A Distributed Algorithm for 2D Shape Duplication with Smart Pebble Robots

Kyle Gilpin and Daniela Rus

***Abstract*— We present our digital fabrication technique for manufacturing active objects in 2D from a collection of smart particles. Given a passive model of the object to be formed, we envision submerging this original in a vat of smart particles, executing the new shape duplication algorithm described in this paper, and then brushing aside any extra modules to reveal both the original object and an exact copy, side-by-side. Extensions to the duplication algorithm can be used to create a magnified version of the original or multiple copies of the model object. Our novel duplication algorithm uses a distributed approach to identify the geometric specification of the object being duplicated and then forms the duplicate from spare modules in the vicinity of the original.**

This paper details the duplication algorithm and the features that make it robust to (1) an imperfect packing of the modules around the original object; (2) missing communication links between neighboring modules; and (3) missing modules in the vicinity of the duplicate object(s). We show that the algorithm requires $O(1)$ storage space per module and that the algorithm exchanges $O(n)$ messages per module. Finally, we present experimental results from 60 hardware trials and 150 simulations. These experiments demonstrate the algorithm working correctly and reliably despite broken communication links and missing modules.

I. Introduction

In this paper, we present a new approach to rapidly manufacturing active objects using a large collection of smart particles capable of communicating and bonding with their neighbors. Traditional manufacturing creates devices using a hierarchical approach that is organized as a sequence that includes mechanical design, fabrication, electronic design, PCB production, assembly, and finishing. In contrast, we package the necessary mechanical, perceptual, and computational capabilities in small, intelligent building blocks that integrate actuation, sensing, communication, and control. Manufacturing in our proposed system consists of selecting the correct aggregation of these modules for a required shape or design. By using a homogeneous set of smart modules instead of heterogeneous, task-specific materials, we (1) allow a device's components to be easily recycled and reused in a different application; (2) automate the fabrication of complex structures; and (3) imbue objects with inherent sensing and computation capabilities.

Figure 1 illustrates our approach to fabrication by distributed shape duplication. Given a passive physical model of the object to be manufactured, the distributed algorithm that we present in this paper identifies the geometric specification of the object and forms one or more, potentially magnified, replicas from our programmable matter particles in a single-step process. The resulting duplicate is endowed not only with the shape of the original passive model, but also intelligence, sensing, and communication capabilities.

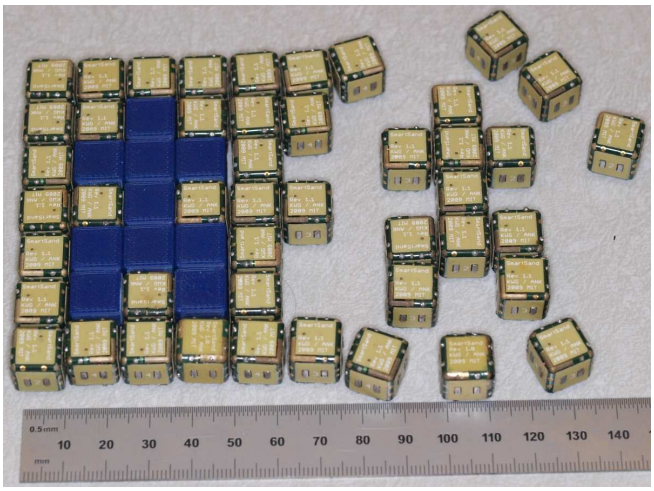

Fig. 1. The distributed duplication algorithm presented in this paper uses an ensemble of smart modules (here our Robot Pebbles) packed around a passive object to sense that object's shape and then make a duplicate from the spare modules in the vicinity of the original. (While this figure is only illustrative, the paper demonstrates the algorithm's applicability with a variety of smaller scale hardware experiments.)

This approach to digital fabrication by shape distribution is generic and independent of the architecture of the individual particles that compose the programmable matter system. In this paper we illustrate the algorithms in the context of the Robot Pebbles [1] which are 1cm cubic computers with programmable connectors and neighbor-to-neighbor communication. Current manufacturing technology lower-bounds the size of the Pebbles, (and therefore the resolution of the objects that we can duplicate), but as manufacturing techniques evolve, we will be able to further miniaturize these modules. Eventually, we hope to produce sub-millimeter *Smart Sand* particles that have all the communication and computation capabilities of the current Robot Pebbles.

