that modules arranged as a cube are, on average, closer to the root than are modules arranged as a line. Consequently, the average distance traveled by each REF message is less.

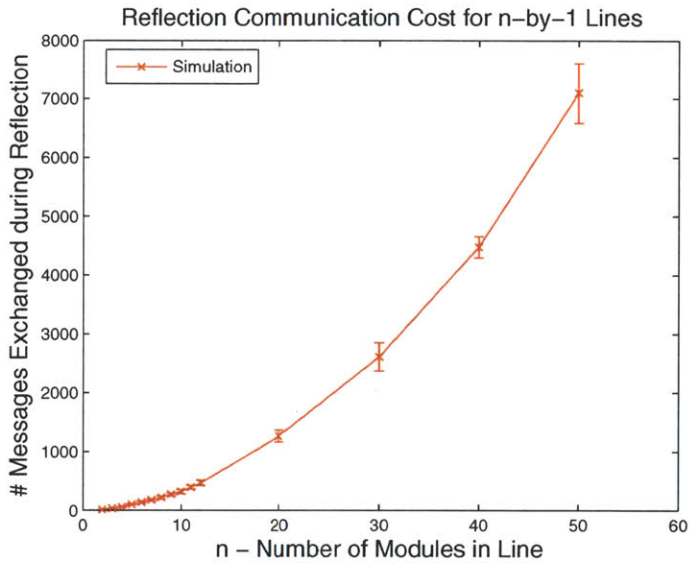


Figure 6-14: In a line of n modules, there are n unique REF messages that must propagate back to the root. The path they must take is $O(n)$ modules long, so the total communication cost, that is the number of individual inter-module messages, scales as $O(n^2)$. The bars on each data point indicate one standard deviation.

Figures 6-17, 6-18, and 6-19 show the time required for the root module to receive all REF messages in lines, squares, and cubes, respectively. The correlation between the hardware and simulator is particularly high. As expected, the running time is roughly linear. There is some degradation from this ideal relationship that is especially noticeable in the case of cubes (Figure 6-19).

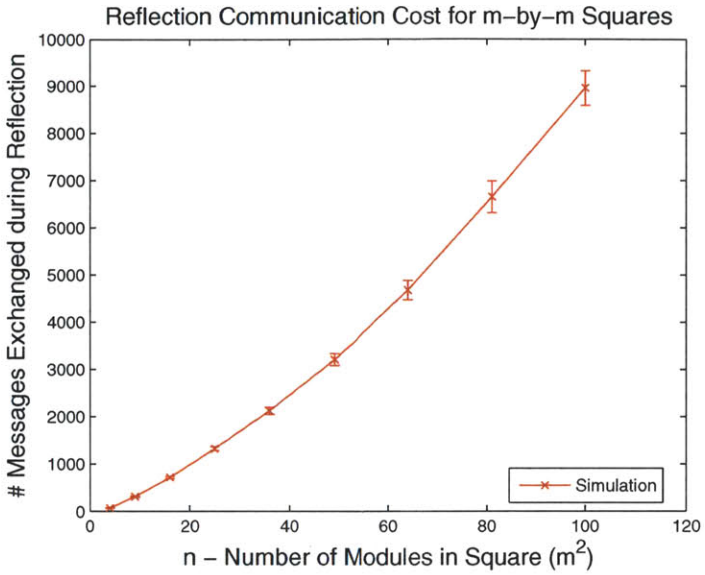


Figure 6-15: In a square sheet of n modules, there are n unique REF messages that must propagate back to the root. The path they must take is, on average, $O(n^{1/2})$ modules long, so the total communication cost, that is the number of individual inter-module messages, scales as $O(n^{3/2})$. The bars on each data point indicate one standard deviation.

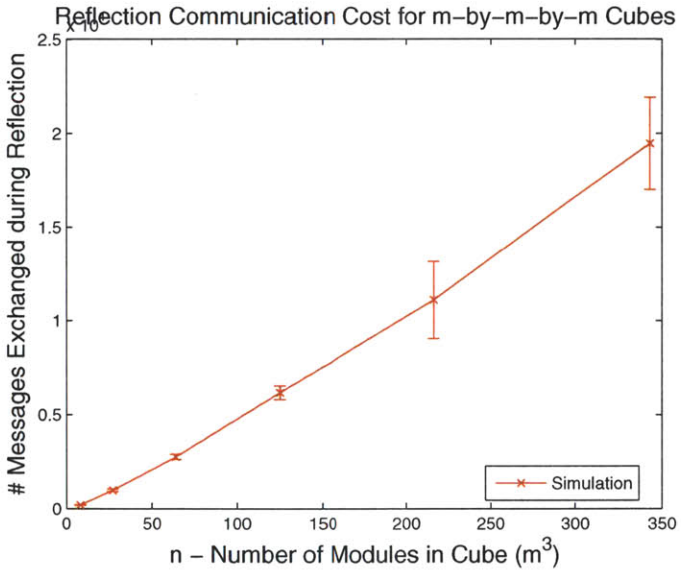


Figure 6-16: In a cubic block of n modules, there are n unique REF messages that must propagate back to the root. The path they must take is, on average, $O(n^{1/3})$ modules long, so the total communication cost, that is the number of individual inter-module messages, scales as $O(n^{4/3})$. The bars on each data point indicate one standard deviation.

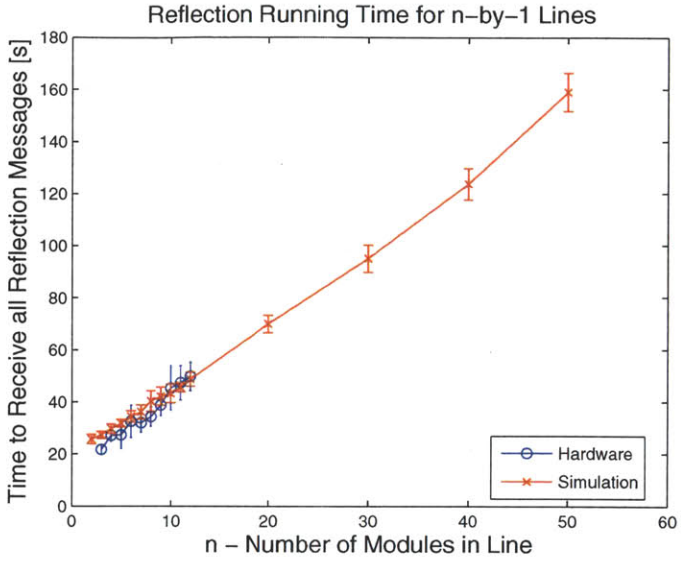


Figure 6-17: In a line of modules, there are $O(n)$ reflection (REF) messages that must propagate back to the root module. These messages can move in lock-step so that the total time for all to reach the root depends only the distance between the distance between the most distant module and the root. The bars on each data point indicate one standard deviation.

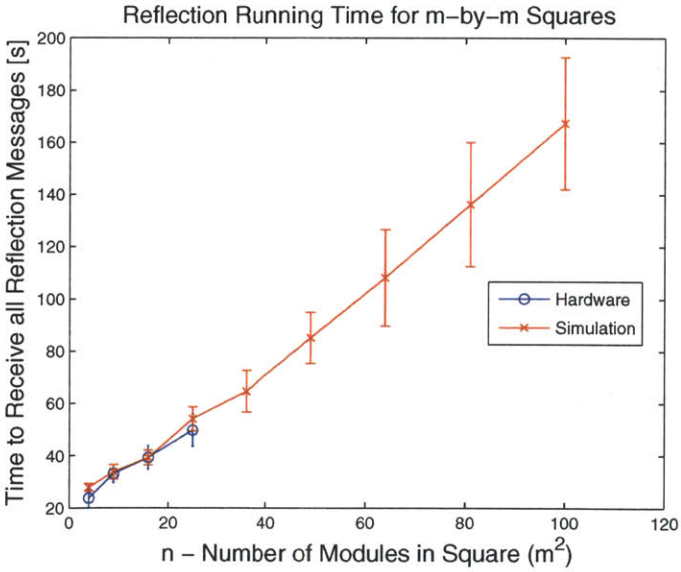


Figure 6-18: The average reflection time in square sheets scales roughly as $O(n)$, but as explained in Section 6.3.4, this relationship is not guaranteed. The bars on each data point indicate one standard deviation.

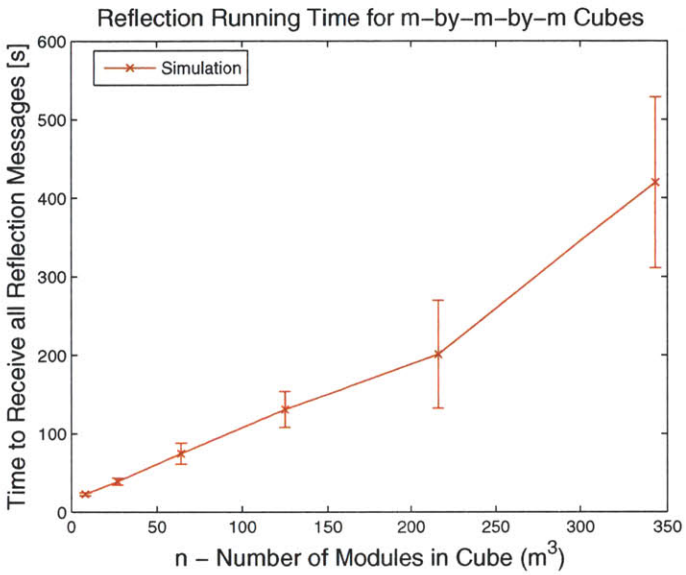


Figure 6-19: The average time required for all reflection (REF) messages from a cubic block of n modules to reach the root scales as n^k where $k > 1$. The root module becomes a choke point which cannot transmit messages to the external PC as quickly as it can receive messages from its neighbors. As a result, the time required to transmit all REF messages degrades from its ideal $O(n)$ bound. The bars on each data point indicate one standard deviation.

Figure 6-20 illustrates the reflection time for 12-module rectangles with different aspect ratios. The left-most data point corresponds to a 4-by-3 module rectangle, the middle point to a 6-by-2 rectangle, and the right most a 12-by-1 rectangle. The plot demonstrates that even though the number of modules remains fixed, their arrangement plays a large role in the reflection time.

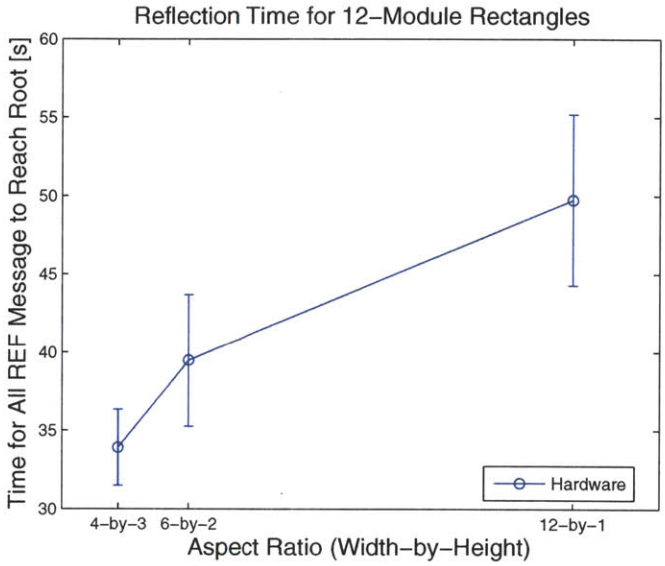


Figure 6-20: As a rectangular arrangement of modules approaches a square (left-most data point), the time required for all REF messages to reach the root is minimized. The bars on each data point indicate one standard deviation. Each data point is averaged from 15 trials.

6.4 Shape Distribution Algorithm

In this section we present an algorithm that controls and optimizes the formation of multiple shapes by sculpting an initial block of connected material. While prior work [34] has shown that self-disassembly can form a particular shape from an initial block of material, the previous algorithm was only able to form a single shape during each iteration of the self-disassembly process, and the resulting shape had to include a unique root module. The algorithm developed in this thesis removes these restrictions. Multiple shapes that are contiguous or separated by any number of unused modules can now be formed. This flexibility allows the sculpting of objects with interlocking

sub-parts and internal degrees of freedom.

The shape distribution algorithm operates by transmitting a single inclusion (INC) message to each module in the initial structure that is destined to be a part of a goal shape. Modules not included in any goal shape do not receive an INC message. Modules assume, by default, that they are not included in the final structure. They will wait forever to receive an INC message. Consequently, once the user knows that all INC messages have been delivered to the modules in the goal shapes, he must explicitly start the self-disassembly process. INC messages originate from the sculptor's desktop computer, pass through the test fixture, and, once in the structure, create and follow a dynamic *inclusion chain*. This inclusion chain is constructed from a constant amount of information per message, and it grows in length with each additional INC message. The algorithm avoids encoding the detailed path that each inclusion message must follow, and it avoids flooding the system with messages.

The total communication cost of the inclusion chain algorithm is $O(n^2)$ where n is the number of modules included in the final structure. This bound arises because for each of the n modules, the INC message that informs each module of its status may have to travel from the root module through $O(n)$ other modules. In contrast, using a shortest-path algorithm to route a message from the root to each included module also has a theoretical communication cost of $O(n^2)$ but only if a gradient descent approach is employed and there are no obstacles in the structure that could form local minima. Once one considers broken inter-module communication links and voids within the initial structure, the communication cost of the routing algorithm increases as each message must contain more specific routing instructions. Given the uncertainty over which approach will perform better on average, we choose the inclusion chain approach for its simplicity given the hardware's limited processing capabilities.

INC messages are generated by the system's user, often with the help of a GUI. All INC messages, like all other messages, enter the initial block of modules through the root module's serial connection to the user's desktop computer. As an inclusion message moves from a module to its neighbor, it extends the tail of an *inclusion pointer chain*. Figure 6-21 shows how inclusion messages follow this chain for a specified distance termed the *hop count*. Once a message has traveled

the specified number of hops, it branches off of the chain in the specified *branch direction.* The hop count and branch direction are pieces of information carried by the message itself—they do not come from the modules in the structure. However, the modules in the structure do store the inclusion pointer chain. Each module only needs to remember where to redirect an incoming INC message with a hop count greater than one.

After branching, the old inclusion pointer chain may be lengthened, or it may be truncated and redirected. The module that the message reaches after branching off of the inclusion pointer chain is included in the structure. If the old chain is truncated, the modules in the discarded portion of the chain maintain their pointers, but do not affect the shape formation process.

INC messages carry additional information. First, each message contains an *ignore* field which may be used to counteract the message's typical effect at its destination module. This module, instead of assuming to be included in the final structure, effectively ignores the INC message. The advantage is that a module may be part of the inclusion pointer chain without being a part of the final shape. This allows the formation of an unlimited number of disjoint shapes from one initial block of material during a single self-disassembly process.

The second auxiliary piece of information carried by an inclusion message is the *group number.* When an INC message reaches its destination, the group number is assigned to the module. During the disassembly phase, if two included modules have different group numbers, they disconnect from each other. Likewise, if their group numbers are identical, they remain bonded. Group numbers will allow the formation of contiguous interlocking shapes.

In practice, INC messages are ASCII strings:

$$\texttt{INC},\langle \textit{hop count} \rangle,\langle \textit{branch dir.} \rangle,\langle \textit{ignore} \rangle,\langle \textit{group} \rangle.$$

Each module employs Algorithm 10 when processing an incoming INC message. When a module receives an INC message, it first checks the hop count (line 6). If that value is 0, the receiving cube is the intended destination. In addition, if the ignore flag is not set, the module records the fact that it should be a part of the final structure and saves the group number included in the message (lines 7–8). A hop count of 1 indicates that one of the receiver's immediate neighbors is the message's intended destination. The receiving module uses its own known rotation and the

message's branch direction to determine the face that should retransmit the message after the hop count is set to 0 (line 10–11). This inclusion pointer direction is stored as part of the module's state (line 12). Finally, if a module receives an INC message with a hop count greater than 1, it decrements the hop count and then retransmits the message on the face indicated by the previously stored inclusion pointer direction (line 14). The algorithm terminates when the module receives a disassemble (DIS) message.

Algorithm 10 parseIncMsg—Inclusion Message Processing Algorithm

```
 1: included = false
 2: incChainPtr = NULL
 3:
 4: repeat
 5:     wait for INC msg. w/ hop count HC, branch dir. BD, ignore flag IGN, and group number GRP
 6:     if HC = 0 and IGN = 0 then
 7:         included = TRUE
 8:         myGroup = GRP
 9:     else if HC = 1 then
10:         txFace = branchDirToFace(BD)
11:         queueINC(txFace, ∞, 0, BD, IGN, GRP)
12:         incChainPtr = BD
13:     else
14:         queueINC(incChainPtr, ∞, HC − 1, BD, IGN, GRP)
15:     end if
16:     txQueuedMsgs()
17: until DIS message received
```

The `queueINC` function on line 11 builds an inclusion message and places it in the transmit queue of the specified face using the `fillTxBuf` function from Chapter 5. The second argument to the function specifies the number of attempts that the system should make to transmit the message before giving up.

Proving the correctness of the shape distribution algorithm requires a description of the shape one wishes to form. In our system, generating a description of the goal shape is facilitated by a GUI that allows the user to virtually sculpt the desired shape and then generates a list of inclusion messages that are transmitted to the root and distributed. In [34] we show that this approach is efficient and correct. While the prior proof did not incorporate the concept of ignored modules, their effect is negligible, and we will not repeat the proof here. Additionally, the group code carried in each inclusion message has no effect on the algorithm.

Figure 6-21 illustrates the propagation of eight inclusion messages as they form a simple wrench from a 3-by-4 block of material. As indicated by the text above the modules in Figure 6-

21(a), the first inclusion message is `INC,0,n/a,true,0`. The second inclusion message reaches module A with a hop count of 1, indicating that one of A's neighbors is to be included. The branch direction of this message is "up," so the hop count is decremented to 0 and the message is sent to module E. Module A sets its inclusion chain pointer to E. The third message reaches A with a hop count of two, follows A's inclusion chain pointer to E, and obeys the message's branch direction by moving to the right and including module F. Jumping ahead, after Module L is included as shown in (f), the next inclusion message modifies module G's inclusion chain pointer from "up" to "down" to include C. Module K's inclusion chain pointer still points to module L, but the inclusion of modules C and D is unaffected.

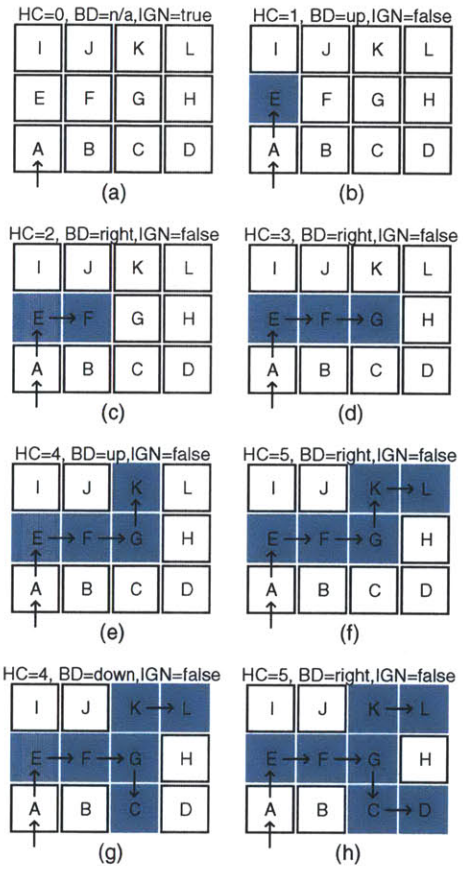


Figure 6-21: Eight inclusion messages are used to create a simple wrench from a 3-by-4 block of programmable matter. The root module is labeled A. As modules are included in the final structure, they change from transparent to shaded. The arrows in the figure represent the inclusion chain pointers stored in the modules.

6.5 Self-Disassembly Algorithm

The key step for shape formation with the Smart Pebbles is the inter-module disconnection process that must occur after all modules know whether to remain as part of a finished object or to disconnect completely. The challenge of this disconnection process is that a module loses its ability to function once it breaks its mechanical connection with the neighbor supplying it with power. Furthermore, all modules that are dependent on that module for power will also lose power and will not be able to break additional magnetic bonds.

6.5.1 Parents, Children, and Neighbors

A tree can be used to represent how power is transmitted through an initial block of modules. Because it is connected to an external power supply, the module connected to the user's PC, is the root of this power transfer tree. Every other module in the tree has one parent, P. This parent is the neighbor that supplies the module with power. Conversely, every module to which a module supplies power is a child. Children of a module are denoted by the set C. Parents and children are both subsets of a module's magnetically bonded neighbors, N. In practice, current often follows many different paths from the root to any other implying that a module should have multiple parents. We disallow multiple parents by definition because they only serve to complicate the disassembly process. The key concept is that although different neighbors could also supply it with power, a module will never lose power so long as it is connected to its parent. These child and parent relationships are defined during the assembly process. A module is not allowed to become the parent of another until it has a parent of its own.

6.5.2 Child-to-Parent Disconnection

We have designed and implemented Algorithm 11 which ensures that an initial block of material can disassemble correctly; that is, disconnecting bonds that should be broken and keeping those that should be preserved. In general, the disconnection algorithm operates by ensuring that a module has no children before disconnecting from its parent. If a module is a part of the same

finished shape as its parent, the child uses a child removal message to inform its parent that it no longer needs to be considered a child. The algorithm uses sets N, P, and C to keep track of a module's bonded neighbors, parent, and children, respectively. We use two additional sets, G and K, that are initially empty. All neighbors from which a module has received group (GRP) messages are added to G. If a neighbor's group matches the receiver's, the neighbor is added to the *keep* list, K.

Algorithm 11 selfDisassemble—Ensure that the self-disassembly process is organized so that modules do not lose power before they have broken all mechanical bonds with their neighbors.

