What's in the Bag: A Distributed Approach to 3D Shape Duplication with Modular Robots

Kyle W. Gilpin and Daniela Rus
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139
Email: {kwgilpin,rus}@csail.mit.edu

***Abstract*—Our goal is to develop an automated digital fabrication process that can make any object out of smart materials. In this paper, we present an algorithm for creating shapes by the process of *duplication*, using modules we have termed *smart sand*. The object to be duplicated is dipped into a bag of smart sand; the particles exchange messages to sense the object's shape; and then the particles selectively form mechanical bonds with their neighbors to form a duplicate of the original.**

Our algorithm is capable of duplicating convex and concave 3D objects in a completely distributed manner. It uses $O(1)$ storage space and $O(n)$ inter-module messages per module. We perform close to 500 experiments using a realistic simulator with over 1400 modules. These experiments confirm the functionality and messaging demands of our distributed duplication algorithm while demonstrating that the algorithm can be used to form interesting and useful shapes.

I. Introduction

In this paper we present a new distributed algorithm that enables the creation of complex 3D shapes from a collection of intelligent robotic particles. Our algorithm enables a collection these small *smart sand* grains to form arbitrary 3D shapes through a process of distributed duplication. The modules have on-board computation, nearest-neighbor communication, and latching capabilities. Given a "bag" of smart sand, we envision dipping a scaled replica of the object we wish to duplicate into the bag. (See Figure 1 for an example of duplicating a mug.) The intelligent modules surrounding the object sense and learn its shape. Then, using programmed communication and connections, they replicate the object 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.

The algorithm in this paper provides a solution to distributed duplication of arbitrary 3D objects (i.e. both convex and concave). The solution relies on local sensing of the boundary of the desired object and coordinated inference and planning to create a solid replica. Modules inside the replica form mechanical bonds with each other, and modules outside break all their bonds to surrounding neighbors. 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.

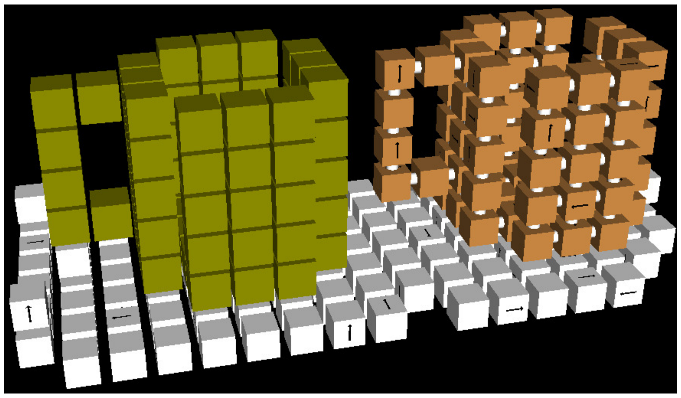


Fig. 1. The distributed duplication algorithm is capable of duplicating arbitrary 3D objects like the coffee mug (left) using a collection of intelligent modules modules. The modules envelop and sense the shape of the original object before forming a duplicate (right) from spare modules. Any extra modules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

We have implemented and evaluated this algorithm in simulations with thousands of modules. The code is written in C and structured such that by retargeting the lowest-level inter-module message passing and bonding functions, it could be deployed on any modular system whose unit modules can (1) form a regular cubic lattice around the object to be duplicated; (2) mechanically bond and communicate with their six nearest neighbors; (3) perform on-board computation; and (4) store unique identifiers (UIDs) which are assigned during fabrication.

II. Related Work

While this paper is focused on algorithmic developments which enable efficient, programmatic shape formation, modular robotics [21] intimately couples hardware and algorithms. The Claytronics project envisions cylindrical [11, 10], and eventually spherical modules, covered in magnetic or electrostatic actuators, that are capable of rolling relative to one another. Lipson et al. have developed a stochastic fluidic assembly system consisting of cubic modules whose assembly is controlled by pressure and suction [19, 17]. Klavins et al. have developed a set of triangular tiles that circulate on an air table and magnetically bond with their neighbors [1]. The Kilobot system [16] consists of 1000 autonomous robots that use stick-slip locomotion to form planar shapes. The 45cm Miche cubes use mechanically switchable permanent magnets to bond together in 3D shapes [4]. The RaChET system is a chain of modules which, when bonded with mechanical latches, can exert practical torques [20].