Irrespective of the dimension, bonding mechanism, or communication interface of the programmable matter modules, the algorithm we present here operates the same way. A passive object is buried under, or submerged into, a collection of programmable matter modules. Upon receiving a start signal, the modules mechanically bond with their neighbors to encase the original object in a single regular lattice. (Currently, the system is incapable of handling grain

K. Gilpin and D. Rus are with the Computer Science and Artificial Intelligence Lab, MIT, Cambridge, MA 02139 [kwgilpin|rus]@csail.mit.edu

boundaries within the lattice, but others [2] have begun addressing this problem, and we hope to incorporate their results in future iterations of our algorithm.) Once solidified around the passive object, the modules execute our duplication algorithm that senses the shape of the 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 by selectively unlatching the unnecessary modules. 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 has several important properties. First, it is completely distributed and runs on a set of identical modules. Second, because a complete description of the goal shape is never known by any module, the distributed shape formation algorithm scales favorably as we form increasingly large shapes. Ignoring the $O(\log n)$ scaling associated with storing large numbers, the duplication algorithm presented here only requires constant storage per module. Furthermore, the total number of neighbor-to-neighbor messages exchanged in the process of duplication scales as $O(n^2)$. Third, the algorithm accounts for both broken communication links and missing modules making it robust to imperfections in the regular lattice of modules surrounding the passive shape. Finally, the algorithm transitions seamlessly from simulation to hardware. It is implemented in `C` with code that compiles and runs in simulation or on hardware without modification.

A. Related Work

Our work builds on a body of research addressing programmable matter and modular self-reconfiguring systems [3], [4], [5], [6]. Researchers have characterized modular robots as either chain or lattice systems [7]. Chain-based systems [8], [9], [10] are built around tree-like topologies. In contrast, lattice-based systems [11], [12], [13] have been the basis of programmable matter as they tend to use homogeneous, symmetric modules arranged in a regular 2- or 3-D grid pattern in which a module can bond and communicate with neighbors in every direction.

Klavins et al. have developed hardware and algorithms for a triangular self-assembling system that operates on an air table [14]. Griffith et al. [15] have shown that self-assembling systems can self-replicate in a distributed fashion. Lipson's group has also been instrumental in developing stochastically-driven self-assembling programmable matter systems [16], [17]. Christensen et al. [18] have developed a language that can direct robotic self-assembly. Despite the promise of self-assembling systems, they are ill-suited for the distributed duplication algorithm that we present here. Instead of a system that constructs objects by adding modules piece-by-piece, we need a system that starts from an initial collection of close-packed modules that (1) envelop that shape to be duplicated and (2) form a block of raw material out of which the duplicate shape can be fabricated. One such system is the Catoms [5]. The general concept of a Catom is a spherical or cylindrical robot whose surface is covered in actuators that allow the Catom to bond with and move relative to another Catom. Goldstein et al. present a clever shape formation algorithm [19] that operates on a densely packed arrangement of Catoms by creating and absorbing bubbles, or movable voids, within the interior of the structure. The advantage is that a description of the shape to be formed only needs to be shared with the surface modules.

In other work, Pillai et al. present an algorithm [6], for the Catoms system that enables duplication of a passive shape. The goal of this algorithm is virtually identical to our own. Their algorithm can successfully capture and replicate the shape of a passive object in simulation. Unlike our algorithm, the Pillai algorithm is centralized and relies on an external computer with significant processing capability to manage the duplication process. It also results in a high communication overhead as all shape data streams out of and into the particle ensemble from a single point. The algorithm is also incapable of operating on lattices that contain voids. In more recent work, Funiak et al. [2] have developed an interesting algorithm capable of distributed localization in a completely irregular packing of smart particles. The distributed duplication algorithm that we present here is generic and can be implemented on the Catoms, the Digital Clay modules [20], or the Miche [21] and Robot Pebble [1] programmable matter systems. While many programmable matter systems do not yet exhibit enough inter-module torque and force to withstand utilization as tools, there is hope that they eventually will [22], [23], [1].