Require: N: set of neighbors
Require: C: set of children, $C \subset N$
Require: P: single-element set containing parent, $P \subset N$
Require: G: set of neighbor from which module has received GRP msgs.
Require: K: set of neighbors with which to retain module's bonds

```
 1: G = K = ∅
 2: wait for DIS msg. to be rcvd. on face rxFace
 3: queueDIS(C \ rxFace, ∞)
 4: queueDIS(N \ (rxFace ∪ C), DIS-RETRIES)
 5: repeat
 6:    txQueuedMsgs()
 7: until txQueueIsEmpty(N \ rxFace)
 8:
 9: if included then
10:    queueGRP(N, ∞, myGroup)
11:    repeat
12:       if GRP msg. rcvd. (on face rxFace specifying a neighbor in group neighborGroup) then
13:          G = G ∪ rxFace
14:          if neighborGroup = myGroup then
15:             K = K ∪ rxFace
16:          else if rxFace ≠ P then
17:             queueUnlatch(rxFace, ∞)
18:          end if
19:       end if
20:       txQueuedMsgs()
21:    until txQueueIsEmpty(N) and N = G and N = (P ∪ K)
22:    if myGroup ≠ parentGroup or parentGroup = ∅ then
23:       queueUnlatch(P, ∞);
24:    else
25:       queueCLD(P, ∞);
26:    end if
27: else
28:    queueUnlatch(N \ (C ∪ P)), ∞)
29:    queueGRP(C, ∞, myGroup)
30:    repeat
31:       txQueuedMsgs()
32:    until N = P
33:    queueUnlatch(P, ∞)
34: end if
35: repeat
36:    txQueuedMsgs()
37: until N = ∅
```

The algorithm begins by waiting for a disassemble (DIS) message from some neighbor. The

face on which the message arrives is represented by the single-element set *rxFace*. When the module receives a DIS message, it forwards the message to its children (line 3). If the children do not receive this message, there is no guarantee that they will receive a DIS from any other source. In line 4, the DIS message is also sent to the module's neighbors that are not children to speed its propagation throughout the structure. To prevent two modules from repeatedly sending DIS messages to each other, a DIS message cannot be sent back to the module from which it was received. After the DIS messages that are to be transmitted have been queued, we continue attempting to retransmit them until the transmit queues for all neighbors are empty (line 7). By passing infinity to `queueDIS` in line 3 when the algorithm queues the DIS messages for the module's children, the algorithm ensures that the `txQueuedMsgs` function will never stop attempting to deliver the message until it is successful. This guarantees that module's children receive the DIS message before the algorithm moves past line 7. In contrast, the DIS-RETRIES parameter in line 4 indicates that the `txQueuedMsgs` function only makes a finite number of attempts to send the DIS message to the module's non-child neighbors before the `txQueueIsEmpty` returns true. Once the children have received the DIS message, the algorithm branches (line 9) depending on whether the module is included in any of the final structures being formed.

If the module is not included in the final structure, the relevant pseudo-code begins on line 27. The algorithm begins with the module queuing an unlatch message for all of the module's neighbors except the module's children and parent. Then, in line 29, the module queues group (GRP) messages for its children. Group messages simply inform their recipients of the transmitter's group. The infinity parameters passed to the `queueUnlatch` and `queueGRP` functions in lines 28 and 29, would normally indicate that all of the unlatch and GRP messages will be repeatedly transmitted until successfully received, but the receipt of an unlatch message purges the corresponding transmit queue; there is no point in continuing to transmit a message to a neighbor that is no longer present. (This behavior is not shown in the pseudo-code.) Now that the unlatch and GRP messages are queued, the algorithm continually transmits them (line 30–32) until the module's only remaining neighbor is the module's parent. This elimination of neighbors results from the pseudo-code on line 33. Once a module's only neighbor is its parent, the module queues an unlatch message for the

parent and waits (lines 35—37) until the message is successfully transmitted. When it is, the parent is removed from the module's list of neighbors, indicating that the module is now completely disconnected.

Alternatively, if the module is included in the final structure, it behaves differently. Lines 11–21 of Algorithm 11 form a repeat-until loop that eliminates all of a module's neighbors (except its parent) with group numbers that do not match its own. Before the loop begins in line 10, the algorithm queues GRP messages for all neighbors, including the module's parent and children. The infinity parameter in line 10 ensures that these GRP messages are sent repeatedly by the *txQueuedMsgs*() function until they are received. Once the loop beings, the algorithm checks for any incoming GRP messages from its neighbors (line 12). If one is received, the transmitting neighbor, *rxFace*, is added to G, the list of neighbors from which the module has received GRP messages. If the GRP message indicates that the neighbor's group is the same as the module's (line 14), then that neighbor is added to the module's keep list, K (line 15). If the neighbor's group number differs from the module's, and if the neighbor is not the module's parent, the module queues an unlatch message for the neighbor in line 17. This unlatch message overwrites any pending GRP message destined for that neighbor.

This process of transmitting and receiving GRP messages will eliminate all of a module's neighbors other than its parent and the neighbors in K. The loop ends in line 21, when the transmit queues of all neighbors have been emptied, the module has received a GRP message from each of its remaining neighbors, and its only remaining neighbors are $P \cup K$.

The module's children are eliminated over the course of the repeat-until loop in lines 22–26. To consider the disconnection process complete, the module only needs to inform its parent that it is no longer the parent's child. Exactly how the module informs its parent is determined by line 22. If the module's group is different than its parent's, (or if its parent does not belong to a group because it is not included in the final structure), the module queues an unlatch message for its parent. When this message is received, the two modules disconnect and the parent no longer considers the module its child. Alternatively, if the module and parent share the same group, the module sends a child removal (CLD) message to its parent. This message informs the parent that

the module has performed all necessary tasks and no longer requires a power source. As a result, the parent removes the module from its list of children, C. In this manner, the parent will eventually be left with no children so that it can sever the bond with its own parent.

6.5.3 Disconnection in Action

Figure 6-22 shows Algorithm 11 in action. In the figure, (a) represents the state of the modules after the DIS message has been distributed and the modules have exchanged GRP messages, but before any have begun to disconnect from their neighbors. The color of each module indicates the group to which it belongs. Module C is not included in any of the final structures. As shown by the transition from Figure 6-22(a) to (b), disconnection begins when the modules without children (E, F, and H) sever the relationships with their parents. In the case of E, its parent belongs to the same group, so it sends a CLD message that breaks the parent relationship while maintaining the physical bond. Module F belongs to the same group as its neighbor, G, so G is in F's keep list. Given that all modules have already exchanged GRP messages, F's state satisfies the conditional in line 21 of Algorithm 11. Consequently, F executes line 23 of the pseudo-code and transmits an unlatch message to its parent. Module H disconnects from its only non-parental neighbor, module D, because they are in different groups.

In subfigure (b), module D has no remaining neighbors except its parent, module C, which is not included in the structure, and therefore lacks a group code. Module D therefore satisfies the condition on line 22 of Algorithm 11, and it sends an unlatch message to C to disconnect from its parent. Without any remaining connection to the structure, the shape formed by modules D and E loses power in subfigure (c). Also shown in the transition from (b) to (c), module H sends its parent, G, a CLD message leaving G without children.

As soon as G has no children, it disconnects from its parent because it is not included in the structure. After disconnecting, the shape formed by modules F, G, and H loses power. This disconnection is the only change as the system transitions from Figure 6-22(c) to (d). Once in the state shown by (d), module C realizes that it now has no children and no neighbors except its parent. Because it is not included, C can disconnect from its parent.

In Figure 6-22(e), module B has no children and no neighbors other than its parent, allowing B to send a CLD message to its parent, A. Module B sends a CLD message instead of an unlatch message because it knows that A is a part of the same group. When A receives this message, the parent-child bond between A and B is broken, transitioning the system to the state shown in subfigure (f). Finally, module A is left with only its parent, so A symbolically disconnects from the user's desktop computer. At this point, all modules have lost power, but all of the necessary bonds have been maintained, and the desired shapes have been formed.

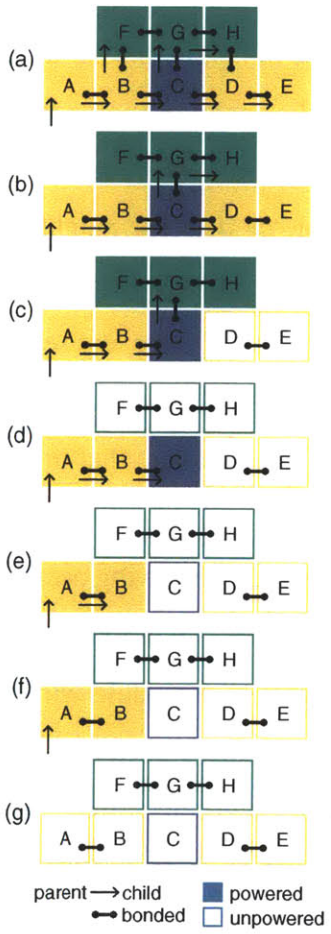


Figure 6-22: Disconnection occurs in an orderly fashion. Each color of module in the figure represents an object that is to be formed from the initial block of material. As modules disconnect from the structure and lose power, they change from filled to empty. Before disconnecting, a module must ensure that all of its neighbors that depend on it for power have completed their disconnection process. Module A is the root.

6.5.4 Correctness

The correctness of Algorithm 11 can be proven using induction on the height of the power transfer tree.

Theorem 3. *Algorithm 11 results in a neighbor disconnection order that maintains power in each module until it has finished disconnecting from all unincluded neighbors and neighbors with group numbers different from its own.*

Proof. Base case: Tree height 1. A tree of height one has a single parent and multiple children that are the leaves of the tree. These children may or may not be magnetically connected neighbors. If a child has magnetically connected neighbors, it exchanges GRP messages with them. If the groups of two neighbors are different, or if either neighbor is not included in any of the final structures, the neighbors unlatch (line 17). If they are in the same group, they do nothing. Once a leaf module handles its neighbors appropriately, the leaf severs its parent-child bond with the root. If the root and leaf are in different groups, the leaf sends the root an unlatch message (lines 22–23). Otherwise, the leaf sends the root a CLD message that breaks the parent-child connection while maintaining the magnetic bond.

As we set out to prove in the theorem, the following occurs for each module before it potentially loses power by disconnecting from its parent: an unincluded module completely disconnects from all neighbors and then its parent; an included module with magnetically connected neighbors in groups other than its own detaches from these neighbors; and simultaneous with power loss, a module disconnects from its parent if their groups differ.

Induction: Assume the disconnection process operates correctly for trees of height n. To complete the proof, we need to show that the disconnection process works correctly for trees of height $n+1$. Following this approach, a tree of height $n+1$ can be viewed as a tree of height n with one additional set of leaves. These leaves may or may not be magnetically bonded with any other module in the entire tree. Whether or not they are bonded does not change how they act. Just as in the height-1 base case, the leaves exchange GRP messages with their magnetically bonded neighbors and break their magnetic connection if they are in different groups or if either is not included in the

final structure. Once the leaves have broken all bonds except those they share with their parents, the leaves break their parent-child bonds (by unlatching or sending a CLD message).

As in the base case, all leaves, before losing power, have disconnected from their neighbors as needed. Unincluded leaf modules have broken all of their magnetic connections. Included leaf modules have broken their magnetic connections with neighbors belonging to groups different than their own and maintained their connections with neighbors of the same group. With the leaves removed, the $n+1$ height tree is now an n height tree. □

6.5.5 Self-Disassembly Running Time Experiments

We characterized the running time and communication cost of self-disassembly following the same procedure as in Sections 6.3.3 and 6.3.5, which we used to characterize the localization and reflection algorithms. After receiving all REF messages, we issued a disassembly (DIS) message instructing the system to break all inter-module bonds. Starting as we issued the DIS command, we measured the time required for the system to break all bonds.

We performed 154 hardware trials on 3- to 12-module lines and 63 trials on 2-by-2- to 5-by-5-module square sheets of modules. In hardware, the self-disassembly algorithm operated correctly in 202 trials, or 93.1% of the time. If a single pair of modules failed to break their shared bond, we marked the trial a failure, so the percentage of all bonds that were correctly broken was actually much higher. In simulation, we performed 583 trials with lines, square sheets, and cubic blocks. All of these trials worked correctly. This leads us to believe that the algorithm is working correctly, but its robustness could be improved. One particular problem is that, due to variations in the size of the Smart Pebbles, the modules realign as mechanical bonds are broken. The result is that some communication links fail during the self-disassembly process. This can make it impossible to correctly complete the self-disassembly process.

Figures 6-23, 6-24, and 6-25 illustrate that the running time of the self-disassembly algorithm is $O(n)$. This bound is only tight in the case of lines of modules. In lines, a module must wait for all modules farther away from the root to disconnect before it can disconnect from its neighbor closer to the root. The root module must wait for $n-1$ other modules to disassembly before it can do

so. Because each module requires $O(1)$ time to disassembly once its parent is its only remaining neighbor, the overall running time of the self-disassembly process is $O(n)$. Figure 6-26 illustrates that, for a fixed number of modules, arrangements that more closely approximate a cube will self-disassembly more quickly than arrangements with large aspect ratios. In arrangements other than a line, the many modules can self-disassembly in parallel thereby reducing the running time of the algorithm. In particular, the self-disassembly time is linearly proportional to the distance between the root and the most distant module (see Figure 6-27).

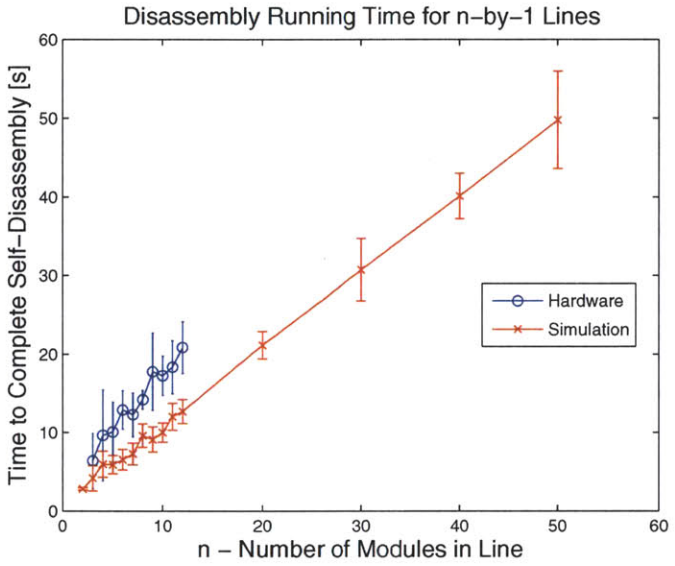


Figure 6-23: The time required for all modules in a line to self-disassemble scales linearly with the length of the line. The bars on each data point indicate one standard deviation.

We recorded the number of messages exchanged during the self-disassembly process using the Sandbox simulator presented in Chapter 4. The self-disassembly process is initialized by broadcasting a single DIS message to all modules in the structure, a task that requires $O(n)$ messages. Then, each of the n modules must exchange $O(1)$ group (GRP) and child (CLD) messages with its neighbors. Therefore, the total number of messages exchanged during the self-disassembly process is $O(n)$. This theoretical bound is confirmed by Figures 6-28, 6-29, and 6-30 which all illustrate a linear relationship between the number of modules and the number of messages exchanged during self-disassembly.

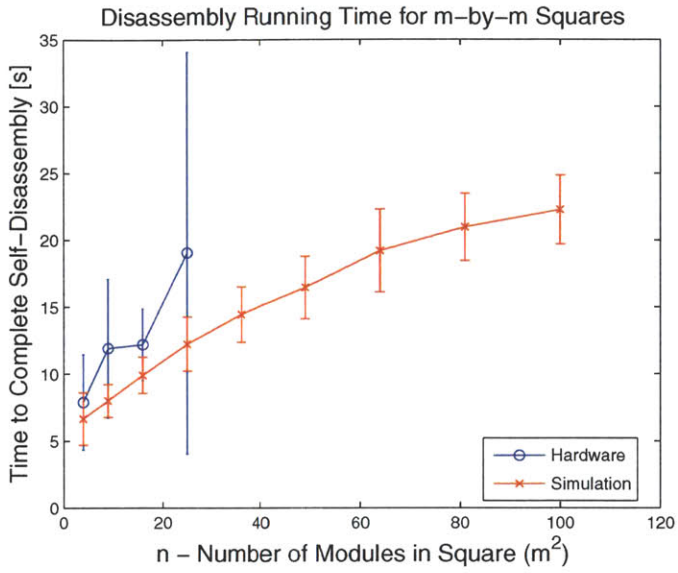


Figure 6-24: The time required for all modules in square sheet to self-disassembly obeys an $O(n)$ bound, where n is the number of modules in the square. The bars on each data point indicate one standard deviation.

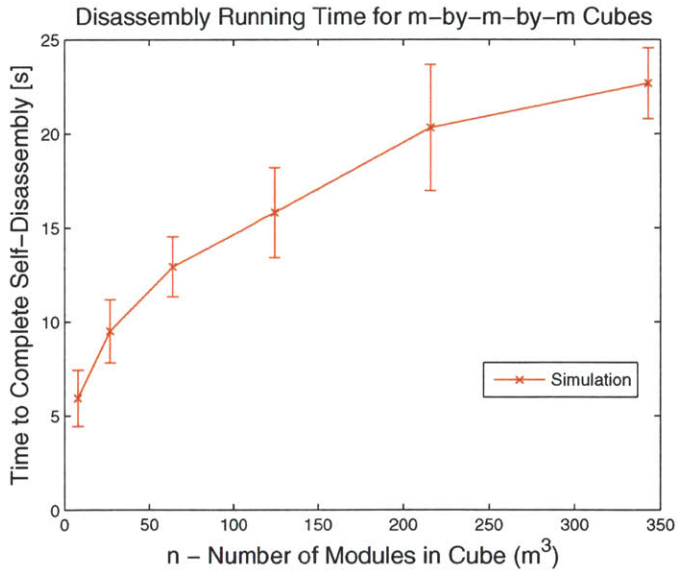


Figure 6-25: The time required for all modules in a cubic block to self-disassembly is less than the theoretical $O(n)$ bound because many bonds can be broken simultaneously. The bars on each data point indicate one standard deviation.

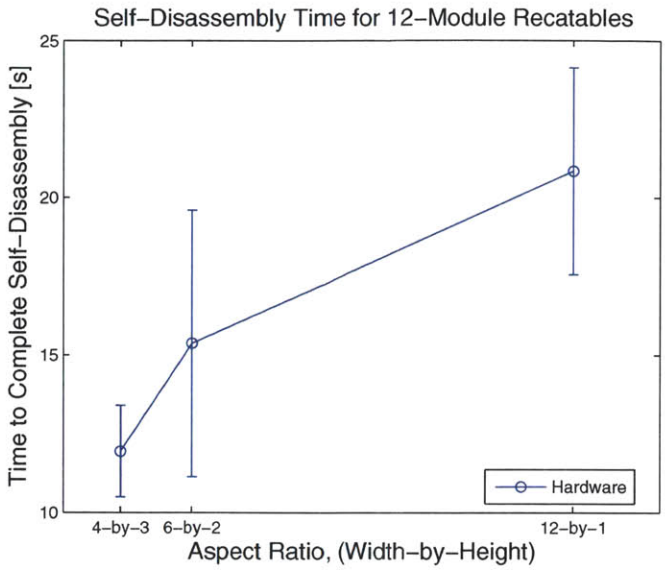


Figure 6-26: As a rectangular arrangement of modules approaches a square sheet, the self-disassembly process runs more quickly because many bonds can be broken in parallel. The bars on each data point indicate one standard deviation. Each data point is averaged from 15 trials.

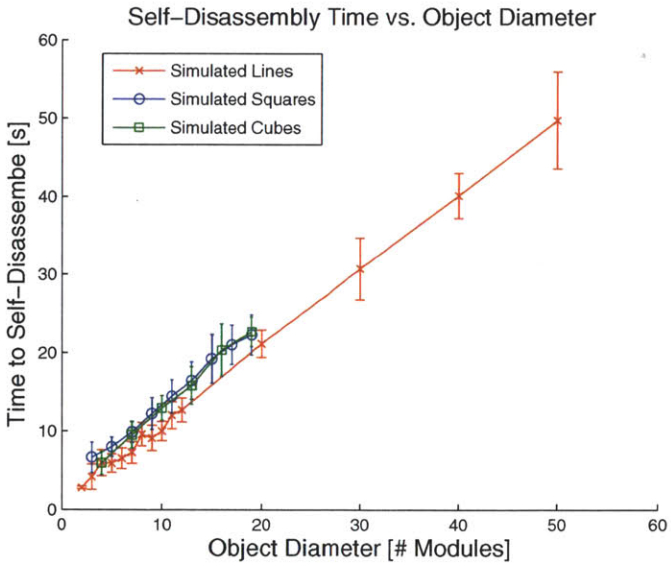


Figure 6-27: The self-disassembly time of a group of modules varies linearly with the distance between the root and the most distant module. When the root is on the perimeter of the collection of modules, (as it is in all of our experiments), this distance is the diameter of the object. The bars on each data point indicate one standard deviation.

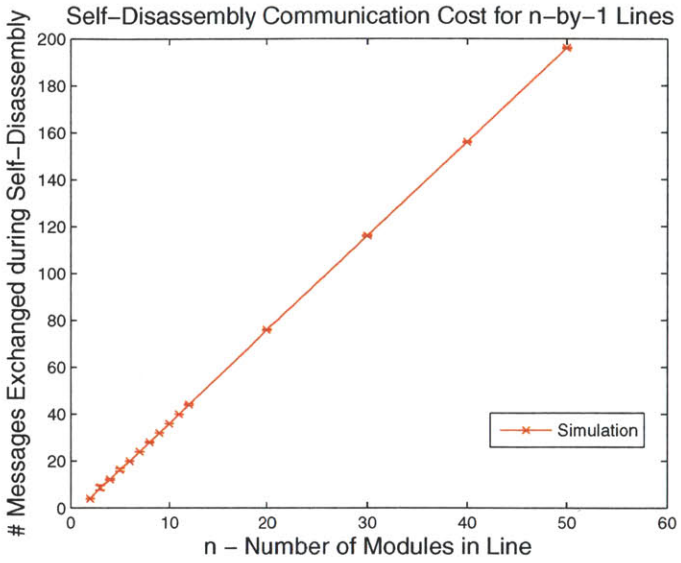


Figure 6-28: In a line of n modules, $O(n)$ messages must be exchanged during the self-disassembly process. The bars on each data point indicate one standard deviation.