To complement the variety of hardware, many shape formation algorithms have also been developed. Researchers have attempted to optimize the path planning process when reconfiguring sets of connected modules [18, 2]. Others have focused on encoding a desired shape in a set of rules that is executed by each module [9]. Shen et al. have developed an approach that conveys a complete description of the desired shape to every module in the system [15]. While expensive in terms of memory, the algorithm allows large collections of mobile modules to form scale-invariant versions of the desired shape. Funiak et al. published a localization algorithm based on SLAM that is capable of localizing tens-of-thousands of irregularly packed modules [3]. Goldstein et al. envisioned an algorithm which conveys a description of the desired shape to all modules on exterior perimeter of a system [7]. Then, by injecting or absorbing holes, the border can be expanded or contracted until it matches the desired shape. Pillai et al. developed an algorithm which enables a collection of modules to act as a 3D fax machine [14]. In contrast to our work, their algorithm is centralized and performs almost all computation externally. Griffith et al. developed a duplication system that mimics DNA [8]. Despite these algorithmic advances, no solution exists if one wishes to form arbitrary 3D shapes, without the use of an external controller, while keeping the memory requirements of each module constant.

The remainder of this paper is organized as follows. In the next section, we explain the challenges of, and our approach to, distributed duplication. Section IV then demonstrates how we solve the duplication problem in 2D. Section V then extends these results to 3D. In Section VI, we present over 1000 experiments and analyze the performance of the algorithm. Finally, Section VII offers a brief discussion of our results.

III. Formulation of Distributed Duplication

We assume that the object to be duplicated is tightly surrounded by a regular cubic lattice of modules. The user sends one module a start signal, and all of the modules form mechanical bonds with their neighbors to encase the object in a rigid block of material. Once solidified, the system senses the geometry of the passive object in a distributed manner. For each location occupied by the obstacle, the algorithm identifies a *conjugate* module some distance away (in a specified offset direction) that will become part of the duplicate shape. Once all of these duplicate modules have been notified, they remain solidified when all other modules self-disassemble.

Sensing the shape of the object being duplicated in a distributed manner that scales favorably is challenging. Figure 2 outlines the essence of our solution, which senses and identifies the perimeter 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 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.

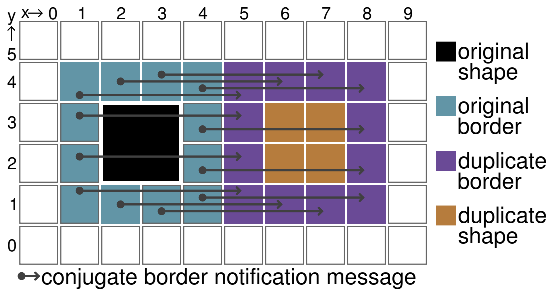


Fig. 2. The distributed duplication algorithm works by sensing the border of the shape to be duplicated. Once the border is identified, each module on the border notifies a conjugate duplicate border module that is offset a fixed distance in a given direction. With all the modules on the duplicate border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

Despite its relatively simple high-level description, there are many challenges when implementing the algorithm. 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. First, if all border modules do not agree on the offset distance by which to shift their conjugates, the conjugate border will not be a closed surface that mirrors the border of the original. Second, if the system starts the flood fill process before the surface surrounding the duplicate is complete, the fill messages will escape from holes in the surface and modules that should not be part of the duplicate will incorrectly remain bonded after self-disassembly. Third, if some module initiates the disassembly process before all modules inside the duplicate surface have received a fill message, some modules will disassemble instead of remaining bonded as part of the duplicate.

We developed a 2D solution [5] which addresses these challenges, but the extension to 3D is not simple. The primary additional challenge that the 3D algorithm must overcome is the fact that there is no efficient, distributed way to identify the perimeter of the passive shape. In particular, we need a distributed, message-based sensing algorithm that can completely envelop the 3D shape. Such an algorithm needs to identify all modules on the perimeter of the passive shape and inform those modules where to create the duplicate's perimeter.