The remainder of this paper is organized as follows. Section II provides an explanation of the basic message routing algorithm that is used as the communication backbone for the duplication algorithms presented in Section III. Section IV analyzes the robustness of the duplication algorithm. Section V gives an overview of how to form multiple duplicates or duplicates that are a magnified version of the original object. Section VI presents experimental results from simulation and hardware which demonstrate the distributed duplication algorithm working correctly and robustly. Finally, we present avenues for future work in Section VII.

II. Routing Algorithm

Distributed shape duplication requires a reliable way of sending messages from one programmable matter module in the system to any other. Because the collection of modules may be non-convex, have missing communication links, and contain concave voids, a simple gradient descent routing algorithm is not sufficient. The limited processing power and storage available to each module 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. We choose to use the traditional bug algorithm [24] 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.

In particular, we use the Bug2 algorithm. This algorithm is provably correct [24] 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 (message) 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 bug algorithm is not without its drawbacks. Primarily, the basic bug algorithm does not function in 3D. See Section VII for a discussion of how we are extending our algorithm to handle 3D originals. The experiments in this paper only consider 2D shapes, but the theory is extensible to 3D shapes.

III. Duplication Algorithms

The distributed duplication algorithm is a multiple 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 2 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 then 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.

A. Encapsulation and Localization

The shape duplication process begins with the user surrounding 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 Robot 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 a single module that begins 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.

B. Shape Sensing / Leader Election

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 obstacle to be duplicated. After a module is localized by an incoming *position* message, it detects which of its neighbors are present by assuming that unresponsive neighbors are absent. It 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), a *sense* message to each of its missing neighbors. Because the destination coordinates are occupied by the obstacle being duplicated, the *sense* message will never be delivered to its destination, but this is the intent. Instead, the *sense* 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.

In the process of traversing the obstacle, the *sense* message is modified by each module through which it passes so that by the time the message returns to its sender, it holds the obstacle's area, perimeter, and the extents of the obstacle's bounding box. Figure 3 shows this process in action. The perimeter computed by the *sense* 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 *sense* message determines the obstacle's bounding box by logging the minimum and maximum x- and y-coordinates through which it travels.

While Figure 3 only shows a single module's *sense* 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 *sense* messages from modules with lower unique IDs than their own. Because there is a single highest ID, all *sense* messages except one will be discarded before they return to their sender. The module whose *sense* message returns is the de facto leader. Figure 3 also omits the fact that all modules on the external perimeter of the entire configuration of modules generate *sense* messages. These messages are routed in an identical manner, 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.

C. Border Notification

The border notification phase duplicates the border of the original shape in the nearby modules and involves three types of messages. *Duplication* messages inform each module on the border of the original shape of their special status. *Border* messages are sent by modules on the border of the original shape and inform each module that is on the border of what will become the duplicate shape of their status. *Confirmation* messages, in turn, are sent by recipients of *border* messages