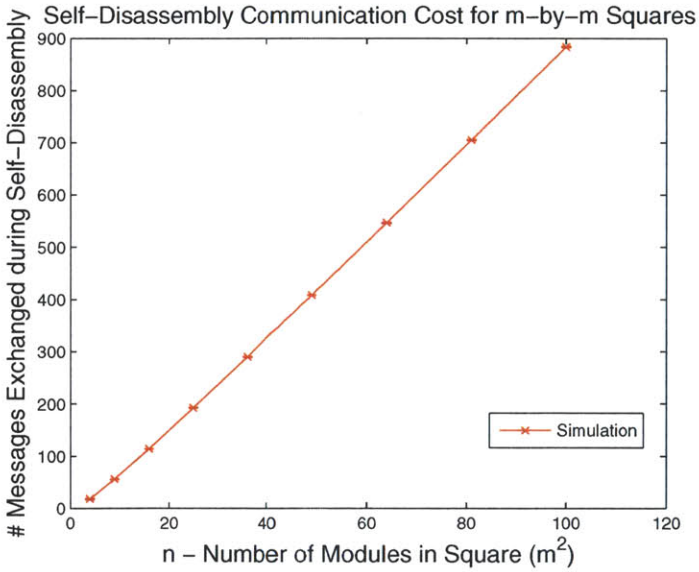


Figure 6-29: In a square sheet of n modules, $O(n)$ messages must be exchanged during the self-disassembly process. The bars on each data point indicate one standard deviation.

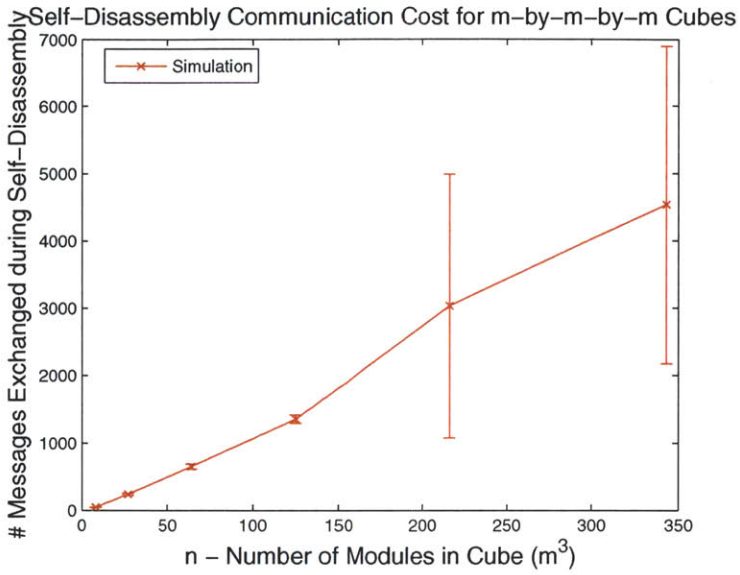


Figure 6-30: In a cubic block of n modules, $O(n)$ messages must be exchanged during the self-disassembly process. The bars on each data point indicate one standard deviation.

6.6 Shape Distribution and Disassembly Experiments

We have performed several end-to-end self-disassembly experiments in both simulation and hardware. In these experiments, we used the sculpting process presented in this chapter to convey the desired shape to the initial block of modules that we assembled by hand. Because the shape distribution and self-disassembly phases are distinct, we are concerned with the success of each. The first experiment we performed consisted of fully disassembling a 3-by-3 block of modules that did not contain any goal shapes. This is shown pictorially in Figure 6-31(a). In 12 of 15 hardware experiments, all bonds were broken as expected. In the other 3 three, there were 2, 3, and 4 unbroken bonds. In all three cases, the initial shape was poorly constructed and the modules far from the root did not align well with their neighbors. As a result, we believe communication failures, not the algorithm, led to the unbroken connections.

We performed 67 additional experiments with other goal shapes to test the system's ability to use the ignore and group fields of an inclusion message. The most complex experiment formed 6 different Tetris pieces from a 4-by-7 block of modules. Twenty-four of the 28 modules were in-

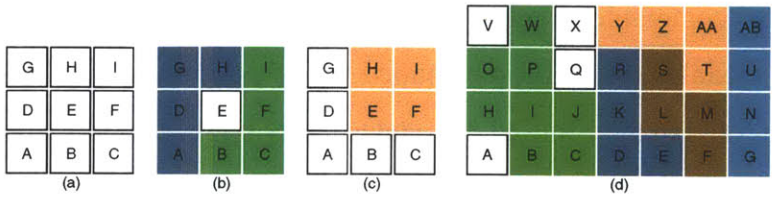


Figure 6-31: We have used the Smart Pebbles to form a number of different shapes that test the ability of the hardware and algorithms to form multiple contiguous and discontiguous shapes.

cluded one of the goal shapes. The results for both the simulations (55 experiments) and hardware (12 experiments) appear in Table 6.1. In the table, the shape distribution success rate is measured by observing which Pebbles know that they should be a part of goal structure. The disconnection success rate is the number of bonds that behaved as expected divided by the total number of bonds in the initial structure

Table 6.1: Experiments show the algorithms working correctly.

Goal Shape(s)	*Sim / HW*	*Number Trials*	*Success Rate [%]*	
			Distribution	*Disconnection*
Figure 6-31(a)	Sim	15	N/A	100.0
	HW	15	N/A	95.0
Figure 6-31(b)	Sim	15	100.0	100.0
	HW	5	100.0	98.3
Figure 6-31(c)	Sim	15	100.0	100.0
	HW	5	100.0	96.7
Figure 6-31(d)	Sim	15	100.0	100.0
Figure 6-21(h)	Sim	10	100.0	100.0
	HW	2	100.0	97.1

The results in Table 6.1 show that the shape distribution algorithm works flawlessly in both simulation and hardware. We only see errors when performing disconnection experiments in hardware. Even so, the overall disconnection success rates are still good. This leads us to believe that the disconnection algorithm functions correctly, but that peculiarities of the hardware are interfering with its operation.

We have observed four particular hardware issues that affect the disconnection process. First, all modules are not exactly the same size. As a result, alignment errors can accumulate, result-

ing in marginal or no communication between neighbors. Second, during the assembly process, previously bonded modules are sometimes pulled out of position as new modules are added to the structure. This results in the connected modules losing power, resetting their states, and introducing inconsistencies in the system. Third, the disconnection process releases internal stresses as some of the magnets turn off. Given that we see the modules moving as they disconnect, we suspect that this may also result in modules temporarily losing power. Finally, the power supply sourcing power to the root module is current limited. When a module deactivates an EP magnet, it momentarily draws 4A. The simultaneous deactivation of many EP magnets during disconnection often pegs the power supply at its current limit, potentially preventing some modules from unlatching.

Chapter 7

Duplication

This chapter explores an alternative to virtual sculpting that we call *distributed duplication.* Distributed duplication operates as follows. A passive object is buried under, or submerged into, a collection of programmable matter modules. Upon receiving a start signal, the all modules mechanically bond with their neighbors to encase the original object in a solid block of material. Once solidified, the modules execute a distributed algorithm that senses the shape of the original object. After the system has captured the shape of the original, it creates one or more, potentially magnified, replicas of the object using the rest of the programmable matter through self-disassembly by selectively unlatching the unnecessary modules from the initial block of material. When this self-disassembly is complete, the user can brush away the newly disconnected modules to reveal a replica of the original object. The algorithm requires $O(1)$ space and exchanges $O(n)$ messages per module in a system with n modules.

7.1 Duplication Algorithms

The distributed duplication algorithm is a multi-step process that is able to sense the shape of a passive object that is surrounded by programmable matter modules and then form a duplicate of that object using additional modules within the same initial block of material. The algorithm is completely distributed, all modules execute the same code, and all computation occurs on-board.

The algorithm, illustrated in Figure 7-1, is composed of five major phases:

1. Encapsulation and Localization
2. Shape Sensing / Leader Election
3. Border Notification
4. Shape Fill
5. Self-Disassembly

In short, after all modules are localized and bond together to encase the object being duplicated, the algorithm senses the border of the original object, creates a duplicate border beside the original, informs all modules inside of this border that they form the duplicate shape, and then prompts all modules except those that form the duplicate shape to self-disassemble. The user can they brush aside the extra modules much like a sculptor would remove extra stone from a block of marble to reveal the newly created duplicate object.

7.1.1 Encapsulation and Localization Algorithm

The shape duplication process begins when the user surrounds the passive object to be duplicated with a collection of programmable matter modules. In a 3D system with sand-sized particles, we envision literally burying the object to the duplicated. Using the 2D, centimeter-scale Smart Pebbles, we can use an inclined vibration table, the 2D analog of a bag of sand, to surround the passive object with active modules. Once the object is surrounded, the user sends a start command to one module to initiate the encapsulation and localization process. The recipient of this message arbitrarily assumes that its coordinates are (0,0), and then it informs all of its neighbors of their coordinates. As each module learns its coordinates within the system, it mechanically bonds with its neighbors to rigidly encapsulate the passive object being duplicated. Once bound to its neighbors, each module enters the shape sensing and leader election phase.

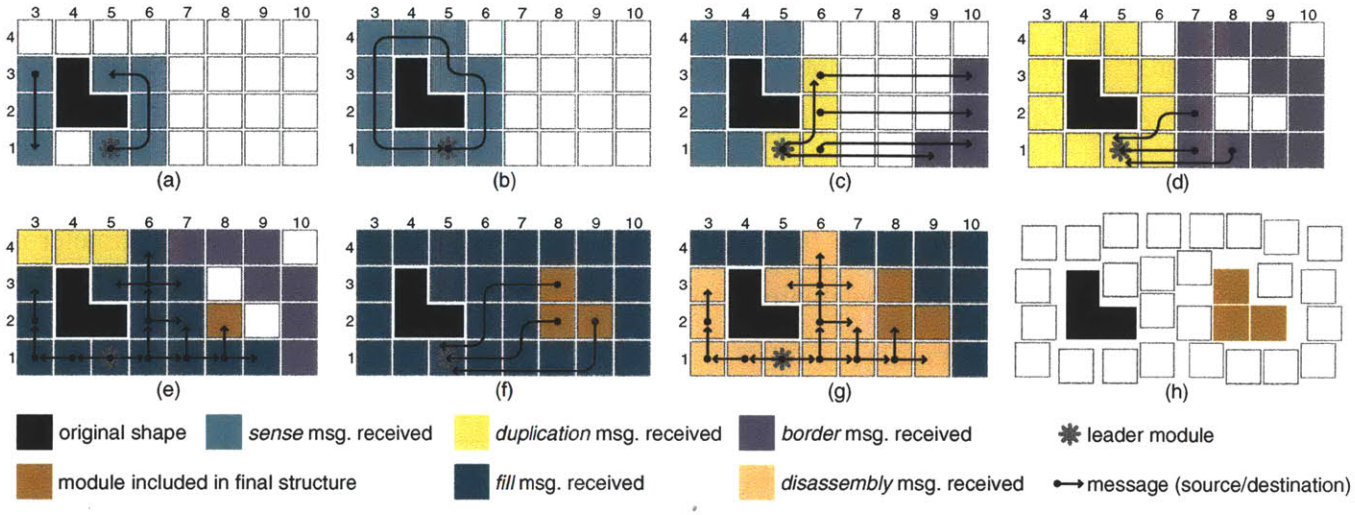


Figure 7-1: After localization, the distributed duplication algorithm begins in (a) by routing a sense (SEN) message around the border of the obstacle. As shown in (b), the message sent by the module with the highest unique ID will eventually return to its sender, prompting that module to route a duplication (DUP) message around the border of the obstacle (c). Upon receiving a duplication (DUP) message, a module sends a border (BOR) message to its conjugate that will become the border of the duplicate object. After all duplicate border modules have sent confirmation (CON) messages back to the leader (d), the leader broadcasts a fill (FIL) message (e) informing modules contained by the new border that they are part of the duplicate shape and causing them to send confirmation (CON) messages back to the leader, (f). Upon receiving all confirmation messages, the leader broadcasts a disassemble (DIS) message (g) causing all modules except those in the duplicate shape to self-disassemble (h). Note: the key for this figure holds for all others in this chapter as well.

7.1.2 Shape Sensing / Leader Election Algorithm

The goal of the sensing phase is to two-fold: determine the perimeter, area, and dimension of the original obstacle's bounding box; and elect a leader module on the perimeter of the object being duplicated. After a module is localized by an incoming position (POS) message, it detects which of its neighbors are present by assuming that unresponsive neighbors are absent. The module assumes that these missing neighbors correspond to the obstacle presented by the original object to be duplicated. Then, a module attempts to route, (using the bug algorithm as explained in Chapter 5), a sense (SEN) message to each of its missing neighbors. Because the destination coordinates are occupied by the obstacle being duplicated, the SEN message will never be delivered to its destination, but this is the intent. Instead, the SEN messages will traverse the entire perimeter of the obstacle being sensed. Eventually, it will return to its sender, who will then know that the message cannot be delivered.

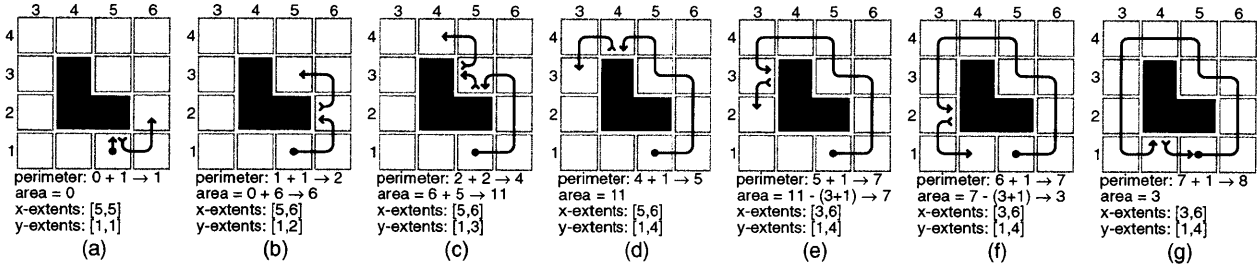


Figure 7-2: As the modules attempt to use the bug algorithm to route a sense (SEN) messages from its source at (5,1) to its non-existent destination at (5,2), they update the perimeter, area, and bounding box fields carried within the message. The perimeter is incremented as the result of every "collision" with the obstacle, and area is accumulated row-by-row. When the message returns to its sender, these parameters accurately describe the obstacle.

In the process of traversing the obstacle, the sense (SEN) message is modified by each module through which it passes so that by the time the message returns to its sender, it holds obstacle's area, perimeter, and the extents of the obstacle's bounding box. Figure 7-2 shows this process in action. The perimeter computed by the SEN message is incremented whenever the bug algorithm causes the message to virtually collide with the obstacle being duplicated. The area of the obstacle is integrated by rows. For each row, the minimum x-coordinate plus one is subtracted from the maximum x-coordinate, but these operations never occur simultaneously. Finally, the SEN mes-

sage logs the minimum and maximum x- and y-coordinates through which it travels to determine the original shape's bounding box.

While Figure 7-2 only shows a single module's SEN message, all modules on the border generate messages. To elect a leader module from those surrounding the obstacle, and to reduce the total number of messages transferred, modules discard incoming SEN messages from modules with lower unique IDs than their own. Because there is a single highest ID, all SEN messages except one will be discarded before they return to their sender. The module whose SEN message returns is the de facto leader. Figure 7-2 also omits the fact that all modules on the external perimeter of the entire configuration of modules generate SEN messages. These messages are routed in an identical manner around the exterior of the entire ensemble of modules, but when the message generated by the module with the highest ID returns to its sender, the sensed area will be negative, so the module will know that it did not detect an obstacle.

7.1.3 Border Notification Algorithm

The border notification phase duplicates the border of the original shape in the nearby modules and involves three types of messages. Duplication (DUP) messages inform each module on the border of the original shape of their special status. Border (BOR) messages are sent by modules on the perimeter of the original shape and inform each module that is on the border of what will become the duplicate shape of their status. Confirmation (CON) messages, in turn, are sent by recipients of border (BOR) messages and allow the leader to determine when the border of the duplicate is complete.

The border notification phase begins with the leader selected by the shape sensing phase attempting to use the bug algorithm to route a duplication (DUP) message to its missing neighbor whose position is instead occupied by the obstacle to be duplicated. Like the sense (SEN) message that is already sent, the duplication (DUP) message traces the perimeter of the obstacle conveying two critical pieces of information to each module on this border: the leader's coordinates and a duplication direction vector, (whose length is determined by the bounding box of the original shape).

As the DUP message passes through the modules on the perimeter of the original shape, each module attempts to route a border (BOR) message to the module identified by the direction vector added to the sender's coordinates. After stimulating each module on the perimeter of the original shape to send a border (BOR) message, the DUP message eventually returns to the leader where it is discarded.

When the BOR messages reach their destinations, these modules become the border of the duplicate shape. Because the BOR messages also carry the coordinates of the leader module, each BOR recipient sends a border confirmation (CON) message back to the leader carrying the length of perimeter of the duplicate shape on which the module borders. By comparing the cumulative length of all received confirmation (CON) messages to the known perimeter of the original shape, the leader determines when all modules on the border of the duplicate have been notified of their role.

7.1.4 Shape Fill Algorithm

The shape fill phase notifies all modules inside the border of the duplicate shape that they form the duplicate object and should remaining solidified when all other modules disassemble. The phase begins when the leader has received confirmation messages from every module on the border of the duplicate shape. With the border of the duplicate complete, the leader sends a fill (FIL) message that floods the entire network of modules. Each instance of the message contains an "included" bit, (initially cleared), that is toggled every time the message passes through a module on the border of duplicate shape. As a result, only modules surrounded by the duplicate border receive a fill (FIL) message with the included bit set. These modules know that they are included in the final structure and do not break their shared bonds during the disassembly phase. Each module inside the border of the duplicate shape sends another (area) CON message to the leader. By comparing the number of received area CON messages to the known area of the duplicate object, the leader can determine when all modules that compose the duplicate object have received a fill (FIL) message.

7.1.5 Self-Disassembly Algorithm

After the leader can verify that each module in the duplicate shape knows that it should not disassemble, the leader broadcasts a disassembly (DIS) message to the entire structure. This message floods the network and the unincluded modules begin disassembling from their neighbors in an orderly fashion (see Chapter 6), until only the duplicate object remains.

7.2 Storage and Communication

The distributed shape duplication algorithm requires only a constant amount of storage per module which is independent of the number of obstacles in the system or size of the object being duplicated. During localization, a module only stores its position. In the sensing phase, a module updates sense (SEN) messages as they pass through the module, but no information is stored. During border notification, the new border modules that surround what will become the duplicate shape must store a list of their faces that border on the duplicate obstacle, but this is constant in size and can never exceed the dimensionality of the system. During the fill process, a module only needs to record a constant amount of information: whether it is in the structure and whether it has already sent a fill (FIL) message to each neighbor (to minimize the number of FIL messages transmitted). Finally, during disassembly, modules do not need to store any information. Throughout the entire process, the leader module only stores a constant amount of information: the perimeter and area of the shape being duplicated. It never holds a complete description of the shape being duplicated. Additionally, it only tracks the cumulative confirmed perimeter and area conveyed by the confirmation (CON) messages instead of keeping a list of exactly which modules have transmitted CON messages. The total storage per module is therefore $O(1)$.

The number of messages exchanged also scales favorably. The worst case scenario occurs when the area of the original object approaches the area of the initial block of material and when the shape of that object approaches a 1-by-n rectangle. During localization, each module may exchange a constant number of messages with each of its neighbors resulting in $O(n)$ messages exchanged. In the sensing phase, there are at most $O(n)$ modules that each transmit sense (SEN)

messages. Each SEN message may travel $O(n)$ hops before being discarded by a module with a larger ID. Therefore, the total number of messages is $O(n^2)$. During duplication, the total number of messages exchanged is also $O(n^2)$ as the number of modules in the perimeter of the duplicate may approach n, and each border (BOR) message may have have to travel a distance of $O(n)$ to arrive at its destination. Normally, the fill process requires $O(n)$ messages, as each module just forwards fill (FIL) messages to its immediate neighbors. If there are many missing modules, the number of messages may approach $O(n^2)$. Finally, disassembly, because it is a flood fill process like localization, only requires $O(n)$ messages. So, the total number of messages scales as $O(n^2)$ implying that the per module number of messages exchanged scales as $O(n)$. While a constant scaling would be preferable, it is unrealistic to expect to duplicate an arbitrarily large shape in a distributed manner using only a fixed number of messages per module.

7.3 Robustness

The system is robust to both missing communication links and missing modules. In what follows, we assume that the physical state of the system is static: once the duplication process has begun, neither communication links nor modules are removed from or added to the system.

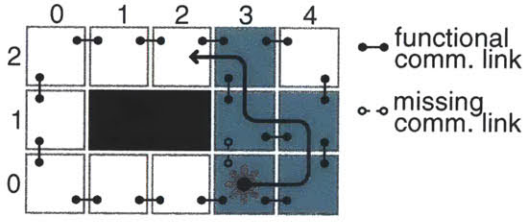


Figure 7-3: The missing link between modules (3,0), and (3,1) will cause the module located at (3,0) to send a sense (SEN) message to (3,1). Instead of discarding this message, and aborting the entire border sensing process, the module at (3,1), (even though it is the intended recipient), must continue routing the message around the perimeter of the obstacle so that it eventually returns to the leader.

First, consider the case of missing communication links. In general, missing links are not an issue so long as each module can communicate with at least one neighbor. When routing messages, the bug algorithm will treat missing links just like obstacles that must be avoided. The

one scenario in which a missing communication link can affect the system is shown in Figure 7-3. In general, missing communication links are problematic when they border on the object to be duplicated because the sense (SEN) messages sent by the two modules that share the missing link will actually reach their destinations, (unlike most SEN messages which are destined for a location occupied by the obstacle being duplicated). Referring to Figure 7-3, the SEN message transmitted by the module at (3,0), that also happens to have the highest ID, would be discarded by the module at (3,1) instead of circumnavigating the obstacle. Furthermore, the module at (3,0) will discard all other SEN message because they come from modules with lower IDs. To alleviate this problem, and make the system robust to missing communication links anywhere, we have modified the routing algorithm so that it never acknowledges when a SEN or duplication (DUP) message reaches its destination. Instead, it will allow the message to keep traveling.

The duplication algorithm can also robustly handle missing modules. There are exactly four distinct locations from which a module can be missing, and each is shown in Figure 7-4. First, when a module is missing from a location adjacent to the original object being duplicated, (such as at location $(5,4)$ in Figure 7-4), missing module appears to be a part of the original object, and the duplicate will reflect this, as shown by the module at $(12,4)$ being included in the duplicate.