To accomplish 3D duplication, we decompose the duplication process into 2D sub-problems using a layered approach. The initial block of material is cut into individual planes, and duplication proceeds semi-independently in each plane. In each plane, the obstacle perimeter identification problem uses the bug algorithm [13]. Any module on the perimeter of the obstacle (as determined by a missing neighbor), attempts to route a message to the unoccupied lattice location. In its futile attempt to reach its destination, the message circumnavigates the entire obstacle before returning to its sender.

The 3D duplication algorithm must synchronize all these planar processes. Concavities in the object to be duplicated need careful processing because they can create planes that have two or more disjoint groups of modules. To route messages from one group to the other, we use recent developments in 3D geographic routing [12].

IV. 2D Duplication

To understand the 3D duplication algorithm, one must understand 2D duplication. For more detail see [5]. The 2D algorithm, (see Figure 3), is a five-step process:

1) Encapsulation and Localization–the modules solidify around the original object and establish a coordinate system.

2) Shape Sensing / Leader Election–each module bordering on the original object transmits a `SENse` message, (which includes the module's hard-coded UID), that circumnavigates the object using the bug algorithm. As modules forward these messages to their neighbors, they discard messages with lower UIDs than their own. As a result, only a single `SENse` message completes its circumnavigation and returns to its sender. This module is elected the *obstacle leader*. The returning `SENse` message has learned the perimeter, area, and bounding box of the obstacle.

3) Border Notification–The obstacle leader sends another message around the border of the original shape which follows the same path taken by the `SENse` message. This `DUPlication` message prompts each module bordering on the original object to transmit a `BORder` message. These `BORder` messages are each addressed to a conjugate border module that is offset, in a user-specified direction, by an automatically calculated distance. The offset distance is determined by the size of obstacle's bounding box as learned during shape sensing, and this distance is carried in the `DUPlication` message. When each conjugate border module receives its `BORder` message, it sends a `CONfirmation` message back to the obstacle leader. The obstacle leader compares the number of received `CONfirmation` messages to the perimeter of the obstacle to determine when the border notification phase is complete.

4) Shape Fill–With a closed duplicate border established, the obstacle leader floods the system with a `FILl` message. This message has an *inside* bit that is initially cleared. As the `FILl` message propagates, the *inside* bit is flipped every time the message passes through the duplicate border. As a result, every module inside the duplicate border will receive a `FILl` message with the *inside* bit set and every module outside will receive the message with the bit cleared. Each module that receives a `FILl` message with the *inside* bit set sends a `CONfirmation` message to the obstacle leader, (whose coordinates are provided as part of the `FILl` message). The obstacle leader compares the number of received `CONfirmation` messages to the area of the obstacle to determine when the fill phase is complete.

5) Self-Disassembly–once all modules that comprise the duplicate shape have been notified of their status, the obstacle leader starts the disassembly process by flooding the system with a `DISassemble` message. Upon receiving this message all modules except those in the duplicate structure break their bonds with the neighbors to reveal the duplicate shape. □

The 2D duplication algorithm uses the bug algorithm [13] for all message routing. In particular, its ability to route `SENse` and `DUPlication` messages along the face of the obstacle is crucial to the algorithm's success. A `SENse` message routed with the bug algorithm determines the obstacle's perimeter by counting the number of times it collides with the obstacle (see Figure 5). It also determines the obstacle's area by integrating on a row-wise basis. While not shown in Figure 3, modules on the exterior border of the entire assembly also believe themselves to be bordering on a obstacle, so they also send `SENse` messages which will circumnavigate the exterior border of the assembly. When one of these messages has completed its circuit, the area it carries will be negative, so the system can differentiate the exterior and interior border.

V. 3D Duplication

As illustrated in Figure 4, the key to the complete 3D duplication algorithm is to virtually cut the initial block of modules into individual planes. As shown in the $z = 2$ plane, a single cut plane may contain multiple distinct groups of modules. We call each of these groups a *slice*.

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 *start 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 the start module.