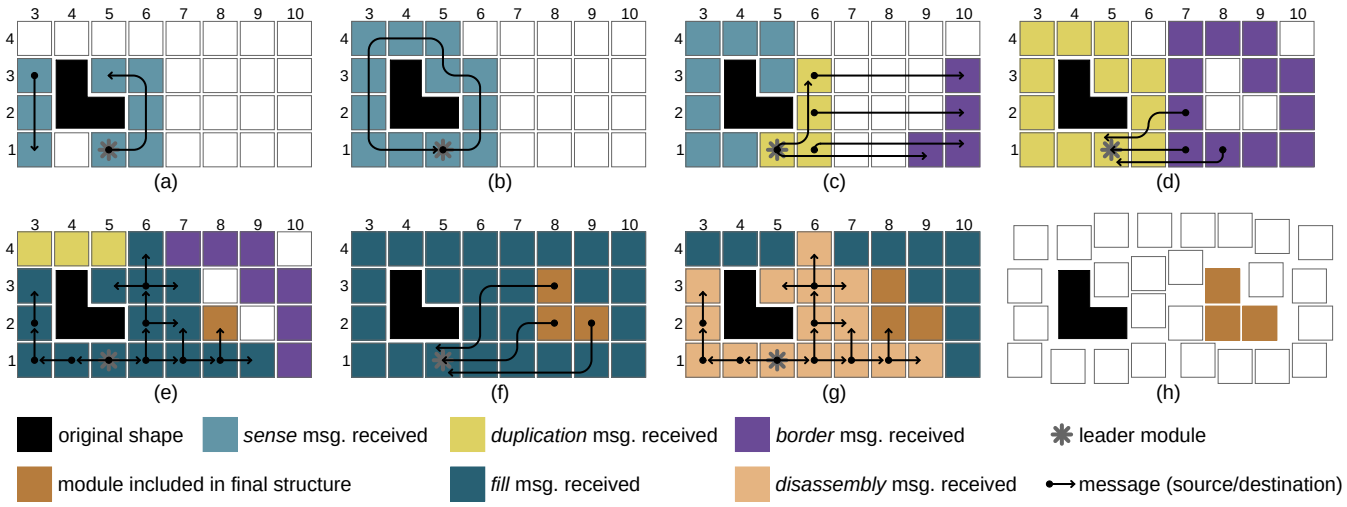

Fig. 2. After localization, the distributed duplication algorithm begins in (a) by routing a *sense* message around the border of the obstacle. As shown in (b), the message sent by the module with the highest unique ID will eventually return to its sender, prompting that module to route a *duplication* message around the border of the obstacle (c). Upon receiving a *duplication* message, a module sends *border* message to its conjugate that will become the border of the duplicate object. After all duplicate border modules have sent *confirmation* messages back to the leader (d), the leader broadcasts a *fill* message (e) informing modules contained by the new border that they are part of the duplicate shape and causing them to send *confirmation* messages back to the leader, (f). Upon receiving all confirmation messages, the leader broadcasts a *disassemble* 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 the paper as well.

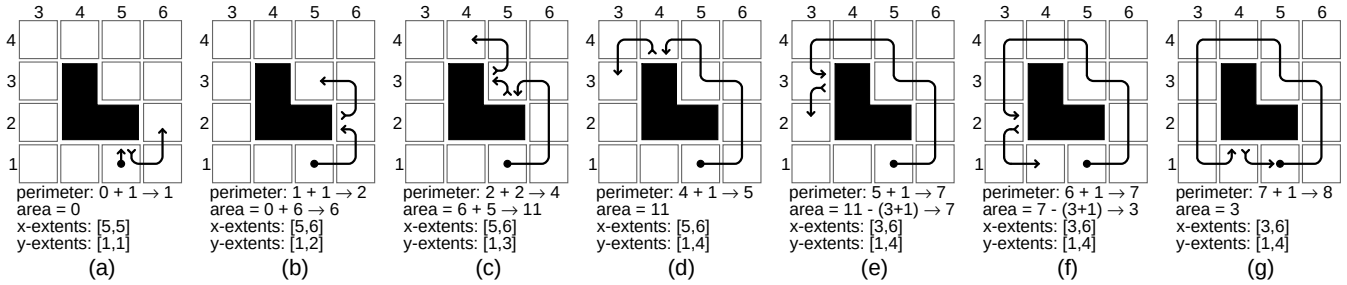

Fig. 3. As the modules attempt to use the bug algorithm to route a *sense* messages from its source at (5,1) to its non-existent destination at (5,2), they update the perimeter, area, and bounding box fields carried within the message. The perimeter is incremented as the result of every "collision" with the obstacle, and area is accumulated row-by-row. When the message returns to its sender, these parameters accurately describe the obstacle.

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* message to its missing neighbor whose position is instead occupied by the obstacle to be duplicated. Like the *sense* message that was previously sent, the *duplication* 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 *duplication* message passes through the modules on the border of the original shape, each module attempts to route a *border* message to the module identified by the direction vector added to the sender's coordinates. After stimulating each module on the border of the original shape to send a *border* message, the *duplication* message eventually returns to the leader where it is discarded.

When the *border* messages reach their destinations, these modules become the border of the duplicate shape. (While Figure 2 does not show it, modules on the border of the original may simultaneously serve as the border of the duplicate.) Because the *border* messages also carry the coordinates of the leader module, each *border* recipient sends a border *confirmation* message back to the leader carrying the length of perimeter of the duplicate shape on which the module borders. By comparing cumulative length of all received *confirmation* 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.