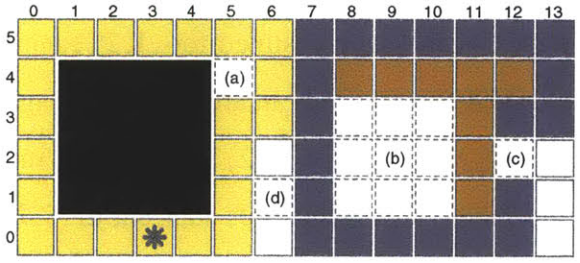


Figure 7-4: The duplication algorithm is robust enough to handle modules missing from any potential location: (a) adjacent to the object being duplicated; (b) in the interior of the duplicate shape; (c) on the border of the duplicate shape; or (d) in any other position.

Second, when a module is missing from another location that is also not the border or interior of the duplicate shape, such as (6,0) in Figure 7-4, we need to ensure that the algorithm does not duplicate this apparent obstacle. We guarantee that the algorithm only duplicates the intended obstacle by placing a threshold on the area of objects that will be duplicated. Sense (SEN) messages that return to their sender specifying an obstacle with an area smaller than this threshold are simply

discarded. This approach to ignoring small holes in the initial packing of modules is reasonable given, that to achieve acceptable resolution, most objects will be orders of magnitude larger than the modules themselves.

Third, the duplication algorithm can gracefully handle modules missing from the interior area that will become the duplicate shape, such as the 9 modules centered at $(9,2)$ in Figure 7-4. In general, the algorithm will make its best effort to duplicate the original, but a large chunk of the duplicate will be missing when the process completes. During the shape fill phase, the modules surrounding this gap in the structure will attempt to route Fill (FIL) message to the 9 missing modules. As the system discovers that each of these message is undeliverable, it will attempt to route disconfirm (DCON) messages to the missing modules' conjugate locations in the original obstacle. For example, if the module at $(7,2)$ in Figure 7-4 determines that a FIL message destined for $(8,2)$ is undeliverable, $(7,2)$ will attempt to route a disconfirm (DCON) message to $(1,2)$. Because location (1,2) is occupied by the passive obstacle being duplicated, this DCON message will never be delivered. As the system discovers that each of these DCON messages is undeliverable, it sends an area confirmation (CON) message to the leader so that the leader can account for the entire area of the duplicate in order to trigger the self-disassembly phase. Continuing our example, if module at $(5,2)$ determines that the DCON message destined for $(1,2)$ cannot be delivered, the module acts as a proxy for the module at $(8,2)$ and sends an area CON message to the leader at $(3,0)$. Additionally, $(5,2)$ sends FIL messages to each of $(8,2)$'s neighbor's, including in particular, $(9,2)$. This last step is critical because there are no modules adjacent to $(9,2)$ that could otherwise generate the necessary (though undeliverable) FIL message. Without this last step, the leader would never receive a CON message from a module proxying for the missing module at $(9,2)$. We use a combination of highest ID and distance to discard many FIL messages so that we do not generate an excess of area CON messages that would confuse the leader. In this particular example, there will be four undeliverable FIL messages sent by proxy modules to the module at $(9,2)$. The system will discard all except the one sent by the module with the highest UID to ensure that only one additional proxy CON message is generated.

Fourth, and finally, it is easy to handle modules missing from the border of the duplicate shape

such as the module missing from $(12,2)$ in Figure 7-4. During the border notification phase, the border (BOR) message sent from $(5,2)$ to $(9,2)$ will be determined to be undeliverable by some module. This module will in turn act as a proxy for the missing module and send a border CON message to the leader on behalf of the module at $(12,2)$. The leader can then account for all border modules before initiating the shape fill phase.

During the shape fill phase, the algorithm handles missing border modules as it does missing interior modules. An undeliverable FIL message destined for $(12,2)$ generates a DCON message that is sent to the missing module's conjugate at $(5,2)$. In contrast to the interior case, this DCON message is actually delivered because location $(5,2)$ is on the border, not inside, of the original shape. Because this message is delivered, we know that the module at $(12,2)$ is itself a border module. As a result, there is no need to send an area CON message to the sender.

7.4 Automated Duplication Placement

We created an algorithm which allows the system to automatically decide where within the initial block of programmable matter to place the duplicate shape. In an automated duplication system with millions of minuscule programmable matter modules, it would be difficult for the user to explicitly instruct the system where to place the duplicate. Our automated shape placement algorithm eliminates the need for the user to specify where the duplicate should be placed. The automated shape placement algorithm is executed by the obstacle leader between the sensing and border identification steps of the larger distributed duplication process. In short, the algorithm attempts to find the optimal placement of the duplicate object's bounding box within the rectangular bounding box surrounding all modules in the system.

The placement algorithm represents the duplicate object with a rectangular bounding box. As explained above, the shape sensing algorithm learns the size and position of original shape's bounding box by routing a sense (SEN) message around the perimeter of the shape being duplicated. When this sense message returns to the obstacle leader, it contains the bounding box. Figure 7-5 shows the bounding box of the original shape being duplicated (labeled "O") as a dashed line. A bounding box of the same dimensions will also enclose the duplicate shape that the system is

attempting to form. The system must decide where to place the duplicate's bounding box. Figure 7-5 shows four potential positions for the duplicate (labeled "A" through "D"). The bounding box associated with each potential position is drawn as a dotted line.

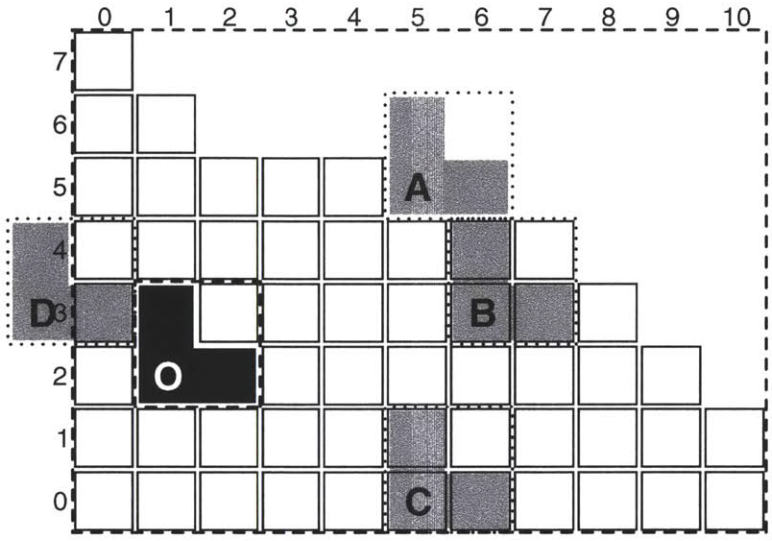


Figure 7-5: The automated duplicate placement algorithm attempts to place the duplicate shape as far from the edges of the surrounding block of material as possible. To simplify the optimization, the algorithm uses rectangular bounding boxes, (dashed and dotted lines), to represent both the original shape and the collection of programmable matter modules surrounding it. In this example, the algorithm determines that placement B is ideal.

Modules on the perimeter of the collection of programmable matter modules encasing the shape being duplicated also route sense messages around the perimeter of the collection of modules. To be consistent with terminology introduced in Chapter 8, we call this two-dimensional collection of communicating modules a *slice*. One of the sense messages traversing the inside perimeter of the slice will eventually return to the *slice leader*, which is the module on the perimeter of the collection of modules with the highest UID. Because it circumnavigated the inside of the slice, (in contrast to the outside of an obstacle), this sense will indicate that the sensed area is negative. The slice leader uses this fact to determine that it is, in fact, the *slice*, not *obstacle*, leader. Despite returning with a negative area, the sense message that returns to the slice leader holds the position and size of rectangular bounding box surrounding all modules in the system. This bounding box is also represented by a dashed rectangle surrounding all modules in Figure 7-5.

With both the obstacle's and the slice's bounding boxes known, the automated shape placement algorithm attempts to find the optimal placement of the obstacle's bounding box within the slice's bounding box. To do so, the algorithm instructs the slice leader to broadcast the slice's bounding box to all modules in the system. Because the slice leader does not know the coordinates of the obstacle leader, this is the easiest way to guarantee that the obstacle leader learns the slice's bounding box. Once the obstacle leader receives the message containing the slice's bounding box, it considers four distinct placements of the duplicate shape. In particular, the obstacle leader considers placing the duplicate shape in the four cardinal directions relative to the original shape's location. Starting to the north, and moving clockwise, the obstacle leader considers placements A through D in Figure 7-5.

Determining the optimal placement is a two step process. First, for each cardinal direction, the algorithm determines if the duplicate has any chance of fitting between the original shape's bounding box and the slice's bounding box. In the example of Figure 7-5, the algorithm determines that placements A, B, and C are all potential candidates. It eliminates placement D because the duplicate's bounding box is two modules wide, but the space between the left side of the original's bounding box and the slice's bounding box is only one module wide.

Second, having eliminated cardinal directions that it knows will not fit the duplicate, the obstacle leader attempts to find the optimal placement among the remaining directions. For each direction, the algorithm attempts to center the duplicate's bounding box in both the x- and y-directions. For example, when attempting to place the duplicate north of the original, the algorithm attempts to center the duplicate between the top edge of the original's bounding box and the top edge of the slice's bounding box. Simultaneously, the algorithm attempts to center the duplicate between the left and right edges of the slice's bounding box.

The algorithm scores each potential placement. The score is the sum of the extra space surrounding the duplicate object. Returning to our example in Figure 7-5, the score associated with placement A is 11. This score comes from the 1 module of space above, 4 modules to the right, 5 modules to the left, and 1 module below the duplicate. The reason placement A is only credited with 1 module of space below is that the algorithm measures the between the bottom of the du-

plicate and the top of the original shape. It assumes that the original shape may extend farther to the right that it actually does. The scores for all placements are shown in Table 7.1. Ultimately, placement B is declared optimal with a score of 12.

Table 7.1: The automated placement algorithm chooses the viable position with the highest score. For reference, consult Figure 7-5.

Cardinal Dir.	Placement	Viable	Score
North	A	Yes	11
East	B	Yes	12
South	C	Yes	9
West	D	No	n/a

While the automated shape placement algorithm we describe makes some attempt at optimality, it is not ideal. While it just happens to be the case that placement B in Figure 7-5 results in an accurate copy of the original being formed, all of the modules that will be used to build that duplicate could be removed without affecting the algorithm's decision to place the duplicate at position B. This shortcoming is due to the fact that the algorithm uses rectangular bounding boxes to represent more complex shapes. A simple bounding box cannot capture the fact that there are many modules missing from the upper right in Figure 7-5. Additionally, the algorithm fails to account for any modules missing from the interior of the structure. These non-idealities are the result of a calculated decision. If we used more complex data structures to describe the shape of the obstacle and slice, the communication, storage, and computation costs would rise. For an arbitrarily complex slice, containing an arbitrarily complex object to be duplicated, we would need $O(n)$ bits of storage to determine the optimal placement. We have chosen rectangular bounding boxes because they only require $O(1)$ space, and they make the optimal placement calculations easy and fast.

We have not extended the algorithm to scenarios in which we wish to magnify the original or create multiple copies, but this task should not be difficult. For example, instead of using an exact replica of the original's bounding box to place the duplicate, we could use a modified bounding box to describe the duplicate object(s). The algorithm could simply magnify or replicate the original shape's bounding box to create the duplicate's bounding box. It could then use this larger bounding box during the optimization process.

7.5 Multiple Duplicates and Magnification

We can extend the duplication algorithm to form multiple copies of the original shape or a magnified duplicate that is an integer factor, M, larger than the original. The process of forming multiple duplicates is accomplished by adding row and column count fields to each border (BOR) message sent by the modules on the perimeter of the original shape. These row and column counts specify the dimensions of the array of duplicates that will be formed next to the original object. When the BOR messages reach their destinations, they both inform the destination modules of their status as border modules and forward themselves along to notify the next set of border modules. So long as the remaining column count of a BOR message is greater than one, the receiving module decrements it and forwards the message in the x-direction. Likewise, if the remaining row count is greater than one, the receiving module decrements it and forwards the BOR message in the y-direction. For a concrete example, see Figure 7-6.

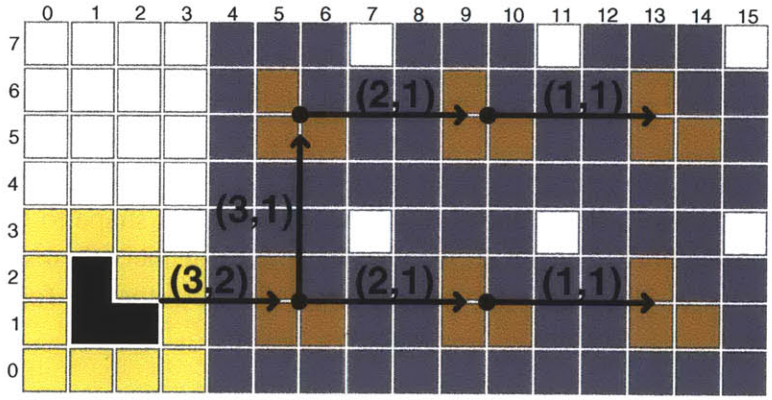


Figure 7-6: When creating multiple copies of an original shape, we arrange the copies in a array whose dimensions (here 3-by-2) are appended to each original border (BOR) message. As the BOR messages move through the structure, the remaining remaining row and column counts are decremented as shown.

The process of magnifying the duplicate shape is likewise simple and is illustrated by Figure 7-7. We append the magnification factor field, M, to each BOR message. In addition, the modules on the perimeter of the original shape modify the destination of the BOR message they each send so that the destination includes an additive factor that depends on the product of M with the module's relative location within the bounding box of the original shape. Each module that receives one of

these primary BOR messages become a local leader of an M-by-M group of modules. As shown in Figure 7-7, each local leader, (in red), may or may not actually border on what will become the duplicate shape. As a result, each local leader computes which of the modules within its M-by-M domain, (outlined by a black border), should actually border on the delicate shape. The local leader then sends each of these true border modules a secondary BOR message.

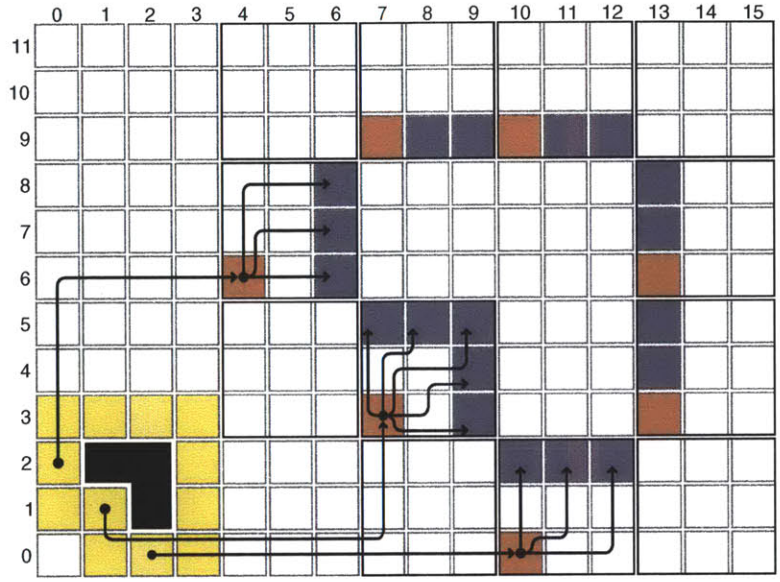


Figure 7-7: When creating a magnified version of the original shape, a set of primary border (BOR) messages are sent by the modules on the perimeter of the original shape to local leaders (in red) which may not lie on the border of the duplicate. These local leaders then send secondary BOR messages to the modules within their domain (outlined in black) that do form the border of the duplicate shape. Note: for clarity, not all messages are shown.

When forming multiple duplicates or a magnified duplicate, missing modules are dealt with as they are in the 1-to-1 duplication case. Also, the leader now waits for a number of border and area confirmation (CON) messages that is multiplied by either the number of duplicates or the magnification factor before beginning the shape fill and self-disassembly phases, respectively.

7.6 Experimental Results

We performed simulated and hardware-based experiments, the results of which are shown in Table 7.2. We had 20 Smart Pebble modules available to use in when duplicating small shapes. We

used the Sandbox simulator to perform large experiments that would otherwise require more hardware modules than we currently have available. The simulator can be told to randomly break a set percentage of inter-module communication links or randomly remove a set percentage of modules, as indicated by the Broken Links and Missing Modules columns in Table 7.2. Exactly which links and modules are removed changes with each trial. The Disassembly Begun column in Table 7.2 indicates in what percentage of trials the self-disassembly phase was started by the leader module, indicating that the leader module at least received all border and area CON messages. In cases where self-disassembly did start, the Correct Bonds column indicates what percentage of all inter-module bonds were in the correct state after the self-disassembly finished. It excludes trials in which the leader failed to initiate the self-disassembly.

Table 7.2: Experiments show the duplication algorithms working correctly in both simulation and hardware.

Shape	*Sim/ HW*	*Broken Links [%]*	*Missing Modules [%]*	*Mag. Factor*	*Array Size*	*No. Trials*	*Avg. Time [s]*	*Disassembly Begun [%]*	*Correct Bonds [%]*
Fig. 7-8(a)	HW	Unknown	0.0	1x	1x1	15	29.3	80.0	89.8
Fig. 7-8(b)						16	38.0	100.0	94.0
Fig. 7-8(c)						15	47.1	100.0	87.5
Fig. 7-8(d)						15	50.6	93.3	90.7
Fig. 7-4	Sim	5.0	0.0	1x	1x1	25	Unknown	100.0	100.0
Fig. 7-6	Sim	10.0	5.0	1x	3x2	25	Unknown	100.0	100.0
Fig. 7-7	Sim	10.0	5.0	3x	1x1	25	Unknown	100.0	100.0
Fig. 7-8(e)	Sim	10.0	5.0	1x	1x1	25	Unknown	100.0	100.0
		5.0	2.5	1x	2x2	25	Unknown	100.0	100.0
		5.0	2.5	2x	1x1	25	Unknown	100.0	100.0

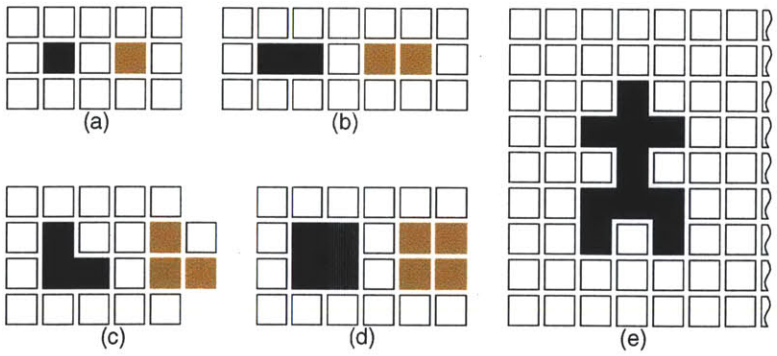


Figure 7-8: We verified the duplication algorithm using a variety of shapes in both hardware, subfigures (a-d), and simulation, subfigure (e).

In general, the distributed algorithm performed well. In hardware trails, there were four instances where the leader module did not initiate the self-disassembly phase. It is difficult to isolate the cause of these failures, but we suspect that some communication links may have lost connectivity after the duplication process began. Additionally, in hardware, the self-disassembly phase only resulted in 90.0% of bonds being successfully broken. While this is a problem that deserves further investigation, it is does not reflect on the core of the duplication algorithm's reliability. In simulation, despite challenging environments with 10% of communication links removed and 5.0% of modules removed, the algorithm performed flawlessly when creating a 1-to-1 duplicates, magnified duplicates, or multiple duplicates.

While the algorithm must handle a small number of broken links when using the Smart Pebble hardware, we wanted to ensure that the algorithm could handle an even higher percentage of broken links and missing modules. We used the simulator to replicate the humanoid shown in Figure 7-8(e). Table 7.2 shows that the algorithm performed flawlessly for missing link rates as high as 20%. Higher broken link rates typically result in a configuration that does not include a closed communication path around the original shape.

We also used the simulator to measure the running time and communication cost of the duplication process. Our experiments were focused on duplicating the simple 7-module wrench shown in Figure 7-9. We configured the simulator to count the number of messages exchanged by each module during each phase of the duplication process. We also used the simulator to record the time required to complete each phase. To measure how the number of messages scaled with the size of the wrench, we ran 8 different experiments as we scaled the size of the wrench by all integer factors between 1 and 8, inclusive. In each experiment, we kept a 2-module border between the original wrench and the duplicate and between the exterior border and the two wrenches. As an example, Figure 7-10 illustrates the 5x-scaled original and its successfully created duplicate.

We ran a total of 117 trials, and at least 9 trials for each of the 8 magnification factors. The running times of the duplication process, as well as those of each major sub-algorithm, are shown in Figure 7-11. The figure plots the running time against the number of active modules in the initial block of material surrounding the wrench. The eight tick marks along the x-axis correspond to the

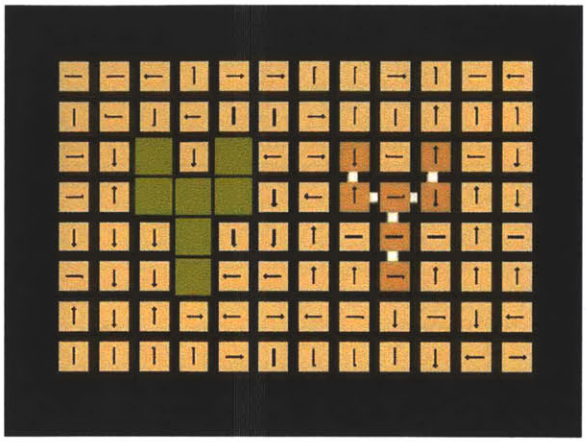


Figure 7-9: To characterize how the duplication algorithm scales, we duplicated scaled versions of the wrench on the left using the Sandbox simulator. The completed duplicate is shown in the right.