2) Hull Tree Construction–For 3D routing, we employ an algorithm [12] based on convex hull trees. Each module stores a single convex hull that encompasses its position and the

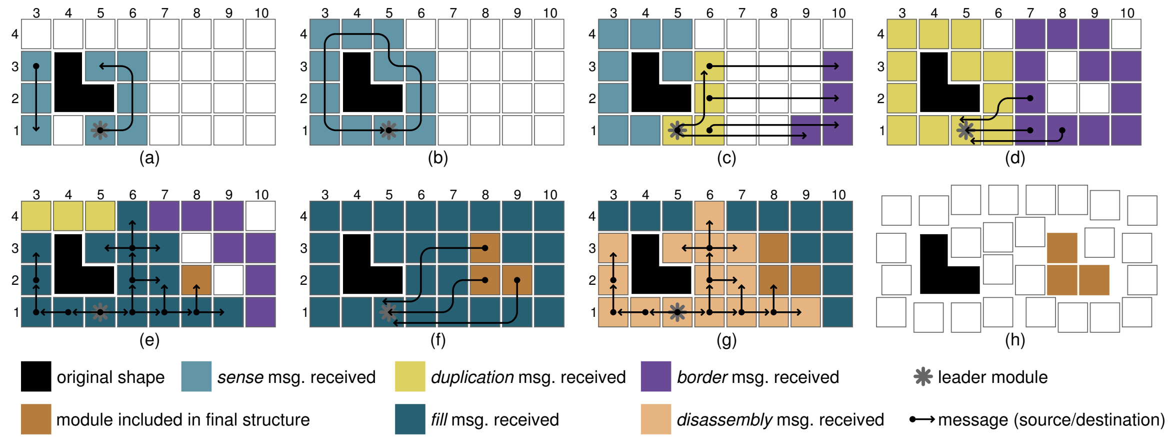

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
original shape
sense msg. received
duplication msg. received
border msg. received
leader module
module included in final structure
fill msg. received
disassembly msg. received
message (source/destination)

Fig. 3. 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 (marked with an asterisk) 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 obstacle 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) [5].

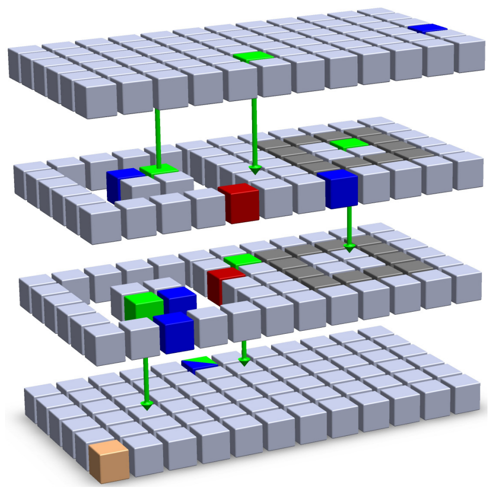


Fig. 4. Here a 12x6x4 block of material encasing a 4x4x2 tube (transparent) is sliced along the x-y plane. We reference all coordinates to the *start module* in the lower-front-left corner. Each distinct group of modules within a slice (there are two in the $z = 2$ slice) is termed as *slice* and has 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).

positions of all its descendant modules in the tree. During this step, the tree is built from the start module, which becomes the root of the tree, down to the leaves. Then, starting at the leaves, the convex hulls are constructed and propagated upward back to the start module, whose convex hull holds the positions of all modules in the system.

The 3D routing algorithm routes messages greedily, moving each message directly towards its destination whenever possible. When blocked by an obstacle, the message switches to traversing the convex hull tree. In particular, a message only descends into a node if that node's convex hull contains the message's destination. If the message exhausts all of the convex hulls associated with a module's children, the message then moves up the tree to the module's parent. We simplify the computation and storage requirements for the algorithm described in [12] by distilling each convex hull into a simple rectangular bounding box. Despite some theoretical loss in performance, this simplification the algorithm works well.

3) Shape Sensing–Within each slice, shape sensing operates nearly identically in 3D as it does in 2D by using the bug algorithm to route SENse messages around any apparent border. 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 5. 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 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.