D. Shape Fill

The shape fill phase notifies all modules inside the border of the duplicate shape that they form the duplicate object and should remain 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* message that floods the entire network of mod-

ules. 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 the duplicate shape. As a result, only modules surrounded by the duplicate border receive a *fill* 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) *confirmation* message to the leader. By comparing the number of received area *confirmation* 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* message.

E. Self-Disassembly

After the leader can verify that each module in the duplicate shape knows that it should not disassemble, the leader broadcasts a *disassembly* message to the entire structure. This message floods the network and the unincluded modules begin disassembling from their neighbors in an orderly fashion [25] until only the duplicate object remains.

F. Storage and Communication

The distributed shape duplication algorithm requires only a constant amount of storage per module that 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* 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* message to each neighbor (to minimize the number of *fill* 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* messages instead of keeping a list of which modules have transmitted *confirmation* 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* messages. Each *sense* 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* 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* 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.

IV. 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 the system. This assumption is reasonable given the strength of the physical bonds [1]. While we do not have space to consider it here, it is straight-forward to prove that the robustness modifications operate correctly, always terminate, and do not modify the theoretical bounds on the storage and communication.

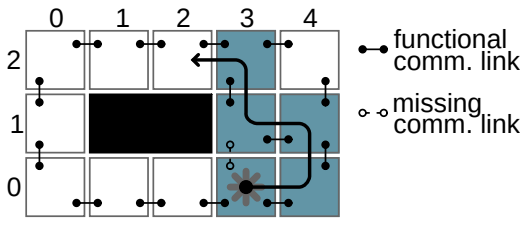

Fig. 4. The missing link between modules (3,0), and (3,1) will cause the module located at (3,0) to send a *sense* message to (3,1). Instead of discarding this message, and aborting the entire border sensing process, the module at (3,1), (even though it is the intended recipient), must continue routing 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 Bug2 algorithm will treat missing links just like another obstacles that must be avoided. The one scenario in which a missing communication link can affect the system is shown in Figure 4. In general, missing communication links are problematic when they border on the object to be duplicated because the *sense* messages sent by the two modules that share the missing link will actually reach their destinations, (unlike most *sense* messages which are destined for a location occupied by the obstacle being duplicated). Referring to Figure 4, the *sense* 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 *sense* 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 *sense* or *duplication* message reaches its destination. Instead, it will 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 5. First, when a module is missing from a location adjacent to the original object being duplicated, (such as at location $(5,4)$ in the Figure), the 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).

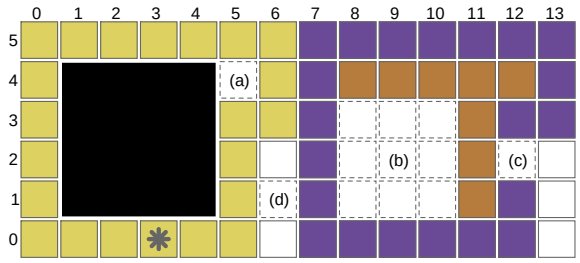

Fig. 5. The duplication algorithm is robust enough to handle modules missing from any potential location: (a) adjacent to the object being duplicated; (b) in the interior of the duplicate shape; (c) on the border of the duplicate shape; or (d) in any other position.

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 5), 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. 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 5. 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* message to the 9 missing modules. As the system discovers that each of these messages is undeliverable, it will attempt to route *disconfirm* messages to the missing modules' conjugate locations in the original obstacle. For example, if the module at $(7,2)$ in the Figure determines that a *fill* message destined for $(8,2)$ is undeliverable, $(7,2)$ will attempt to route a *disconfirm* message to $(1,2)$. Because location (1,2) is occupied by the passive obstacle being duplicated, this *disconfirm* message will never be delivered. As the system discovers that each *disconfirm* message is undeliverable, it sends an area *confirmation* 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 the module at $(5,2)$ determines that the *disconfirm* message destined for $(1,2)$ cannot be delivered, the module acts as a proxy for the module at $(8,2)$ and sends an area *confirmation* message to the leader at $(3,0)$. Additionally, $(5,2)$ sends *fill* 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) *fill* message. Without this last step, the leader would never receive a *confirmation* message from a module proxying for the missing module at $(9,2)$. While the details are beyond the scope of this paper, we use a combination of highest ID and distance to discard many *fill* messages so that we do not generate an excess of area *confirmation* messages that would confuse the leader.