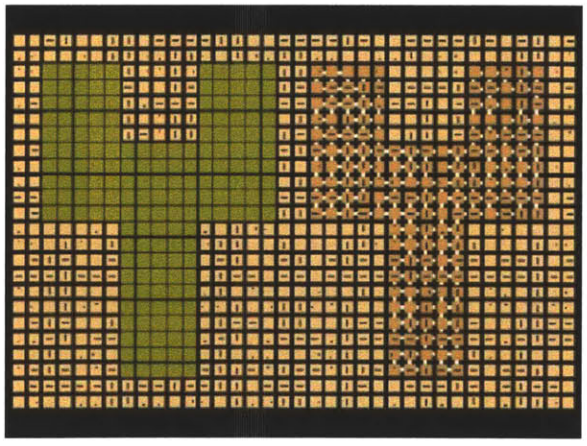


Figure 7-10: We duplicated up to 8x-scaled versions of the basic 7-module wrench. Here we illustrate a 5x-scaled original and its duplicate. In all cases, we left a 2-module border between the wrenches and the external perimeter of the initial block of Smart Pebbles.

eight different magnification factors. The running time is weakly quadratic in the number of active modules. This makes sense given that the total number of messages exchanged is $O(n^2)$ and each message will require $O(1)$ time to exchange. While many messages are exchanged in parallel, it is not enough to produce an overall linear running time.

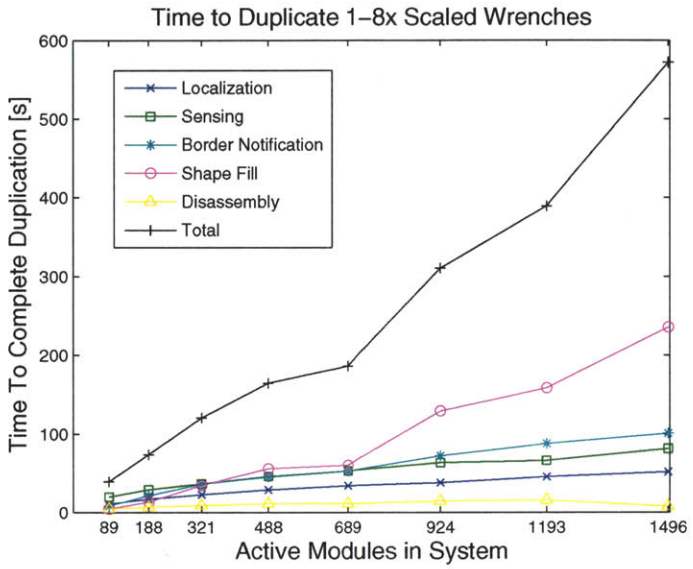


Figure 7-11: The time to complete the duplication process scales in a roughly quadratic fashion. This makes intuitive sense given that the total number of messages exchanged scales as $O(n^2)$. While some messages can be exchanged in parallel, others must be exchanged sequentially giving rise to the quadratic scaling of time. Each of the eight tick marks on the x-axis corresponds to one of the eight scaling factors.

We also logged the number of messages exchanged during these 117 trials. The results, plotted in Figure 7-12, show that the number of messages exchanged in a practical example scales better than the expected $O(n^2)$ result. When duplicating the 8x-scaled wrench, the 1496 active modules exchanged a total of 90,000 messages. On average, that is only 60 messages per module. Figure 7-13 plots the imperfect, but roughly linear relationship between the average number of messages per module and the total number of active modules in the system. The same figure also illustrates that the maximum number of messages exchanged by any module in the system scales in a linear fashion. It is worth noting that the average number of messages exchange by any given module is roughly an order of magnitude lower than the maximum number of messages exchanged by a module. Figure 7-14 better illustrates the distribution of the number of messages exchanged by

each module in the system. In particular, the figure shows that the vast majority of modules in a system exchange a number of messages quite close to the average number of messages exchanged, i.e. there is little variation in the number of messages exchanged. In summary, these results confirm the duplication algorithm's theoretical communication cost of $O(n)$ messages exchanged per module and the worst case $O(n^2)$ total messages exchanged.

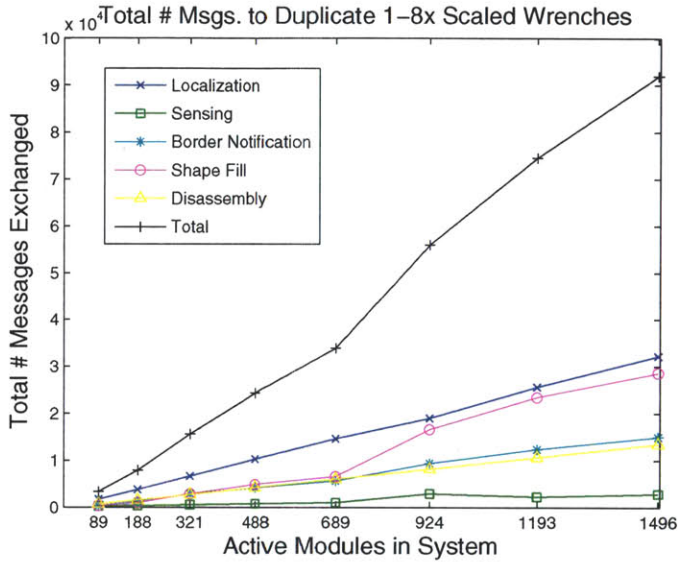


Figure 7-12: The number of inter-module messages exchanged (i.e. communication cost) scales as $O(n^2)$ in the worst case. Here we attempt to illustrate the relationship by plotting the total number of inter-module messages against the number of active modules required to duplicate scaled version of the wrench shown in Figure 7-9. In practice, the relationship appears more linear than quadratic. Each of the eight tick marks on the x-axis corresponding to one of the eight magnification factors. Note, we do not include the localization (LOC) messages repeatedly sent by each module while waiting for a position (POS) from its neighbor.

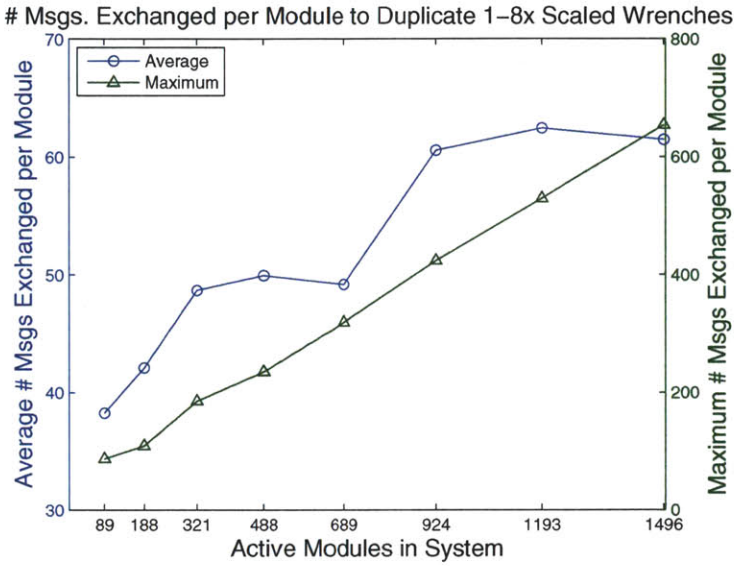


Figure 7-13: The average number of messages exchanged per module scales roughly linearly with the number of active modules in the system. The maximum number of messages exchanged by any module in the system scales in a highly linear fashion. Note the different scales used for the average and maximum number of messages. The average is almost an order of magnitude lower than the maximum. Each of the eight tick marks on the x-axis corresponding to one of the eight magnification factors.

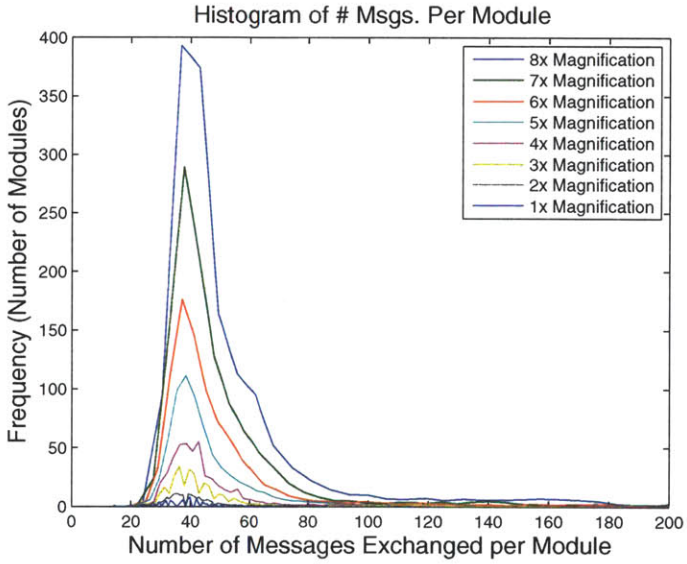


Figure 7-14: This plot illustrates the distribution of the number of messages exchanged by all modules in the systems duplicating the 8 scaled versions of the basic 7-module wrench. It shows that the variance from module to module is low. Most modules exchange a number of messages close to the average number. There are only a few modules that exchange a significantly higher number of messages.

Chapter 8

Three-Dimensional Duplication

In this chapter, we extend the two-dimensional duplication algorithm to duplication of three-dimensional objects that may be convex or concave (e.g. the cup in Figure 8-1). Much like the 2D duplication algorithm, the solution relies on local sensing of the boundary of the desired object and coordinated inference and planning to create a solid replica. The major addition to the 2D algorithm is the slicing of the 3D collection of modules into planes that operate semi-independently. As before, no module ever stores the complete goal shape nor the global state of the system; the memory required by each module is $O(1)$. Furthermore, the number of inter-module messages exchanged is $O(n)$ per module, where n is the number of modules in the system. We have implemented and evaluated this algorithm in simulation for environments containing thousands of modules.

The two-dimensional duplication process functions as follows (see Chapter 7 for details). For each location occupied by the obstacle, the algorithm identifies a *conjugate* module some distance away in a specified offset direction that will remain solidified when all other modules self-disassemble. Because the object being duplicated is inert and not composed of active modules, sensing and modeling its geometric shape is challenging. Figure 8-2 outlines our solution, which senses and identifies the border of the void in the module lattice that is occupied by the object. The algorithm identifies the surface of the object by message passing and marks all the lattice modules on the object's perimeter. A shifted replica of this perimeter is created at a different location in the

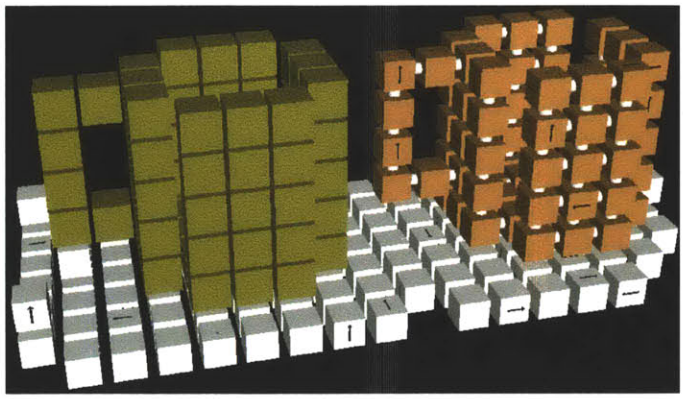


Figure 8-1: The distributed duplication algorithm is capable of duplicating arbitrary 3D objects like the coffee mug (left) using a collection of programmable matter modules. The modules envelop and sense the shape of the original object before forming a duplicate (right) from spare modules. Any extra modules (white) are then brushed aside to reveal the completed object.

lattice some automatically determined offset distance away from the original. Then, the algorithm uses a flood fill process to notify all the modules within this surface that they are a part of the duplicate object. The result is the desired one-to-one correspondence between voids in the lattice and conjugate duplicate modules. This approach works for arbitrarily complex surfaces, both convex and concave.

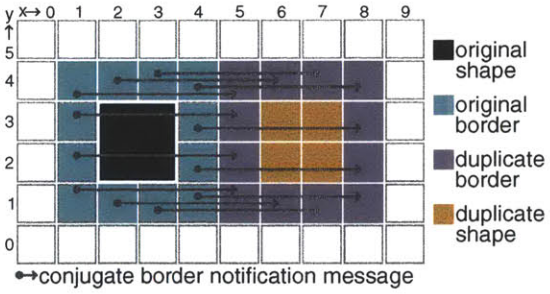


Figure 8-2: The distributed duplication algorithm works by sensing the border of the shape to be duplicated. Once the border is identified, each module on the border notifies a conjugate duplicate border module that is offset a fixed distance in a given direction. With all the modules on the duplicate border aware of their status, the algorithm notifies all modules inside the duplicate border that they are part of the duplicate shape.

8.1 Challenges of Three-Dimensional Duplication

Despite its relatively simple high-level description, there are many challenges when implementing the duplication algorithm in three dimensions. The modules must (1) all agree on the offset distance between the original and the duplicate; (2) posses a way to differentiate between bordering on the obstacle to be duplicated and the very exterior of the initial block of material; (3) synchronize when each module's contribution to the duplicate border is complete so as to not start disassembling prematurely; (4) be able to do all of the above while using a constant amount of memory and a number of messages that scales favorably.

A naive solution that considers the border as a set of individual modules instead of a closed surface will fail for a number of reasons when duplicating complex objects. First, if all border modules do not agree on the offset distance by which to shift their conjugate, the duplicate object may appear skewed, even worse, completely incoherent. Second, if the system does not duplicate the correct set of border modules, concavities in the original will be filled in the duplicate. Third, if some group of conjugate border modules decides to start filling their interior before the entire surface of the duplicate has been constructed, the fill message may spill out of the duplicate. Every module in the system may then decide that it is part of the duplicate. Finally, if some module initiates the disassembly process before all modules in the duplicate have received a fill message, some modules will disassemble instead of remaining part of the duplicate shape.

The first challenge that the algorithm must overcome is the fact that there is no efficient way to sense and identify the border of the passive shape directly in three dimensions. In particular, we need a message passing algorithm that can "wrap" around, and thereby sense, the entire 3D shape. To accomplish this, we decompose the duplication process into 2D subproblems using a layer-by-layer approach. The initial block of material is cut into individual planes, and duplication proceeds semi-independently in each plane. In each plane, the border identification problem uses the bug algorithm [66]. Any module on the border (as determined by a missing neighbor), attempts to route a message to the unoccupied lattice location. The message acts as the bug, and the void as the obstacle to be avoided. In its futile attempt to reach its destination, the message will circumnavigate the entire obstacle before returning to its sender. In its circumnavigation, the message learns about

the shape of the obstacle. The challenge for 3D duplication is to synchronize all these planar processes. Concavities in the object to be duplicated need careful processing. Consider duplicating a coffee mug. If the mug is sitting upright, and if the planarization is confined to horizontal plane, some planes will contain two disjoint sets of lattice modules. There will be one group of modules surrounding the outside of the mug and a second, isolated set inside the mug. We call each of these groups a *slice*. Our algorithm can handle an arbitrary number of disjoint slices.

8.1.1 Three-Dimensional Routing Algorithm

The system also faces the problem of routing messages from modules on the inside surface of the mug to modules outside the mug. These messages must travel in three dimensions, so the bug algorithm that we use for intra-slice routing will not guarantee their delivery. Our approach leverages developments [65] in geographic routing to enable 3D routing while keeping the amount of routing information stored in the messages and nodes constant and quite small.

The three-dimensional routing algorithm developed in [65] operates on a tree composed of convex hulls. Each Smart Pebble module stores its own convex hull which holds the coordinates of other modules in the system that can be reached by descending the tree to one of the module's children. When there is no direct path available between a three-dimensional routing message's source and destination, the routing messages begins to traverse the tree of convex hulls. Because all modules are included in the tree, the message will eventually reach its destination. The convex hull information stored at each node helps to speed up the process by eliminating the need to traverse branches of the tree which the system knows do not contain the message's destination.

To further simplify the algorithm, we use rectangular boxes instead of arbitrary convex hulls. Each of our rectangular convex hulls can therefore be represented by a set of six points: the minimum and maximum x-, y- and z-coordinates of the box. Our approach ensures that it is simple to check whether a point lies inside of a convex hull, and it also ensures that it is simple to find the union of two convex hulls. In simplifying the representation of the convex hull, we do not affect the correctness of the routing algorithm. Compared to the optimal convex hulls that tightly bound the set of points they contain, our rectangular hulls include some number of additional points. Con-

sequently, a three-dimensional routing message may have to travel a longer path as it traverses the rectangular convex hulls. The modules that the messages visits will be a superset of the modules that it would have visited using the optimal convex hulls.

8.2 Three-Dimensional Duplication Algorithm

As illustrated in Figure 8-3, the key to the complete 3D duplication algorithm is to virtually cut the initial block of programmable matter into individual planes and then coordinate the two-dimensional duplication processes occurring in each plane. As shown in the $z = 2$ plane, a single cut plane may contain multiple distinct groups of nodes. We call each of these groups a *slice*.

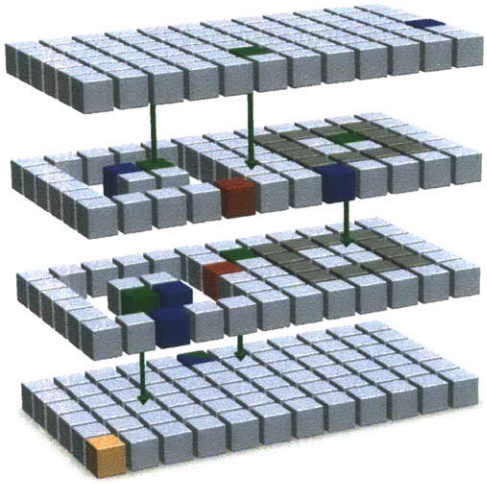


Figure 8-3: Here a 12x6x4 block of material encasing a shallow 4x4 bowl (transparent) is sliced along the x-y plane. We reference all coordinates to the node in the lower-front-left corner which we assume is located at position $(0,0,0)$. Each distinct group of modules within a plane (there are two in the $z = 2$ plane) is termed as *slice* and has a slice leader (blue) that is always on the slice's exterior border. Additionally, each slice has an inter-slice parent link module (green) that can be located arbitrarily. The arrows point from inter-slice parent link modules to their parent slices. Finally, each obstacle has an associated obstacle leader (red).

At a high level, each slice executes the basic 2D duplication process semi-independently, but the slices must synchronize and exchange information for the duplication to succeed. As a result, there are unique steps in the 3D algorithm that have no counterpart in the 2D case. The duplication process is initiated by sending one module on the exterior of the raw block of material a start

message specifying (1) the slice plane; (2) the coordinate direction in which the duplicate should be formed; and (3) which of the module's faces is an exterior face of the initial block. This module then assumes its position is $(0,0,0)$ and that it has a standard orientation. Once begun, the 3D duplication algorithm has 10 steps:

1) Encapsulation and Localization–As in the 2D case, the modules in the system solidify around the shape to be duplicated and exchange messages to learn their positions and orientations relative to $(0,0,0)$ within the lattice.

2) Hull Tree Construction–The three-dimensional routing algorithm uses a tree of convex hulls to route messages when there is no direct path available between the message's source and destination [65]. The tree is structured identically to the power supply dependency tree explained in Chapter 6. The structure of the tree is formed during the localization process. The first neighboring module to send a position (POS) message to an unlocalized module becomes the unlocalized module's parent in the tree. Hence, the structure of the tree is built from the root down to the leaves. The rectangular convex hulls associated with each node are constructed in the opposite order: from leaves up to root. When a module determines that it is a leaf of the tree (because it has no children), it sends its coordinates, (in the form of a 1-by-1-by-1 rectangular convex hull), to its parent. Each parent waits for all of its children to send their convex hulls. Once the parent has received them, it finds the union of its children's convex hulls and add its own coordinates. The parent module then forwards this new convex hull, (which contains its coordinates and the coordinates of all of its children), up the tree to its own parent module. Eventually, the convex hulls will propagate up to the root of the tree so that the root module's convex hull contains the coordinates of all modules in the system.

3) Shape Sensing–Within each slice, shape sensing operates nearly identically in three dimensions as it does in two. The only difference is that, in addition to electing an obstacle leader, the algorithm also elects a leader for the entire slice. This is illustrated in Figure 8-4. Just as each obstacle leader is the module on the border of the obstacle with the largest UID, the slice leader is the module on the border of the slice with the largest UID. Slice leader modules are differentiated from obstacle leaders because the sense (SEN) message that circumnavigates the exterior border

of the slice will return to the slice leader indicating a negative area. The magnitude of this number is the actual area of the slice, including the space consumed by any obstacles.

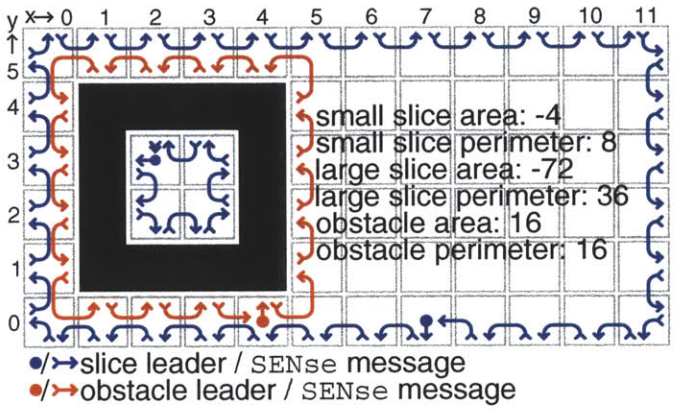


Figure 8-4: Examining the $z = 2$ plane from Figure 8-3, we see that each of the two slices detects its exterior border by allowing a sense (SEN) message (in blue) from the border module with the highest UID (the slice leader) to trace the slice's exterior by virtually colliding with all of the missing modules. The larger outer slice contains an obstacle, so the obstacle leader also transmits a SEN message (in red) that makes a complete circuit around the obstacle. For each $-x$ ($+x$) collision, the messages increment (decrement), their area count field by their current x-coordinate (x-coordinate $+1$). The messages increment their perimeter field after any collision.

4) Roll Call–The new slice leader broadcasts its position to all modules in the slice. Each module then replies with a roll call (RLC) message indicating whether it has zero, one, or two out-of-slice-plane neighbors. Obstacle leaders supplement their returned roll call messages with the size of the obstacle they represent. By counting the returning roll call messages, the slice leader can positively account for the entire area of the slice. Additionally, the slice leader learns how many out-of-slice-plane neighbors the slice has.

5) Slice Tree Construction–The algorithm constructs a tree in which each node is a slice. Eventually, this tree will be used to synchronize all slices. Each slices knows that it has accounted for all possible child slices when each of its out-of-slice-plane neighbors reports that it has a dedicated parent. This is detailed below.

6) Offset Distance Consensus–The slices need to agree on where the duplicate shape should be placed. To do so, each slice transmits the bounding box surrounding all obstacles in the slice to its parent slice. Then union of all these bounding boxes propagates up the slice tree to the root slice.