4) Roll Call–The new slice leader broadcasts its position to all modules in the slice. Each module then replies with a RoLl Call 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. The root of the tree is the slice containing the start module. During construction, each slice

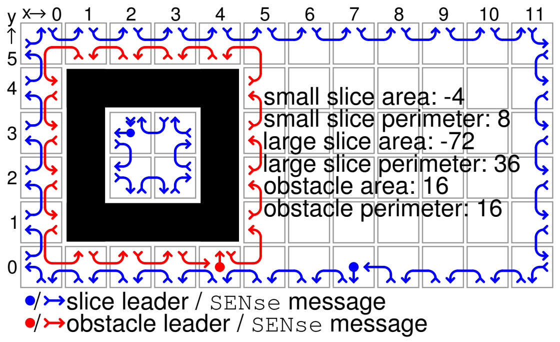
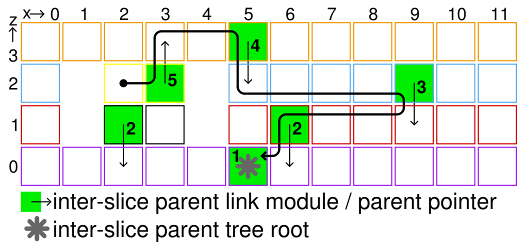


Fig. 5. Examining the $z = 2$ plane from Figure 4, we see that each of the two slices detects its exterior border by allowing a SENse message (in blue) from the border module with the highest UID (the slice leader) to trace the slice's exterior by virtually colliding with all of the missing modules. The larger outer slice contains an obstacle, so the obstacle leader also transmits a SENse 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.

knows that it has accounted for all possible children when each of its out-of-slice-plane neighbor modules reports that it has a parent. This is detailed below. The slice tree is be used to synchronize all slices after several of the following steps.



z
x
0 1 2 3 4 5 6 7 8 9 10 11
3 2 1 0
4
5
3
2
2
1
inter-slice parent link module / parent pointer
inter-slice parent tree root

Fig. 6. 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 4 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.

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. 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 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 4, the interior border of the tube must be duplicated along with the exterior border or else the duplicate object will become a solid block instead of another tube. 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 message to their conjugate modules that will become the border of the duplicate shape. The conjugate border modules reply with CONfirmation messages that are counted by either the obstacle leader or slice leader (depending on the type of border being duplicated–interior or exterior, respectively). When the obstacle and slice leaders have received CONfirmation messages from each duplicate border module for which they are responsible, they each send *secondary* CONfirmation messages to their respective inter-slice parent link modules. Once the inter-slice parent link module has received secondary CONfirmation messages for each obstacle, the slice as a whole, and any child slices, it forwards a secondary CONfirmation message to its parent slice. Eventually, secondary CONfirmation 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 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 V-C, the shape fill phase terminates just like the border notification phase. Each included module sends a CONfirmation message to the appropriate obstacle leader. Then the obstacle leader sends a secondary CONfirmation message to its inter-slice parent link module. The inter-slice parent link module waits for this and secondary CONfirmation messages from all child slices before propagating the secondary CONfirmation up the slice tree to the root.

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

A. Synchronization

To enable synchronization before the border notification, shape fill, and disassembly steps, the 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. Figure 6 shows an example. (Note: this 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 4 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 its slice. As a result, all modules in the slice learn the location of, what we term, their *inter-slice parent link module*. Once a module knows the location of its inter-slice parent link module, it can service incoming requests from neighboring slices looking for a parent in the slice tree.

In the neighboring slices that are not yet incorporated in the slice 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), responds. The module in the unincorporated slice to which it responds becomes that slice's inter-slice parent link module (after checking with its slice's leader module to ensure that no other module has already been appointed as the 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 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.

B. Exterior Face Identification

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 tube, the concave, or interior part of the tube'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 initial block of material. Figure 5 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 start module initiates the exterior face identification process because it was told, (as part of the start command), which of its faces was an exterior face. The start module uses the bug algorithm in an attempt to route two `EXTerior` messages in the direction of the specified exterior face. The first `EXTerior` 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. These `EXTerior` messages will circumnavigate the exterior border. As they pass through the modules on their routes, they notify those modules that they too are on the exterior of the whole block. Additionally, they prompt those modules to emit their own `EXTerior` 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 `EXTerior` message completes its circuit around an exterior slice, it sends an `CONfirmation` 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.