Fourth, and finally, it is now easy to handle modules missing from the border of the duplicate shape such as the module missing from $(12,2)$ in Figure 5. During the Border Duplication phase, the *border* 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 *confirmation* 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 *fill* message destined for $(12,2)$ generates a *disconfirm* message that is sent to the missing module's conjugate, $(5,2)$, in Figure 5. In contrast to the interior case, this *disconfirm* message is actually delivered because location $(5,2)$ is 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 *confirmation* message to the sender.

V. 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* 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 *border* 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. For a concrete example, see Figure 6.

The process of magnifying the duplicate shape is illustrated by Figure 7. We append the magnification factor field, *M*, to each *border* message. In addition, the modules on the perimeter of the original shape modify the destination of the *border* message they each send so that the destination includes an additive factor that depends on the product of

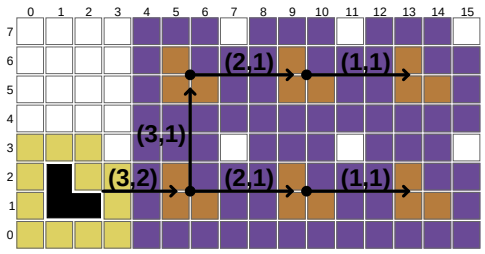

Fig. 6. When creating multiple copies of an original shape, we arrange the copies in an array whose dimensions (here 3-by-2) are appended to each original *border* message. As the *border* messages move through the structure, the remaining row and column counts are decremented as shown.

M with the module's relative location within the bounding box of the original shape. Each module that receives one of these primary *border* messages becomes a local leader of an *M*-by-*M* group of modules. As shown in Figure 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 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 *border* message.

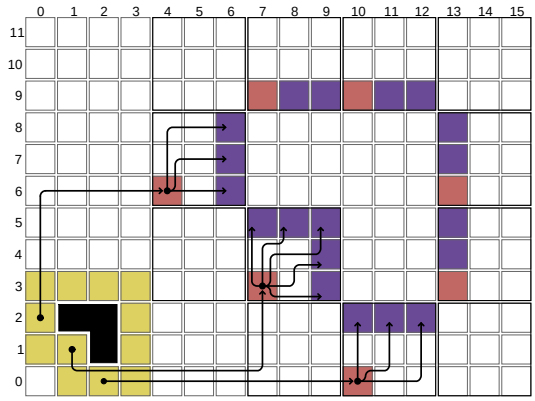

Fig. 7. When creating a magnified version of the original shape, a set of primary *border* messages are sent by the modules on the perimeter of the original shape to local leaders (in red) which may not lie on the border of the duplicate. These local leaders then send secondary *border* messages to the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

VI. Experimental Results

We performed a variety of simulated and hardware-based experiments, the results of which are shown in Table I. We had 20 Robot Pebble modules available to use for duplicating small shapes. (While we assembled these around the passive object by hand, we have previously shown that automated assembly is possible [26]). We used a custom-designed 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 I. The locations of the missing links and modules change with each trial. The *Disassembly Begun* column in the Table indicates in what percentage of trails the Self-Disassembly phase was started by the leader module, indicating that the leader module at least received all border and area *confirmation* 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.

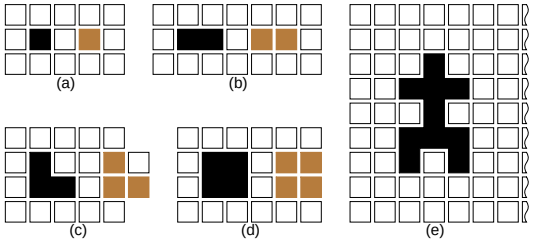

Fig. 8. We verified the duplication algorithm using a variety of shapes in both hardware, subfigures (a-d), and simulation, subfigure (e).

In hardware trails, there were four instances where the leader module did not initiate the Self-Disassembly phase. Additionally, in hardware, the Self-Disassembly phase only resulted in 90.0% of bonds being successfully broken. While these problems deserve further investigation, we do not believe they reflect on the core of the duplication algorithm's reliability. In simulation, despite 10% of the communication links broken and 5.0% of the modules removed, the algorithm performed flawlessly when creating a 1-to-1 duplicate, a magnified duplicate, or multiple duplicates.