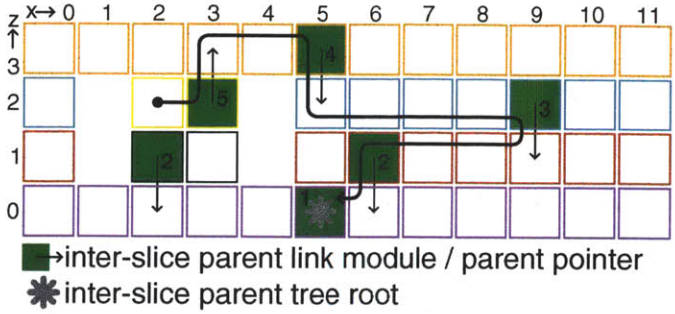


Figure 8-5: To aid inter-slice communication, the duplication algorithm forms a tree on the slices in the initial structure. Here, we see the modules of Figure 8-3 projected onto the x-z plane and each of the six slices outlined in a different color. The solid green modules are inter-slice parent link modules, and they serve as connection points from a slice to its parent. Every module in a slice learns the location of its inter-slice parent link module, so a message can be forwarded from any location, (in this case from the slice leader of the small slice in the $z = 2$ plane), to the root inter-slice parent link module.

The slice tree root module can then determine the global offset necessary to prevent a duplicate object in which the slices are skewed relative to one another. The root slice broadcasts this offset so that it can be incorporated into the border (BOR) messages sent to the conjugate border modules.

7) Exterior Face Determination–The 3D duplication algorithm must duplicate both the convex and concave portion of the original shape. For example, in Figure 8-3, the interior border of the cylinder must be duplicated along with the exterior border or else the duplicate object will become a cube instead of another cylinder. This means that in addition to duplicating the border of any obstacle contained within a slice, slices must also duplicate their exterior borders. The algorithm makes one exception to this rule: it does not duplicate any slice's exterior border if that border is also an exterior border of the entire block of material. We explain the differentiation process below.

8) Border Notification–As in the 2D case, all border modules, (except modules on the exterior of the entire block of material), send border (BOR) message to their conjugate modules that will become the border of the duplicate shape. The conjugate border modules reply with confirmation (CON) messages that are counted by the either the obstacle leader or slice leader (depending on the type of border being duplicated–exterior or interior). When the obstacle and slice leaders have received CON messages from each duplicate border module for which they are responsible, they

send a secondary CON messages to their respective inter-slice parent link modules. Once the inter-slice parent link module has received secondary CON messages for each obstacle, the slice as a whole, and any child slices, it forwards the CON message to its parent slice. Eventually, secondary CON messages will propagate to the root of the slice tree, and the root module will know that the border notification process is complete.

9) Shape Fill–The shape fill procedure in 3D is similar to the process in 2D. The fill (FIL) messages still carry the *inside* bit that is toggled every time the message crosses the duplicate border. The messages also need a *live* flag that is cleared when a message crosses a slice border. Until the *live* flag is again set–when the message crosses a border module–the *inside* flag is ignored. The reason for the *live* flag is that an included module in one slice has no way to determine whether a module in neighboring slice is also included. Ignoring some caveats addressed below in Section 8.6, the shape fill phase terminates just like the border notification phase. Each included module sends a CON message to the appropriate obstacle leader. Then the obstacle leader sends a secondary CON message to its inter-slice parent link module. The inter-slice parent link module waits for this and secondary CON messages from all child slices before propagating the secondary CON up the slice tree.

10) Self-Disassembly–Once the slice tree root receives CON messages from all child slices, it floods the network with a disassembly (DIS) messages causing all module except those forming the duplicate shape to disassemble. □

8.3 Message Routing Algorithm

The algorithm uses a combination of 2D (bug) and 3D routing techniques. For the shape sensing and exterior notification, the algorithm uses the bug routing algorithm which restricts messages to a specified plane. This limitation is actually an advantage because the messages tightly contour around all obstacles learning about their shape in the process. When sending border (BOR) and confirmation (CON) messages, the system must use a 3D routing algorithm. To see why, consider Figure 8-3: some of the CON messages returning to either of the leaders of the slice planes inside the cylinder have no 2D route available. In particular, a message from $(8,2,2)$ cannot use a 2D

routing algorithm to reach $(2,3,2)$.

We chose a 3D routing algorithm [65] based on convex hull trees because it was designed for 3D environments, requires only a small amount of fixed storage in each node, and is easy to implement. In short, messages act greedily and move in a direction toward their destination whenever possible. When blocked, messages switch to traversing the convex hull tree, only descending into nodes whose convex hulls contain the destination. We further simplify the algorithm described in [65] by, among other things, distilling each convex hull into a simple rectangular bounding box. As a result, each node only needs to store two coordinates per child (with a maximum of six children) to maintain the hull tree. Despite this simplification the algorithm works well for our purposes.

8.4 Synchronization Algorithm

To enable synchronization before the border notification, shape fill, and disassembly steps, the 3D duplication algorithm needs a way to ensure that all slices have completed the active step. To do so, the system must determine the total number of slices, so it builds a tree of slices, as shown by Figure 8-5. Note that this slice tree is separate from the convex hull tree used for 3D routing.

The slice tree is built from the root downward. The module originally given the start signal by the user informs its slice's leader that the leader should also be the root of the slice tree. (This is why, in Figure 8-3 the module at $(5,5,0)$ is both blue and green.) Once the slice leader is told that it is also the root of the slice tree, it broadcasts its location to all other modules in the slice. As a result, all modules in the slice learn the location of, what we term, their *inter-slice parent link.* Once a module knows the location of its inter-slice parent link, it is allowed to service incoming requests from neighboring slices looking for a parent in the slice tree.

In the neighboring slices that are not yet incorporated in to the tree, all modules send parent request messages to their out-of-slice-plane neighbors. Eventually, some out-of-slice-plane module, (which already knows the position of its inter-slice parent link module), will respond. The module in the unincorporated slice to which it responds becomes that slice's inter-slice parent link. That is, the module is the location of the link to the parent slice. This process repeats until all slices have selected an inter-slice parent link module.

When a slice is incorporated into the tree, the modules in the slice inform all of their out-of-slice-plane neighbors that they are now a part of the tree. Because each slice knows, (thanks to the roll call step), how many out-of-slice-plane neighbors it has, each slice can determine when all of its out-of-slice-plane neighbors have been incorporated into the tree. Therefore, we can guarantee that all slices are incorporated into the tree.

8.5 Exterior Face Identification Algorithm

When duplicating even a simple 3D shape like a coffee mug, the algorithm must account for both the concave and convex parts of the object's border. In the case of hollow cylinder, the concave, or interior part of the cylinder's face will correspond to the exterior border of multiple slices. The algorithm must duplicate the exterior border of these slices but not duplicate the exterior borders of slices that also serve as the exterior border of the entire block of material. Figure 8-4 provides an example: the algorithm should duplicate the exterior border of the inner 2-by-2 slice, but it should not duplicate the exterior border of the 12-by-6 slice that surrounds the smaller slice. We term the larger slice an *exterior slice.*

Our approach to differentiating exterior borders relies on the bug routing algorithm. By default, all exterior border modules assume that they should be duplicated. The module at $(0,0,0)$ initiates this process because it was told, (as part of the start command), which of its faces was an exterior face of the entire structure. This module then uses the bug algorithm in an attempt to route two exterior (EXT) messages in the direction of the specified exterior face. The first EXT message is routed in the slice plane, and the second is routed orthogonal to it. Because the bug algorithm is fundamentally a 2D algorithm, these messages will remain in their given planes. Like sense (SEN) messages, these EXT messages will circumnavigate the exterior border. As they pass through the modules on their routes, they notify those module that they too are on the exterior of the whole block (and should not duplicate their border). Additionally, they prompt those modules to emit their own EXT messages. Specifically, a message arriving in the slice plane will prompt an out-of-slice-plane message, and vice versa.

When an in-slice-plane EXT message completes its circuit around an exterior slice, it sends

a CON message to the root inter-slice parent module. This root can determine when the entire process is complete because it knows, as a result of the slice tree construction step, how many exterior slices comprise the entire structure.

8.6 Area Accounting During Shape Fill

One particular challenge of 3D duplication is that two slices occupying the same plane do not know of each other's existence. Consider again Figure 8-4. During the shape fill step, the obstacle leader in the outer slice, expects to receive sixteen CON messages. It does not know that the inner-most four of those lattice positions are not part of the duplicate obstacle. Our solution, illustrated in Figure 8-6, is to send a fake CON message from a conjugate border module corresponding to the leader of the interior slice.

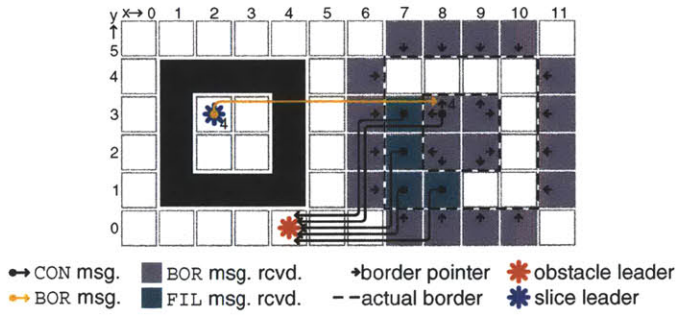


Figure 8-6: The obstacle leader of the exterior slice, expects to receive sixteen confirmation (CON) messages after the shape fill process beings. Although the obstacle leader does not know it, the obstacle is hollow. To ensure that the obstacle leader still receives all CON messages, the slice leader of the inner slice sends a special border (BOR) message to its conjugate instructing it to confirm four additional units of area despite the fact that the conjugate module is not included in the duplicate structure.

First, whenever an exterior border of a slice is duplicated, the slice leader, when sending a BOR message to its conjugate, includes its slice's area. This is shown in Figure 8-6. Second, during the shape fill process, the slice leader's conjugate border module sends a CON message to the outer slice's obstacle leader even though it is not part of the duplicate shape. This CON message is special because it accounts for four units of area, not just one like other messages. As a result of this two-step process, the outer slice's obstacle leader accounts for all sixteen units of area.

8.7 Storage and Message Scaling

The algorithm requires only a constant amount of storage per module. No part of the algorithm requires a module to amass any data that correlates with the number of modules in the system. The key to this attribute is the one-to-one correspondence between modules on the border of the original shape and modules on the border of the duplicate. Furthermore, when collecting CON messages, modules do not track the origins of the messages, only the total number received. The convex hull tree and the slice tree also require a fixed amount of space. A module only needs to store the rectangular hulls of at most six neighbors. While a slice in the slice tree may have an unlimited number of children, it does not track them. Messages only flow up the tree, so a slice only needs to know the location of its inter-slice parent. Finally, the routing algorithms forgo routing tables and other storage intensive strategies in favor of storing a constant amount of routing information in the messages themselves.

The total number of messages exchanged between modules in the system scales as $O(n^2)$ and is dominated by the exterior notification, shape sensing and border notification steps. The worst case message scaling occurs when the initial block of material approaches a 1-by-n line of modules. In this case, there will be $O(n)$ modules sending EXT messages and each message will have to circumnavigate $O(n)$ other modules before returning to its sender. The same scaling applies to the shape sensing phase if the object being duplicated also approaches a long rod: $O(n)$ modules will each transmit a SEN message and each message may travel $O(n)$ hops before being discarded. During border notification, there will again be $O(n)$ modules sending messages that each have to travel $O(n)$ hops before reaching their conjugates.

8.8 Experimental Results

We performed 818 experiments duplicating rods, cubes, square tubes, and the mug shown in Figure 8-1. The results are listed in Table 8.1. The overall success rate was 98.5%. The twelve failures were traced to routing deadlocks arising from congestion at choke points in the inter-module communication network. These were not failures of the high-level duplication algorithm. Congestion

arises because the modules have only a single transmit, receive, and payload buffer associated with each face. Each of these buffers can only hold a single message. Furthermore, a single incoming message can often prompt a module to send multiple outgoing messages. Two neighboring modules can create deadlock when each is parsing a message from the other that would send a message back to its neighbor. If each module's transmit buffer is already full, the parsing function has nowhere to queue the outgoing message, so it cannot purge the incoming message, (that is currently being parsed), from the receive buffer. With their receive buffers full, the modules cannot transmit messages to each other, so the transmit buffers remain occupied. While this situation is not common, the simulations show it is possible, and future work should aim find a solution.

Table 8.1: Experiments show the three-dimensional duplication algorithm working correctly in a variety of test cases.

Original Shape	*Encasing Shape*	*# Trials*	*# Successes*	*Avg. Msgs./ Trial*
1x1x1 Rod	5x3x3 Block	75	75	3586
2x1x1 Rod	7x3x3 Block	60	60	5520
3x1x1 Rod	9x3x3 Block	52	52	7573
4x1x1 Rod	11x3x3 Block	54	54	9926
5x1x1 Rod	13x3x3 Block	59	59	12203
6x1x1 Rod	15x3x3 Block	57	57	15149
7x1x1 Rod	17x3x3 Block	51	51	18079
2x2x2 Cube	7x4x4 Block	55	55	10321
3x3x3 Cube	9x5x5 Block	61	61	22034
4x4x4 Cube	11x6x6 Block	54	54	41155
5x5x5 Cube	13x7x7 Block	52	51	70521
6x6x6 Cube	15x8x8 Block	60	60	111850
7x7x7 Cube	17x9x9 Block	6	6	173060
4x4x4 Tube	11x6x6 Block	44	44	43157
5x5x5 Tube	13x7x7 Block	22	21	78583
6x6x6 Tube	15x8x8 Block	21	18	125980
7x7x7 Tube	17x9x9 Block	20	20	193950
Figure 8-1 Mug	17x7x7 Block	20	20	113690

Figure 8-7 shows a breakdown of the total number of inter-module messages passed as a function of the number of active modules in the system. The seven points along the x-axis correspond to cubes with side lengths 1—7. We observed identical scaling behavior when duplicating rods and

tubes. The quadratic that fits the total message count data is $0.058n^2 + 108.6n$. Even though the n^2 term will dominate past a few thousand modules, the average number of messages per module still scales approximately as $O(n)$.

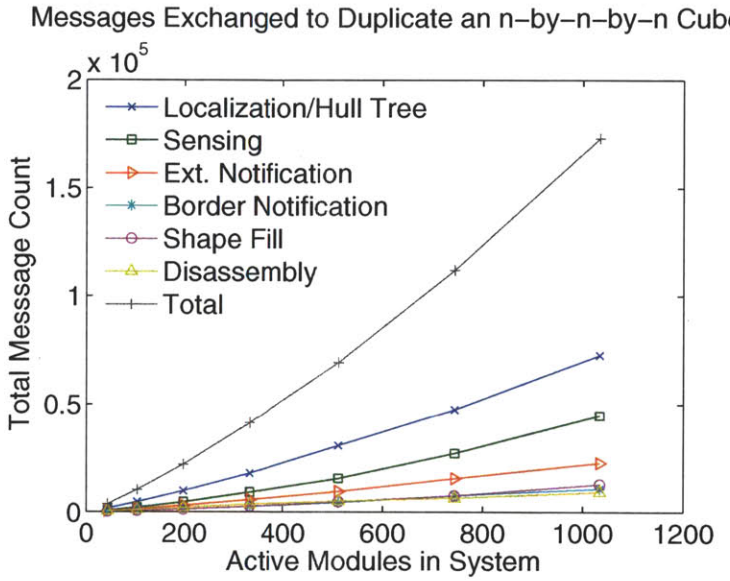


Figure 8-7: When duplicating cubes with edges lengths from one to seven, the total number of messages exchanged by all modules scales quadratically.

To further reinforce that the algorithms scale favorably, Figure 8-8, shows a histogram of the number of messages sent by each module when duplicating a 6x6x6 square tube with a wall thickness of one module. The results show that the vast majority of modules send between 100—200 messages.

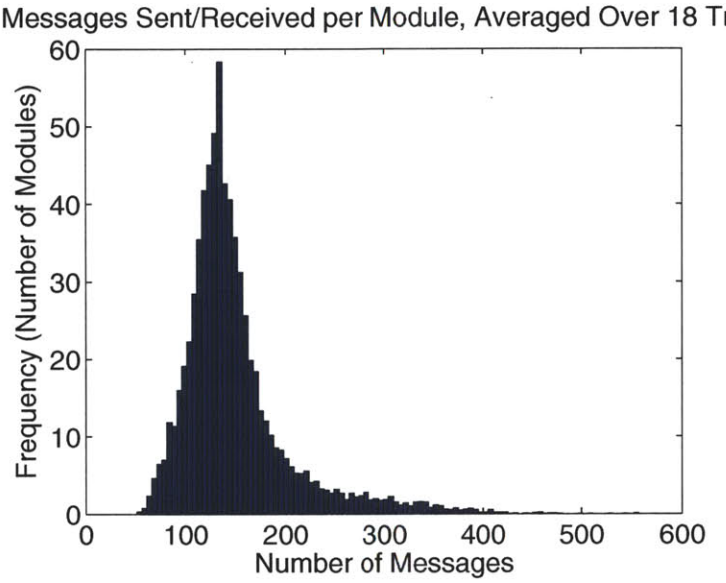


Figure 8-8: When duplicating an object, the variance from modules to module in the number of messages sent is small. Here, the average is number of messages per module when duplicating a 6x6x6 square tube is 150.1 and the standard deviation is 55.3.

Chapter 9

Conclusion

Over fifty years ago, L. S. Penrose dreamt of the following:

> Suppose we have a sack or some other container full of units jostling one another as the sack is shaken and distorted in all manner of ways. In spite of this, the units remain detached from one another. Then we put into the sack a prearranged connected structure made from units exactly similar to those already within the sack.
>
> ...
>
> Now we agitate the sack again in the same random and vigorous manner, with the seed structure jostling about among the neutral units. This time we find that replicas of the seed structure have been assembled from the formerly neutral or "lifeless" material [86].

This thesis has made significant progress towards Penrose's dream of an artificial self-duplicating system. There is still much work to be done, but we have created hardware and algorithms that enable duplication of 2D objects with our Smart Pebbles programmable matter system. Even more, the objects being duplicated need not be composed of units identical to those used to produce the duplicate. The resulting duplicates can be magnified copies of the original objects that are inherently imbued with sensing, computation, and actuation abilities absent from the original.

The document was dedicated to exploring the thesis statement:

Digital fabrication can be accomplished with smart particles capable of self-disassembly.

Chapter 3 presented the Smart Pebble hardware: 12mm intelligent particles that are capable of self-disassembly. Chapter 5 then demonstrated how we implemented reliable nearest neighbor communication between neighboring modules. Low level communication forms the foundation of the high level shape formation algorithms that we explored in Chapters 6–8. Specifically, Chapter 7 presents an algorithm for distributed shape duplication and hardware experiments demonstrating the algorithm executing on the Smart Pebble modules. While there are always areas for improvement, we must conclude that it is indeed possible to achieve digital fabrication using the Smart Pebbles programmable matter system. The remainder of this chapter first summarizes our contributions and then explores limitations of the system along with areas for future improvement.

9.1 Contributions

This thesis makes a number of hardware contributions to programmable matter and modular robotic systems:

- Small electropermanent magnets that can be used for mechanical bonding, inter-module communication, and module-to-module power transfer. Theses electropermanent magnets are simply to control and have zero static power consumption. Mated pairs of magnets can support over 3N.
- A set of 50 cubic modules, each 12mm per side, capable of bonding with their neighbors to form arbitrary structures in 2D. The modules are among the smallest autonomous robots capable of actuation, sensing, and computation.
- Several test fixtures, including an inclined vibration table, that allow the user to power and communicate with Smart Pebbles system.

To support the Smart Pebble hardware, this thesis contributes several pieces of key software:

- Low-level communication code which makes the inherently fragile inter-module communication process more robust. This code also monitors communication link status, and enables the system to route messages around communication links that are dynamically broken.

- A simulator that allows us to develop and easily debug high-level shape formation algorithms. The simulator executes the same code that runs the Smart Pebbles. Each Pebble runs as an independent process, and because the processes communicate with UDP packets, they can be distributed across multiple physical machines.

Finally, this thesis also makes a number of algorithmic contributions:

- A distributed duplication algorithm that allows the system to produce many, possibly magnified, copies of an original passive object that is submerged in a collection of Smart Pebble modules. The algorithm is robust to broken communication links and modules missing from the square lattice.

- A 3D extension to the 2D shape duplication algorithm. The 3D algorithm enables the duplication of complex shapes by partitioning the initial configuration of modules in slices.

- Algorithms that enable the organized disassembly of a group of Smart Pebble modules. Because the modules rely on their neighbors for power, the disassembly process must carefully determine the order in which mechanical bonds between neighboring modules are broken. Our algorithm ensures that a module does not lose power before it has broken all necessary bonds with its neighbors.

9.2 Limitations

There are obvious limitations to the current Smart Pebbles system. For example, the Smart Pebbles are larger than desired; the modules are only capable of forming 2D structures; the inter-module bonding strength is not yet sufficient to allow the structures to be used as tools. In addition to these obvious areas for improvement, there are several more subtle limitations to the system that should be addressed in future iterations.

The hardware is still fragile. Despite having manufacturing fifty Smart Pebble modules, typically only half are functional at any given time. We made many trade-offs in the design process, and to minimize the volume of the modules, we pushed the limits of the components and thereby sacrificed robustness. For example, each module runs from a 20V supply, but we use a linear regulator rated for 20mA to supply 18mA at 5V to the processor. To dissipate the heat generated by the regulator, we keep a fan pointed at the modules during all experiments. As another example of a design trade-off, the PFETs which drive the EP magnets and the 150μF reservoir capacitors are all operating at their rated voltages (20V) with no margin. When the PFETs fail, the 20V rail can be shorted to ground. To prevent catastrophic damage, we have a 22Ohm series resistor that acts like a fuse. It is the most common component that we replace.

Inter-module communication is not error free. Bit errors are acceptable so long as they are corrected, or at least detected, but the system cannot always accomplish this. While not common, bit errors in the messages do propagate to the high-level algorithms. The current 7-bit CRC checksum is not sufficient to protect messages that may be several hundred bits long. The decision to use a simple 7-bit checksum was another design trade-off designed to save code space, (which is completely occupied by the 2D duplication algorithm), and reduce the amount of processor time devoted to transmitting and receiving messages.

The routing algorithms are subject to deadlock. With just two modules, it is easy to envision a scenario where each module's receive and payload buffers are full. Suppose that the message in each payload buffer instructs the module to send a routing message to its neighbor. Each modules will attempt to send this routing message, but until it is successfully delivered, the payload buffer will not be emptied. Of course, the message will not be delivered because the receive buffers into which the messages should be transferred are already full. Furthermore, the receive buffers cannot be emptied because each contains a routing message whose payload must be moved to the payload buffers but, as already stated, those are occupied as well. As a result, the two modules have entered a deadlock situation. In practice, we do not see deadlock often, but we have identified it as the cause of failure in a small number of 3D duplication experiments.

The 3D shape duplication algorithm is not as robust to defects in the original block of material

as the 2D algorithm. In particular, 3D duplication may fail if there are modules missing from the lattice in the vicinity of the duplicate shape. In particular, if the missing modules should be part of the duplicate or its border, some slice leader will never receive the correct number of confirmation messages, so the duplication process will hang.

The 3D duplication process does not yet support magnification or the production of multiple copies of the original shape. Magnification and multiple copies are difficult because they break the isolation of each layer. A single original border module must sender border identification messages to several conjugate border modules in multiple slices. While this is theoretically possible, it requires a great deal of bookkeeping to ensure that all necessary messages are received and confirmed.

The Sandbox simulator is not as scalable as we had hoped. On a 2.4GHz quad-core Intel Core2 processor with 4GB of RAM, the simulator begins to struggle as the number of modules approaches 1000. We have run a few simulations with over 1000 modules, but they run too slowly to be practical. The simulator is not scalable because each process wakes from sleep on a regular basis to check for incoming messages, regardless of whether a message is actually available. A better architecture would keep each module in its sleep state until a message arrives. The number of IP port numbers also limits the simulator to several thousand modules. There are approximately 6,0000 ports available on most systems. If each simulated module uses 6 to communicate with its neighbors and another to communicate with the GUI, we will be limited to simulating 8,000 modules.

9.3 Lessons Learned

We learned several important lessons while working to develop the Smart Pebbles system:

- With respect to hardware, we discovered that small manufacturing defects can have a noticeable effect on the reliability on the self-assembly process. The mechanical variation from module to module is not large, but it can be large enough to prevent an ensemble of modules from self-assembling into a close-packed lattice. There are several mechanical aspects that

were important. In addition to overall module dimensions, the orthogonality of neighbors connectors was also important. While the brass frames around which the flexible circuit boards were wrapped ensured that the module faces where nearly orthogonal, ensuring that the EP magnets were mounted parallel to the faces in which they were embedded was more difficult. As a result, one pole often protruded farther from the exterior of the module than the other. This effect was exacerbated by mechanical variance in the EP magnets themselves. Often, the two pole pieces were not perfectly co-planar or parallel.

- Completely autonomous systems are particularly prone to failure. When manually assembling the Smart Pebbles modules into an initial block of material module-by-module, it is easy to identify and replace a misshapen module with a more appropriate module. In comparison, when the system is self-assembling, this is impossible. Before starting the self-assembly process, we had to ensure that the modules being used were as uniform as possible. Except when using only a few modules, this was difficult. We learned that for a large system to be robust, it must be able to tolerate significant variations in module size and shape.

- For a system to be as robust as possible to variations in module size, we now believe that we must abandon the assumption that the modules are all a standard size and will pack into a uniform lattice. Abandoning this assumption becomes all the more important when the dimensions of the object around which the modules are packed are not integer multiples of the basic module dimensions. When self-assembling the modules around a passive shape, we had to ensure that the granularity of the passive shape matched the module size. If we did not, we found that the modules would rather align with the boundary of the passive shape than the grid positions that would allow them to bond with their neighbors. If we are to allow the modules to form non-uniform 3D lattice, the modules will almost certainly need to be spherical. No other shape will all the modules to pack as neatly around an arbitrarily shape object.

- While developing the high-level algorithms, we learned how crucial it is to have access to high-quality debugging tools. Typically, it is enough to debug an embedded system with

a few LEDs, a serial connection, or at the most, a JTAG bus. While we were able to use Atmel's proprietary debugWIRE interface to step, line-by-line, through the code running on a single module, this proved insufficient when diagnosing problems related to the interaction of multiple modules.

- While we were constrained to debugging only the root module, we came to realize that even the ability to debug any arbitrary module would be ineffective. We learned that we needed the ability to debug multiple modules simultaneously. The result of this realization was the simulator that we developed. The simulator allows us to observe the internal state of multiple modules virtually simultaneously. Additionally, we could stop and step through the code of multiple modules. Finally, by recording all inter-modules messages exchanged during an experiment, we were able to reconstruct the actions of the high-level shape formation algorithms.

- In the process of performing large-scale experiments, we learned how important it is for a distributed system to be robust to multiple points of failure. We were shocked how many different types of errors arose when performing hardware experiments. Despite the duplication algorithms working predictably in simulation, the hardware was always problematic. We saw modules behave in completely unexpected ways that, given the state of our debugging tools, we could not always explain.

- Future iterations of the system should have more program storage available. The fact that the modules experienced un-handled errors was not unexpected. In order to fit the core of the duplication algorithm into the limited memory of the Smart Pebble hardware, we had to eliminate a significant amount of error handling from the code. When a module encountered an unknown error, it ideally entered a safe state in which the processor prevented harm to the EP magnet drivers and used the LED to flash a rudimentary error code to the user. In other cases, the modules processors reset themselves and began to behave as if they had just joined the existing configuration of modules. Unfortunately, there was not enough code space available to handle these conditions.

- Mundane errors can lead to serious faults. While the duplication algorithm is capable of handling broken communication links and missing modules, it struggles to handle communication links that break or form after the duplication process has begun. Even the basic bug routing algorithm struggles to handle cases in which a module is added or removed from the system. For example, if a routing message departs from its ideal path while avoiding an obstacle but the module at which it deviated from the idea path is removed from the system, the message may never realize that it cannot reach its destination. This is because the message will never again pass through the departure point, so it cannot say with certainty that is has completely circumnavigated the obstacle it was avoiding.

- Robust error handling is more difficult and time consuming to implement than the basic application code. Communication links between neighboring modules that form after all the modules have been localized can be problematic. For example, a broken communicate link between neighboring modules on the perimeter of the shape being duplicated will cause the duplication algorithm to believe that the shape's perimeter is two units larger than it actually is. This becomes a problem when the broken communication link is healed before each of the modules on the perimeter of the original sends a border message to its conjugate module. In such a scenario, the leader module will wait indefinitely for two border confirmation messages that never arrive. Consequently, the self-disassembly process will never complete. Future improvements to the shape formation algorithms must be able to deal with these relatively common dynamic changes to the configuration of the structure. In light of all these challenges, development of any practical system will require just as much, if not more, code devoted to error handling than code for the shape formation algorithm.

9.4 Near-Term Improvements

The most obvious area for future development is improved programmable matter hardware. Hardware improvements can be divided into both short-term and long-term goals. In the immediate future, we should develop a three-dimensional version of the Smart Pebbles. The first generation

of three-dimensional Pebbles may be slightly larger than the current two-dimensional Pebbles, but the three-dimensional modules could serve as a testbed on which to experiment with new fabrication techniques that could be miniaturized later. A new version of hardware would allow us to improve the robustness of the modules using what we have learned from the current generation.

There are many extensions to the current shape duplication algorithms that should be implemented. In particular, many of the features of the two-dimensional algorithm have not yet been ported to three-dimensions. The three-dimensional algorithm is not yet capable of shape magnification or creating multiple copies of a single original shape. The three-dimensional algorithm is also not as robust as the two-dimensional algorithm. It is more sensitive to missing modules. Finally, the three-dimensional algorithm does not support automated shape placement.

We should also aim to better automate the placement of the duplicate shape within the block of host material. Currently the system makes a simplistic attempt to optimize the placement of a single unmagnified instance of the duplicate. In the future, the system should attempt to both translate and rotate multiple, magnified duplicates to achieve the most efficient packing of duplicate shapes. Additionally, the three-dimensional system should attempt to optimize the plane used to slice the initial block of material. Currently, it must be specified by the user.

We would also like to imbue the system with the ability to create more exotic shapes. Currently, the system can sense and duplicate three-dimensional concavities accessed by through small openings, but actually removing the unused modules from such cavities is difficult to impossible. The module will get jammed as the try to pass through the cavity's exit. We would like an extension to the duplication algorithm that allows us to form the duplicate in two or more pieces that are initially un-bonded with each another. Once the pieces, or sub-assemblies have been removed from the larger collection of modules, we would like the user to be able to easily bond them together to form the complete object. This approach would allow for the creation of cavities, and it would also allow the user to partition a large object into pieces that are more easily fabricated.

We would like to improve the two-dimensional routing algorithm. When the duplication algorithm is routing confirmation messages to the obstacle leader, many of the messages follow the exact same path. This is especially true when the messages are contouring around some obstacle.

Each message tightly hugs the perimeter of the obstacle. As a result, some modules experience a much higher communication load than others. Future iterations of the system should use a more intelligent approach. Messages should attempt to take alternate paths even if those paths are longer. This will decrease congestion and improve the system's running time.

Finally, we would like to investigate ways to improve the simulator. As already mentioned, it is useful for simulating collections of, at most, a few hundred modules. If we abandon the simulator's ability to accurately mimic the low-level communication of the Smart Pebbles, we should be able to construct a simulator that is capable of handling a few thousand modules. Past that, we will need to explore alternatives to using UDP packets for inter-module communication.

9.5 Looking to the Future

There are several high-level areas for potential research that have been exposed by this thesis. First, we need to look past systems that rely on the modules that pack into regular lattices. Perfectly regular packings are practically impossible to achieve unless the modules themselves are all perfectly identical. Adding a passive object to the collection of modules only further complicates the packing process and makes perfect packing more unlikely. While developing hardware that support irregular packings will be challenging, the practicality and robustness that it ultimately brings will be notable. In developing hardware modules that support irregular packing we may need to move away from using discrete connectors. When packing randomly, a module a may contact its neighbors almost anywhere over its surface. Wherever a module comes into contact with its neighbors, it must be able to form mechanical, data, and power connections. As a result, a few localized connectors per module may not be sufficient for a system to form a dense lattice of inter-connected modules.

If we do achieve a hardware module that can effectively pack into irregular lattices while bonding and communicating with its neighbors, there are a variety of algorithmic challenges that arise. Without a global coordinate system, simply localizing all of the modules in the structure will be challenging. Our current approach to shape duplication will also require refinement. In an irregular packing, there will no longer be a exact one to one correspondence between modules on the border

of the original and modules on the border of the duplicate. We will be forced to approximate the shape of the original object being duplicated. How we would do this in an optimal manner is not yet clear.

There are other potential areas for algorithmic improvement as well. This thesis has presented algorithms that require $O(1)$ space and $O(n)$ messages per module, but we must do better. Practical programmable matter systems will be composed of millions of modules. We cannot reasonably expect each module to exchange a million messages with its neighbors. We need to pursue algorithms that exchange a number of messages per module that is sub-linear. While potentially impossible in worst-case scenarios, much work remains to improve typical efficiency.

Perhaps the least explored area for additional research is how the functionality of programmable materials can be specified in a structured and provably-correct fashion. Once we have created an object from a collection of smart particles, we should explore how we can continue to leverage the intelligence, communication, sensing, and actuation abilities that are already incorporated into each module. If we treat programmable matter systems as static machines whose inherent abilities are only used to from passive shapes, we are missing the true potential of programmable matter systems. Programmable matter systems hold great potential because they can be dynamic, changing both their software and hardware to adapt to the task at hand.

Personal computers are a ubiquitous example of universal computing machines. They are capable of running any number of different pieces of software, but they cannot change their basic physical form or interact with the tangible world in an unencumbered way. In contrast, programmable matter systems approach universal machines in both an algorithmic and physical sense. They can run arbitrary pieces of software, and they can assume nearly arbitrary shapes and physical properties. While many software programming languages exist, there are no equivalents for programmable matter systems. As computer science enabled the systematic design and analysis of the software systems that have revolutionized high technology in the last sixty years, we need a new science, the science of programmable matter, that will revolutionize the next one-hundred.

Appendix A

Schematics

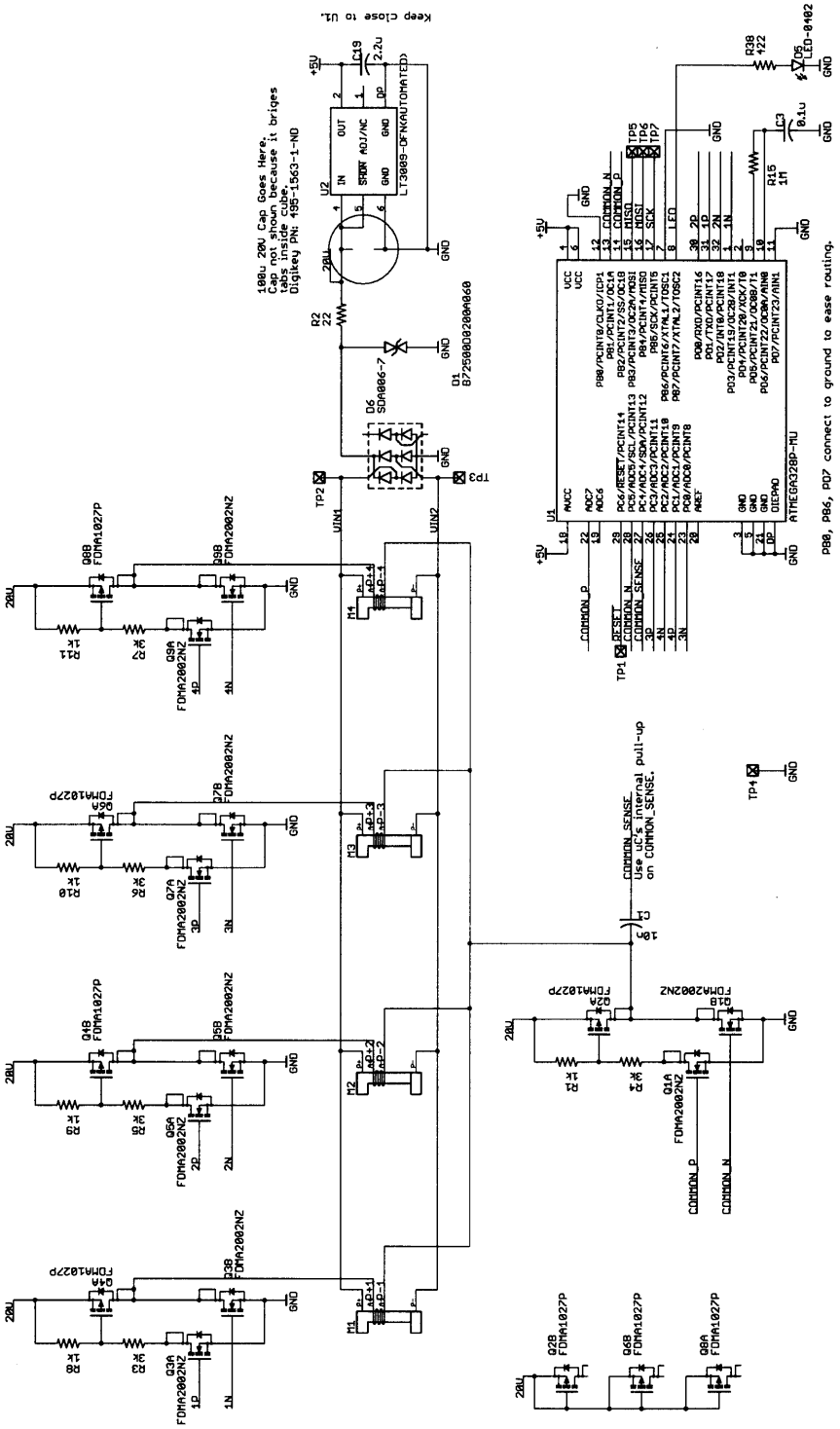


Figure A-1: The Robot Pebbles Schematic

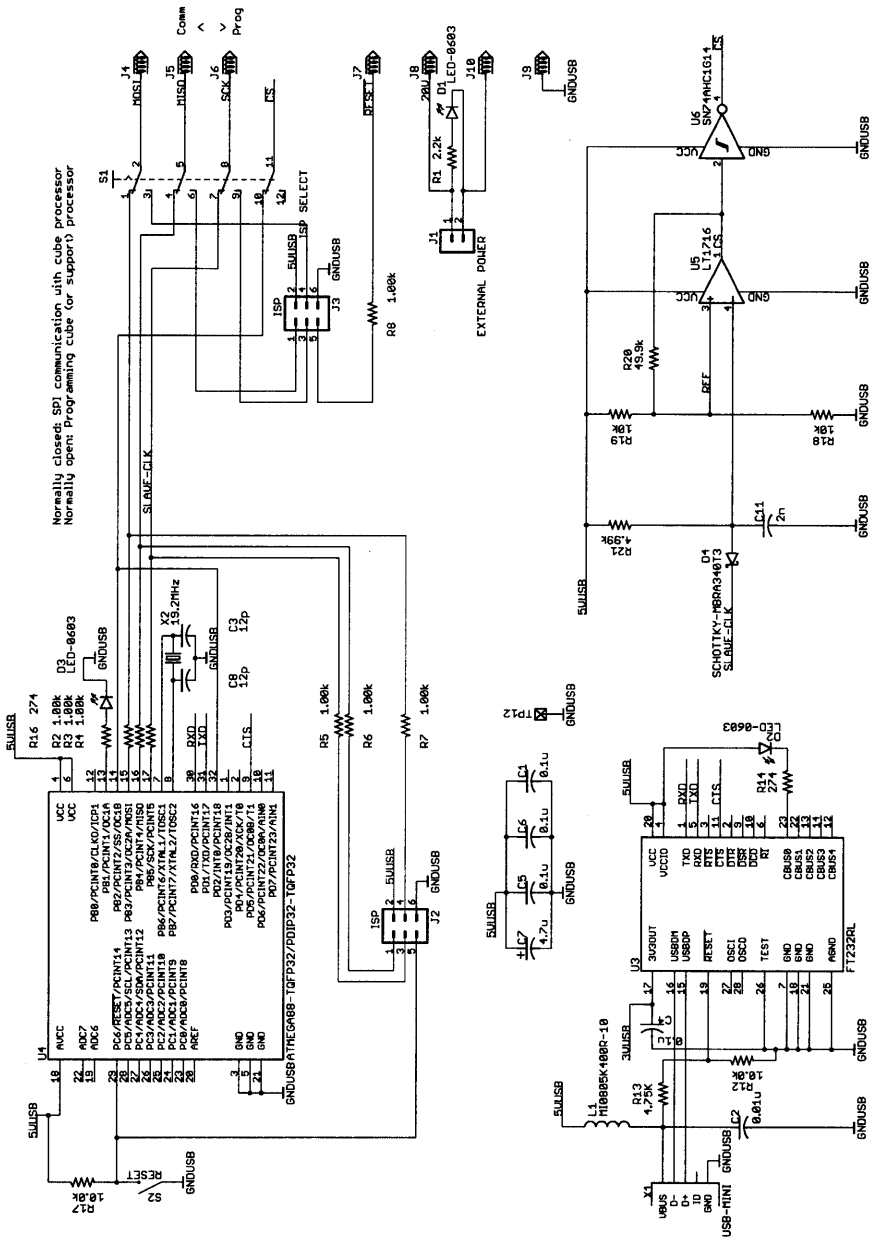


Figure A-2: The Robot Pebbles Test/Programming Fixture Schematic

Bibliography

[1] Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang. Running time and program size for self-assembled squares. In *33rd Annual ACM Symposium on Theory of Computing*, pages 740–748, 2001.

[2] Gagan Aggarwal, Michael H. Goldwasser, Ming-Yang Kao, and Robert T. Schweller. Complexities for generalized models of self-assembly. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 880–889, 2004.

[3] Byoung Kwon An. Em-cube: Cube-shaped, self-reconfigurable robots sliding on structure surfaces. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3149–3155, May 2008.

[4] Byoungkwon An and Daniela Rus. Programming and controlling self-folding sheets. In *IEEE International Conference on Robotics and Automation (ICRA)*, page In Press, May 2012.

[5] Michael Ashley-Rollman, Padmanabhan Pillai, and Michelle Goodstein. Simulating multi-million-robot ensembles. In *ICRA*, page in press, May 2011.

[6] Batteryspace. 1-2 c rate polymer li-ion cells / packs. `http://www.batteryspace.com/1-2cratepolymerli-ioncellspacks.aspx`, 2012.

[7] L. Berry, L. Renaud, P. Kleinmann, P. Morin, M. Armenean, and H. Saint-Jalmes. Implantable solenoidal microcoil for nuclear magnetic resonance spectroscopy. In *IEEE-EMBS Special Topic Conference on Microtechnologies in Medicine and Biology*, pages 171–174, October 2000.

[8] J. Bishop, S. Burden, E. Klavins, R. Kreisberg, W. Malone, N. Napp, and T. Nguyen. Programmable parts: A demonstration of the grammatical approach to self-organization. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3684–3691, August 2005.

[9] Scott T. Brittain, Yuki Sugimura, Oliver J. A. Schueller, Anthony G. Evans, and George M. Whitesides. Fabrication and mechanical performance of a mesoscale space-filling truss system. *Journal of Microelectromechanical Systems*, 10(1):113–120, March 2001.

[10] William Butera. Text display and graphics control on a paintable computer. In *Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 45–54, July 2007.

[11] A. Castano and P. Will. Mechanical design of a module for reconfigurable robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2203–2209, 2000.

[12] Andres Castano, Alberto Behar, and Peter Will. The conro modules for reconfigurable robots. *IEEE Transactions on Mechatronics*, 7(4):403–409, December 2002.

[13] M.L. Chan, P. Fonda, C. Reyes, J. Xie, H. Najar, L. Lin, K. Yamazaki, and D.A. Horsley. Micromachining 3d hemispherical features in silicon via micro-emd. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 289–292, Feburary 2012.

[14] Ian Chen, Bruce MacDonald, Burkhard Wunsche, Geoffrey Biggs, and Tetsuo Kotoku. A simulation environment for openrtm-aist. In *SI International*, pages 113–117, 2009.

[15] Chih-Jung Chiang and Gregory S. Chirikjian. Modular robot motion planning using similarity metrics. *Autonomous Robots*, 10:91–106, 2001.