In the hardware experiments, we believe most of the failures were due to unreliable inter-module communication links. Once a module communicates with a given neighbor, it assumes that link is valid until its neighbor explicitly disconnects from the system. If the link is accidentally broken, both modules will attempt to continue to use it, resending the same message indefinitely. As a result, the hardware system is fragile. With some messages forever lost to these broken links, it is not surprising that we see duplication failures in hardware. This problem is compounded by the fact that small variations in module size create internal stresses in the solidified structure. Forming or breaking one mechanical bond may generate or release stress in other bonds and cause the associated communication links to fail. In an attempt to minimize these internal stresses, we assembled the modules by hand, but ultimately we need to develop more robust algorithms that can handle dynamic link failures.

VII. Discussion

We have shown a distributed, robust algorithm for shape sensing and duplication in a lattice of interconnected smart particles. The algorithm is able to characterize the shape of a passive object submerged in a collection of particles and then use spare particles to form multiple replicas or a magnified replica. In the near term, we have several

TABLE I

WE PERFORMED 61 HARDWARE TRIALS WITH SIMPLE SHAPES AND 150 SIMULATIONS WITH COMPLEX SHAPES IN IMPERFECT LATTICES.

Shape	*Sim/ HW*	*Broken Links [%]*	*Missing Modules [%]*	*Mag. Factor*	*Array Size*	*No. Trials*	*Avg. Time [s]*	*Disassembly Begun [%]*	*Correct Bonds [%]*
Fig. 8(a)	HW	Unknown	0.0	1x	1x1	15	29.3	80.0	89.8
Fig. 8(b)						16	38.0	100.0	94.0
Fig. 8(c)						15	47.1	100.0	87.5
Fig. 8(d)						15	50.6	93.3	90.7
Fig. 5	Sim	5.0	0.0	1x	1x1	25	n/a	100.0	100.0
Fig. 6	Sim	10.0	5.0	1x	3x2	25	n/a	100.0	100.0
Fig. 7	Sim	10.0	5.0	3x	1x1	25	n/a	100.0	100.0
Fig. 8(e)	Sim	10.0	5.0	1x	1x1	25	n/a	100.0	100.0
		5.0	2.5	1x	2x2	25	n/a	100.0	100.0
		5.0	2.5	2x	1x1	25	n/a	100.0	100.0

extensions to pursue. We plan to supplement the system's functionality with the ability to mirror an object about a reference line and to invert an object. We also intend to develop a method for joining multiple objects together after the disassembly process. Additionally, we want the system optimally fit the duplicate object into the available collection of programmable matter modules.

Before programmable matter systems are practical, there are several larger problems that need to be addressed. The modules need to be miniaturized so that the resulting objects have acceptable resolution. Second, we need to implement distributed duplication in 3D. A 3D version of the algorithm will not be a simple extension of the 2D case because perimeter tracing is not efficient in 3D. We are currently investigating the decomposition of the 3D case into 2D subproblems. Additionally, we need to continue to drive down the number of messages the modules exchange. $O(n^2)$ total messages is a formidable communication burden in a system with a million modules. Finally, we need to adapt our algorithm to handle irregular lattices. With these improvements in place, we will be one big step closer to realizing robust programmable matter systems that can rapidly fabricate complex real-world objects that are naturally imbued with useful actuation, sensing, communication, and control capabilities.

ACKNOWLEDGMENTS

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

REFERENCES

[1] K. Gilpin, A. Knaian, and D. Rus, "Robot pebbles: One centimeter robotic modules for programmable matter through self-disassembly," in *IEEE ICRA*, May 2010, pp. 2485–2492.

[2] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, "Distributed localization of modular robot ensembles," *IJRR*, vol. 28, no. 8, pp. 946–961, 2009.

[3] B. J. MacLennan, "Universally programmable intelligent matter summary," in *IEEE NANO*, 2002, pp. 405–408.

[4] R. Nagpal, "Programmable self-assembly using biologically-inspired multiagent control," in *ACM Autonomous Agents and Multiagent Systems*, 2002, pp. 418–425.

[5] G. S. Copen and T. C. Mowry, "Claytronics: An instance of programmable matter," in *Wild and Crazy Ideas Session of ASPLOS*, Boston, MA, October 2004.

[6] P. Pillai, J. Campbell, G. Kedia, S. Moudgal, and K. Sheth, "A 3d fax machine based on claytronics," in *IEEE IROS*, 2006, pp. 4728–4735.

[7] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular self-reconfigurable robot systems: Challenges and opportunities for the future," *IEEE RAM*, vol. 14, no. 1, pp. 43–52, 2007.

[8] A. Castano and P. Will, "Mechanical design of a module for reconfigurable robots," in *IEEE IROS*, 2000, pp. 2203–2209.

[9] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji, "M-tran: Self-reconfigurable modular robotic system," *Trans. on Mechatronics*, vol. 7, no. 4, pp. 431–441, December 2002.

[10] B. Salemi, M. Moll, and W.-M. Shen, "Superbot: A deployable, multifunctional, and modular self-reconfigurable robotic system," in *IEEE IROS*, October 2006, pp. 3636–3641.

[11] G. S. Chirikjian, "Kinematics of a metamorphic robotic system," in *IEEE ICRA*, May 1994, pp. 449–455.

[12] C. Chiang and G. S. Chirikjian, "Modular robot motion planning using similarity metrics," *Autonomous Robots*, vol. 10, pp. 91–106, 2001.

[13] M. Koseki, K. Minami, and N. Inou, "Cellular robots forming a mechanical structure (evaluation of structural formation and hardware design of "chobie ii")," in *Distributed Autonomous Robotic Systems*, June 2004, pp. 131–140.

[14] N. Napp, S. Burden, and E. Klavins, "The statistical dynamics of programmed self-assembly," in *IEEE ICRA*, 2006, pp. 1469–1476.

[15] S. Griffith, D. Goldwater, and J. M. Jacobson, "Robotics: Self-replication from random parts," *Nature*, vol. 437, p. 636, Sept 28 2005.

[16] P. White, V. Zykov, J. Bongard, and H. Lipson, "Three dimensional stochastic reconfiguration of modular robots," in *Robotics: Science and Systems*, June 2005.

[17] M. T. Tolley, M. Kalontarov, J. Neubert, D. Erickson, and H. Lipson, "Stochastic modular robotic systems: A study of fluidic assembly strategies," *Trans. of Robotics*, vol. 26, no. 3, pp. 518–530, June 2010.

[18] A. Christensen, R. OGrady, and M. Dorigo, "Swarmorph-script: A language for arbitrary morphology generation in self-assembling robots," *Swarm Intelligence*, vol. 2, pp. 143–165, 2008.

[19] S. C. Goldstein, J. Campbell, and T. Mowry, "Programmable matter," *IEEE Computer*, vol. 38, no. 6, pp. 99–101, 2005.

[20] M. Yim and S. Homans, "Digital clay," WebSite. [Online]. Available: www2.parc.com/spl/projects/modrobots/lattice/digitalclay/index.html

[21] K. Gilpin, K. Kotay, D. Rus, and I. Vasilescu, "Miche: Modular shape formation by self-disassembly," *IJRR*, vol. 27, pp. 345–372, 2008.

[22] P. J. White, M. L. Posner, and M. Yim, "Strength analysis of miniature folded right angle tetrahedron chain programmable matter," in *ICRA*, 2010, pp. 2785–2790.

[23] M. E. Karagozler, S. C. Goldstein, and J. R. Reid, "Stress-driven mems assembly + electrostatic forces = 1mm diameter robot," in *IEEE IROS*, 2009, pp. 2763–2769.

[24] V. J. Lumelski and A. A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *Trans. on Automatic Control*, vol. 31, no. 11, pp. 1058–1063, 1986.

[25] K. Gilpin, K. Koyanagi, and D. Rus, "Making self-disassemblign objects with multiple components in the robot pebbles system," in *IEEE ICRA*, May 2011, pp. 3614–3621.

[26] K. Gilpin and D. Rus, "Modular robot systems: From self-assembly to self-disassembly," *IEEE Robotics and Automation Magazine*, vol. 17, no. 3, pp. 38–53, September 2010.