[16] Gregory Chirikjian, Amit Pamecha, and Imme Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of Robotic Systems*, 13(5):317–388, 1996.

[17] Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 449–455, May 1994.

[18] David Christensen, David Brandt, Kasper Stoy, and Ulrik Pagh Schultz. A unified simulator for self-reconfigurable robots. In *IROS*, pages 870–876, September 2008.

[19] David Johan Christensen and Kasper Stoy. Select a meta-module to shape-change the atron self-reconfigurable robot. In *IEEE International Conference on Robotics and Automation*, pages 2532–2538, May 2006.

[20] Klaus Cicha, Zhiquan Li, Klaus Stadlmann, Aleksandr Ovsianikov, Ruth Markut-Kohl, Robert Liska, and Jurgen Stampfl. Evaluation of 3d structures fabricated with two-photon-photopolymerization by using ftir spectroscopy. *Journal of Applied Physics*, (110):064911, September 2011.

[21] Adam Cohen, Gang Zhang, Fan-Gang Tseng, Uri Frodis, Florian Mansfeld, and Peter Will. Efab: Rapid, low-cost desktop micromachining of high aspect ratio true 3-d mems. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 244–251, 1999.

[22] Goldstein Seth Copen and Todd C. Mowry. Claytronics: An instance of programmable matter. In *Wild and Crazy Ideas Session of ASPLOS*, Boston, MA, October 2004.

[23] Bruce Donald, Christopher G. Levey, Craig D. McGray, Igor Paprotny, and Daniela Rus. An untethered, electrostatic, globally controllable mems micro-robot. *Journal of Microelectromechanical Systems*, 15(1):1–15, February 2006.

[24] Bruce R. Donald, Christopher G. Levey, and Igor Paprotny. Planar microassembly by parallel actuator of mems microrobots. *Journal of Microelectromechanical Systems*, 17(4):789–808, August 2008.

[25] Dprsim. `http://www.pittsburgh.intel-research.net/dprweb/`.

[26] A. C. Fischer, N. Roxhed, T. Haraldsson, N. Heinig, G. Stemme, and F. Niklaus. Fabrication of high aspect ratio through silicon vias (tsvs) by magnetic assembly of nickel wires. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 37–40, January 2011.

[27] Robert Fitch and Zack Butler. Million module march: Scalable locomotion for large self-reconfiguring robots. *International Journal of Robotics Research*, 27(3-4):331–343, March/April 2008.

[28] Toshio Fukuda and Seiya Nakagawa. Dynamically reconfigurable robotic system. In *IEEE International Conference on Robotics and Automation*, pages 1581–1586, April 1988.

[29] Stanislav Funiak, Padmanabhan Pillai, Michael P. Ashley-Rollman, Jason D. Campbell, and Seth Copen Goldstein. Distributed localization of modular robot ensembles. *International Journal of Robotics Research*, 28(8):946–961, August 2009.

[30] David H. Garcias, Joe Tien, Tricia L. Breen, Carey Hsu, and George M. Whitesides. Forming electrical networks in three dimensions by self-assembly. *Science*, 289(5482):1170–1172, August 18 2000.

[31] B. Gerkey, R. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Intl. Conf. on Advanced Robotics*, June 2003.

[32] Kyle Gilpin. Distributed algorithms for self dissasembly for modular robots. M.Eng. and S.B. Thesis, June 2006.

[33] Kyle Gilpin, Ara Knaian, and Daniela Rus. Robot pebbles: One centimeter robotic modules for programmable matter through self-disassembly. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[34] Kyle Gilpin, Keith Kotay, Daneila Rus, and Iuliu Vasilescu. Miche: Modular shape formation by self-disassembly. *International Journal of Robotics Research*, 27:345–372, 2008.

[35] Kyle Gilpin, Keith Kotay, and Daniela Rus. Miche: Modular shape formation by self-disassembly. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2241–2247, April 2007.

[36] Kyle Gilpin and Daniela Rus. Modular robot systems: From self-assembly to self-disassembly. *IEEE Robotics and Automation Magazine*, 17(3):38–53, September 2010.

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

[38] Saul Griffith, Dan Goldwater, and Joseph M. Jacobson. Robotics: Self-replication from random parts. *Nature*, 437:636, September 28 2005.

[39] Gregory J. Hamlin and A. C. Sanderson. Tetrobot: A modular system for hyper-redundant parallel robotics. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 154–159, May 1995.

[40] E. Hawkes, B. An, N. M. Benbernou, H. Tanaka, S. Kim, E. D. Demaine, D. Rus, and R. J. Wood. Programmable matter by folding. *Proceedings of the National Academy of Sciences*, 107(28):12441–12445, 2010.

[41] K. Hosokawa, I. Shimoyama, and H. Miura. Dynamics of self-assembling systems: Analogy with chemical kinematics. *Artificial Life*, 1(4):413–427, 1994.

[42] Kazuo Hosokawa, Takehito Tsujimori, Teruo Fujii, Hayato Kaetsu, Hajime Asama, Yoji Kuroda, and Isao Endo. Self-organizing collective robots with morphogenesis in a vertical plane. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2858–2683, May 1998.

[43] Jr. John Amend and Hod Lipson. Shape-shifting materials for programmable structures. In *International Conference on Ubiquitous Computing: Workshop on Architectural Robotics*, September 2009.

[44] Chris Jones and Maja J. Matarić. From local to global behavior in intelligent self-assembly. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 721–726, 2003.

[45] Morten Winkler Jørgensen, Esben Hallundbæk Østergaard, and Henrik Hautop Lund. Modular atron: Modules for a self-reconfigurable robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2068–2073, September 2004.

[46] Akiya Kamimura, Haruhisa Kurokawa, Eiichi Yoshida, Satoshi Murata, Kohji Tomita, and Shigeru Kokaji. Automatic locomotion design and experiments for a modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 10(3):314–325, June 2005.

[47] Mustafa Emre Karagozler, Seth Copen Goldstein, and J Robert Reid. Stress-driven mems assembly + electrostatic forces = 1mm diameter robot. In *IEEE Conference on Intelligent Robots and Systems (IROS)*, pages 2763–2769, October 2009.

[48] Jonathan Kelly and Hong Zhang. Combinatorial optimization of sensing for rule-based planar distributed assembly. In *IEEE International Conference on Intelligent Robots and Systems*, pages 3728–3734, 2006.

[49] H. Keum, A. Carlson, J.D. Eisenhaure, J.A. Rogers, and S. Kim. Deterministically assembled three-dimensional silicon microstructures using elastomeric stamps. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 224–227, February 2012.

[50] Brian T. Kirby, Burak Aksak, Jason D. Campbell, James F. Hoburg, Todd C. Mowry, Padmanabhan Pillai, and Seth Copen Goldstein. A modular robotic system using magnetic force effectors. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2787–2793, 2007.

[51] Ara Knaian. *Electopermanent Magnetic Connectors and Actuators: Devices and Their Application in Programmable Matter*. PhD thesis, Massachusetts Institute of Technology, 2010.

[52] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks*, pages 145–154, 2004.

[53] Michihiko Koseki, Kengo Minami, and Norio Inou. Cellular robots forming a mechanical structure (evaluation of structural formation and hardware design of "chobie ii"). In *Proceedings of 7th International Symposium on Distributed Autonomous Robotic Systems (DARS04)*, pages 131–140, June 2004.

[54] Keith Kotay, Daneila Rus, Marsette Vona, and Craig McGray. The self-reconfiguring robotic molecule. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 424–431, 1998.

[55] Keith Kotay and Daniela Rus. Motion synthesis for the self-reconfiguring robotic molecule. In *IEEE International Conference on Intelligent Robots and Systems*, pages 843–851, October 1998.

[56] Keith Kotay and Daniela Rus. Algorithms for self-reconfiguring molecule motion planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2000.

[57] J. Krahn, D. Sameoto, and C. Menon. Controllable biomimetic adhesion using embedded phase change material. *Smart Materials and Structures*, 20(1):015014, January 2011.

[58] K. Kuribayashi-Shigetomi, H. Onoe, and S. Takeuchi. Self-folding cell origami: Batch process of self-folding 3d cell-laden microstructures actuated by cell traction force. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 72–75, 2012.

[59] M. Kurihara, Y.J. Heo, K. Kuribayashi-Shigetomi, and S. Takeuchi. 3d laser lithography combined with parylene coating for the rapid fabrication of 3d microstructures. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 196–199, February 2012.

[60] Haruhisa Kurokawa, Satoshi Murata, Eiichi Yoshida, Kohji Tomita, and Shigeru Kokaji. A 3-d self-reconfigurable structure and experiments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 860–865, October 1998.

[61] Haruhisa Kurokawa, Kohji Tomita, Akiya Kamimura, Eiichi Yoshida, Shigeru Kokahji, and Satoshi Murata. Distributed self-reconfiguration control of modular robot m-tran. In *IEEE International Conference on Mechatronics and Automation*, pages 254–259, July 2005.

[62] Haruhisa Kurokawa, Kohji Tomita, Eiichi Yoshida, Satoshi Murata, and Shigeru Kokaji. Motion simulation of a modular robotic system. In *IECON*, pages 2473–2478, 2000.

[63] Michael D. M. Kutzer, Matthew S. Moses, Christopher Y. Brown, David H. Scheidt, Gregory S. Chirikjian, and Mehran Armand. Design of a new independently-mobile reconfigurable modular robot. In *IEEE International Conference on Robotics and Automation*, pages 2758–2764, May 2010.

[64] Kiju Lee and Gregory S. Chirikjian. An autonomous robot that duplicates itself from low-complexity components. In *IEEE International Conference on Robotics and Automation*, pages 2771–2776, May 2010.

[65] Ben Leong, Barbara Liskov, and Robert Morris. Geographic routing without planarization. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[66] V. J. Lumelski and A. A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063, 1986.

[67] Andreas Lyder, Ricardo Franco Mendoza Garcia, and Kasper Stoy. Mechanical design of odin, an extendable heterogeneous deformable modular robot. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 883–888, September 2008.

[68] Andreas Lyder, Henrik Gordon Peterson, and Kasper Stoy. Representation and shape estimation of odin, a parallel under-actuated modular robot. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 5275–5280, October 2009.

[69] Bruce J. MacLennan. Universally programmable intelligent matter summary. In *IEEE International Conference on Nanotechnology (NANO)*, pages 405–408, 2002.

[70] Daniel Marbach and Auke Jan Ijspeert. Online optimization of modular robot locomotion. In *IEEE International Conference on Mechatronics and Automation*, pages 248–253, July 2005.

[71] Andrew D. Marchese, Harry Asada, and Daniela Rus. Controlling the locomotion of a separated inner robot from and outer robot using electropermanent magnets. In *IEEE International Conference on Robotics and Automation (ICRA)*, page in press, 2012.

[72] Microsoft robotics developer studio. `http://www.microsoft.com/robotics/`.

[73] Shuhei Miyashita, Marco Kessler, and Max Lungarella. How morphology affects self-assembly in a stochastic modular robot. In *IEEE International Conference on Robotics and Automation*, pages 3533–3538, May 2008.

[74] Satoshi Murata, Kiyoharu Kakomura, and Haruhisa Kurokawa. Docking experiments of a modular robot by visual feedback. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 625–630, October 2006.

[75] Satoshi Murata, Haruhisa Kurokawa, and Shigeru Kokaji. Self-assembling machine. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 441–448, 1994.

[76] Satoshi Murata, Eiichi Yoshida, Akiya Kamimura, Haruhisa Kurokawa, Kohji Tomita, and Shigeru Kokaji. M-tran: Self-reconfigurable modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 7(4):431–441, December 2002.

[77] Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, July 2002.

[78] Nils Napp, Samuel Burden, and Eric Klavins. The statistical dynamics of programmed self-assembly. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1469–1476, May 2006.

[79] Michael T. Northern, Christian Greiner, Eduard Arzt, and Kimberly L. Turner. A gecko-inspired reversible adhesive. *Advanced Materials*, 20(20):3905–3909, September 2008.

[80] Open dynamics engine. `http://www.ode.org/`, 2010.

[81] Esben Hallundbæk Østergaard and Henrik Hautop Lund. Evolving control for modular robotic units. In *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 886–892, July 2003.

[82] Raymond Oung, Frederic Bourgault, Matthew Donovan, and Raffaello D'Andrea. The distributed flight array. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[83] Amit Pamecha, I. Ebert-Uphoff, and Gregory S. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–45, 1997.

[84] Chytra Pawashe, Steven Floyd, Eric Diller, and Metin Sitti. Two-dimensional autonomous microparticle manipulation strategies for magnetic microrobots in fluidic environments. *IEEE Transactions on Robotics*, 28(2):467–477, April 2012.

[85] Chytra Pawashe, Steven Floyd, and Metin Sitti. assembly and disassembly of magnetic mobile micro-robots towards 2-d reconfigurable micro-systems. In *International Symposium on Robotics Research*, 2009.

[86] L. S. Penrose. Self-reproducing machines. *Scientific American*, 200(6):105–114, June 1959.

[87] Physx. `http://developer.nvidia.com/object/physx.html`.

[88] Padmanabhan Pillai, Jason Campbell, Gautam Kedia, Shishir Moudgal, and Kaushik Sheth. A 3d fax machine based on claytronics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 4728–4735, October 2006.

[89] Player project. `http://playerstage.sourceforge.net/`.

[90] Konstantine C. Prevas, Chem Ünsal, Mehmet Önder Efe, and Pradeep K. Khosla. A hierarchical motion planning strategy for a uniform self-reconfigurable modular robotic system. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 787–792, May 2002.

[91] Benjamin D. Rister, Jason Campbell, Padmanabhan Pillai, and Todd C. Mowry. Integrated debugging of large modular robot ensembles. In *ICRA*, pages 2227–2234, April 2007.

[92] Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *32rd Annual ACM Symposium on Theory of Computing*, pages 459–468, 2000.

[93] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *IEEE International Conference on Robotics and Automation (ICRA)*, page In Press, May 2012.

[94] Michael Rubenstein and Wei-Min Shen. Scalable self-assembly and self-repair in a collective of robots. In *IEEE International Conference on Intelligent Robots and Systems*, pages 1484–1489, October 2009.

[95] Michael Rubenstein and Wei-Min Shen. Automatic scalable size selection for the shape of a distributed robotic collective. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, page in press, October 2010.

[96] Daniela Rus and Marsette Vona. A basis for self-reconfiguring robots using crystal modules. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2194–2202, October 2000.

[97] Daniela Rus and Marsette Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *International Journal of Robotics Research*, 22(9):699–715, 2003.

[98] Behnam Salemi, Mark Moll, and Wei-Min Shen. Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 3636–3641, October 2006.

[99] N.S. Shaar, G. Barbastathis, and C. Livermore. Cascaded mechanical alignment for assembling 3d mems. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 1064–1068, January 2008.

[100] Wei-Min Shen and Peter Will. Docking in self-reconfigurable robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1049–1054, October 2001.

[101] Masahiro Shimizu, Akio Ishiguro, and Toshihiro Kawakatsu. A modular robot that exploits a spontaneous connectivity control mechanism. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1899–1904, August 2005.

[102] Masahiro Shimizu, Takafumi Mori, and Akio Ishiguro. A development of a modular robot that enables adaptive reconfiguration. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 174–179, October 2006.

[103] Masahiro Shimizu and Kenji Suzuki. A self-repairing structure for modules and its control by vibrating actuation mechanisms. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4281–4286, May 2009.

[104] P. S. Sreetharan, J. P. Whitney, M. D. Strauss, and R. J. Wood. Monolithic fabrication of millimeter-scale machines. *Journal of Micromechanics and Microengineering*, page In Press, 2012.

[105] John W. Suh, Samuel B. Homans, and Mark Yim. Telecubes: Mechanical design of a module for self-reconfigurable robotics. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4095–4101, May 2002.

[106] Naotaka Tanaka, Michihiro Kawashita, Yasu Yoshimura, Toshihide Uematsu, Masahiko Fujiwasa, Hirohisa Shimokawa, Nobuhiro Kinoshita, Takahiro Naito, Takafumi Kikuchi, and Takashi Akazawa. Characterization of mos transistors after tsv fabricatin and 3d-assembly. In *Electronics Systemintegration Technology Conference*, pages 131–134, 2008.

[107] Sindy K. Y. Tang, Ratmir Derda, Aaron D. Mazzeo, and George M. Whitesides. Reconfigurable self-assembly of mesoscale optical components at a liquid-liquid interface. *Advanced Materials*, 23:2413–2418, 2011.

[108] Michael Tolley, J Hiller, and Hod Lipson. Evolutionary design and assembly planning for stochastic modular robots. In *IEEE Conference on Intelligent Robotics and Systems (IROS)*, pages 73–78, October 2009.

[109] Michael Tolley and Hod Lipson. Fluidic manipulation for scalable stochastic 3d assembly of modular robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2473–2478, May 2010.

[110] Michael T. Tolley, Michael Kalontarov, Jonas Neubert, David Erickson, and Hod Lipson. Stochastic modular robotic systems: A study of fluidic assembly strategies. *IEEE Transactions on Robotics*, 26(3):518–530, June 2010.

[111] Michael T. Tolley, Mekala Krishnan, David Erickson, and Hod Lipson. Dynamically programmable fluidic assembly. *Applied Physics Letters*, 93(254105), December 2008.

[112] Michael T. Tolley and Hod Lipson. On-line assembly planning for stochastically reconfigurable systems. *International Journal of Robotics Research*, 30(13):1566–1584, November 2011.

[113] Michael T. Tolley and Hod Lipson. Programmable 3d stochastic fluidic assembly of cm-scale modules. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4366–4371, September 2011.

[114] Cem Ünsal and Pradeep K. Khosla. Mechatronic design of a modular self-reconfiguring robotic system. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1742–1747, April 2000.

[115] Paulina Varshavskaya, Leslie P. Kaelbling, and Daniela Rus. Learning distributed control for modular robots. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2648–2653, 2004.

[116] Karthik Visvanathan, Tao Li, and Togesh B. Gianchandani. 3d-soule: A fabrication process for large sale integration and micromachining of spherical structures. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 45–48, January 2011.

[117] J. W. von Honschoten, A. Lengrain, J. W. Berenschot, L. Abelmann, and N. R. Tas. Micro-assembly of three dimensional thetrahedra by capillary forces. In *IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 288–291, 2011.

[118] Jennifer E. Walter, Elizabeth M. Tsai, and Nancy M. Amato. Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. *IEEE Transactions on Robotics*, 21(4):621–631, August 2005.

[119] Cyberbotics–webots. `http://www.cyberbotics.com/overview`.

[120] Justin Werfel. *Anthills Built to Order: Automating Construction with Artificial Swarms*. PhD thesis, Massachusetts Institute of Technology, 2006.

[121] Paul White, Kris Kopanski, and Hod Lipson. Stochastic self-reconfigurable cellular robotics. In *IEEE Conference on Robotics and Automation*, pages 2888–2893, April 2004.

[122] Paul White, Victor Zykov, Josh Bongard, and Hod Lipson. Three dimensional stochastic reconfiguration of modular robots. In *Robotics Science and Systems*, June 2005.

[123] Paul J. White, Michael L. Posner, and Mark Yim. Strength analysis of miniature folded right angle tetrahedron chain programmable matter. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2785–2790, 2010.

[124] G. Whitesides and B. Grzybowski. Self-assembly at all scales. *Science*, 295:2418–21, March 2002.

[125] George M. Whitesides and Mila Boncheva. Beyond molecules: Self-assembly of mesoscope and macroscopic components. *Proceedings of the National Academy of Sciences*, 99(8):4769–4774, April 16 2002.

[126] J. P. Whitney, P. S. Sreetharan, K. Y. Ma, and R. J. Wood. Pop-up book mems. *Journal of Micromechanics and Microengineering*, 21:115021, October 2011.

[127] Lei Yang, Wei Liu, Chunqing Wang, and Yanhong Tian. Self-assembly of three-dimensional microstructures in mems via fluxless laser reflow soldering. In *IEEE International Conference on Electronic Packaging Technology and High Density Packaging*, pages 1148–1151, 2011.

[128] Se Young Yang, Hyung ryul Johnny Choi, Martin Deterre, and George Barbastathis. Nanostructured origami folding of patternable resist for 3d lithography. In *IEEE International Conference on Optical MEMS and Nanophotonics*, pages 37–38, 2010.

[129] Mark Yim. A reconfigurable modular robot with many modes of locomotion. In *JSME International Conference on Advanced Mechatronics*, pages 283–288, 1993.

[130] Mark Yim. New locomotion gaits. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2508–2514, 1994.

[131] Mark Yim, David G. Duff, and Kimon D Roufas. Polybot: a modular reconfigurable robot. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 514–520, April 2000.

[132] Mark Yim and Sam Homans. Digital clay. `www2.parc.com/spl/projects/modrobots/lattice/digitalclay/index.html`, 2002.

[133] Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular self-reconfigurable robot systems: Challenges and opportunities for the future. *IEEE Robotics and Automation Magazine*, 14(1):43–52, March 2007.

[134] Mark Yim, Babak Shirmohammadi, Jimmy Sastra, Michael Park, Michael Dugan, and C.J. Taylor. Towards robotic self-reassembly after explosion. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2767–2772, November 2007.

[135] Mark Yim, Ying Zhang, Kimon Roufas, David Duff, and Craig Eldershaw. Connecting and disconnecting for self–reconfiguration with polybot. In *IEEE/ASME Transaction on Mechatronics, special issue on Information Technology in Mechatronics*, 2003.

[136] Eiichi Yoshida, Satoshi Murata, Shigeru Kokaji, Akiya Kamimura, Kohji Tomita, and Haruhisa Kurokawa. Get back in shape! a hardware prototype self-reconfigurable modular microrobot that uses shape memory alloy. *IEEE Robotics and Automation Magazine*, 9(4):54–60, 2002.

[137] Eiichi Yoshida, Satoshi Murata, Kohji Tomita, Haruhisa Kurokawa, and Shigeru Kokaji. Distributed formation control for a modular mechanical system. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 1090–1097, 1997.

[138] Chih-Han Yu, Kristina Haller, Donald Ingber, and Radhika Nagpal. Morpho: A self-deformable modular robot inspired by cellar structure. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 3571–3578, September 2008.

[139] Victor Zykov, Efstathios Mytilinaios, Mark Desnoyer, and Hod Lipson. Evolved and designed self-reproducing modular robotics. *IEEE Transactions on Robotics*, 23(2):308–319, April 2007.