C. 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 5. During the shape fill step, the obstacle leader in the outer slice, expects to receive sixteen `CONfirmation` 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 7, is to send a fake `CONfirmation` message from a conjugate border module corresponding to the leader of the interior slice.

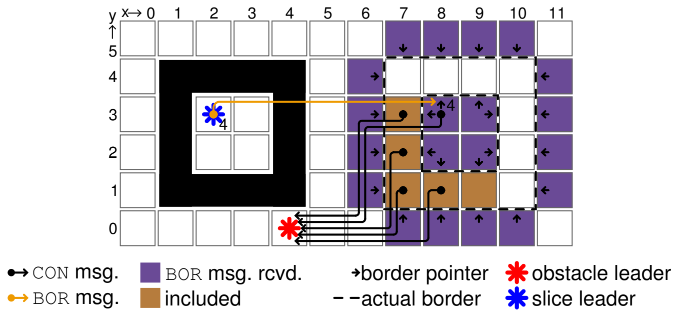


Fig. 7. The obstacle leader of the exterior slice, expects to receive sixteen `CONfirmation` messages after the shape fill process beings. Although the obstacle leader does not know it, the obstacle is hollow. To ensure that the obstacle leader still receives all `CONfirmation` messages, the slice leader of the inner slice sends a special `BORder` 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 `BORder` message to its conjugate, includes its slice's area. This is shown in Figure 7. Second, during the shape fill process, the slice leader's conjugate border module sends a `CONfirmation` message to the outer slice's obstacle leader even though it is not part of the duplicate shape. This `CONfirmation` 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.

D. Storage and Message Scaling

The algorithm requires only $O(1)$ 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. (Note: we ignore the $O(\log n)$ scaling of the number of bits required to store variables whose sizes are proportional to n.) 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. When collecting CONfirmation messages, modules do not track the origins of the messages, only the total number received. The convex hull tree only requires a constant amount of storage per module because each module needs to store the rectangular hulls of at most six neighbors. Likewise, messages only move up the slice tree, so each module only needs to know the location of its inter-slice parent module. All other routing information is constant in size and stored in the messages themselves.

The number of messages exchanged scales as $O(n^2)$ total or $O(n)$ per module, and it 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 EXTerior 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 SENse message and messages 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.

VI. Experiments

Using a custom simulator [6], we performed over 450 experiments duplicating rods, cubes, square tubes, the mug shown in Figure 1, a 103-module hammer, and 128-module wrench. The results are listed in Table I. The overall success rate was 100%. A video of the system in action is linked from the Supplementary Material section below.

TABLE I
Experiments show the 3D duplication algorithm working correctly in a variety of test cases.

Original Shape	*Encasing Shape*	*# Trials*	*# Successes*	*Avg. Msgs./ Trial*
2x1x1 Rod	7x3x3 Block	25	25	5409
3x1x1 Rod	9x3x3 Block	25	25	7413
4x1x1 Rod	11x3x3 Block	25	25	9788
5x1x1 Rod	13x3x3 Block	25	25	12170
6x1x1 Rod	15x3x3 Block	25	25	14757
7x1x1 Rod	17x3x3 Block	25	25	18487
1x1x1 Cube	5x3x3 Block	25	25	3602
2x2x2 Cube	7x4x4 Block	25	25	10199
3x3x3 Cube	9x5x5 Block	25	25	21768
4x4x4 Cube	11x6x6 Block	25	25	41725
5x5x5 Cube	13x7x7 Block	25	25	71698
6x6x6 Cube	15x8x8 Block	25	25	112410
7x7x7 Cube	17x9x9 Block	25	25	182720
4x4x4 Tube	11x6x6 Block	25	25	44381
5x5x5 Tube	13x7x7 Block	25	25	77990
6x6x6 Tube	15x8x8 Block	25	25	131670
7x7x7 Tube	17x9x9 Block	25	25	230320
Figure 1 Mug	17x7x7 Block	25	25	115020
Hammer	22x5x14 Block	5	5	290830
Wrench	20x4x18 Block	5	5	249098

Figure 8 shows a number of different statistics collected as we duplicated cubes with side lengths ranging from 1—7. In particular, Figure 8(a) plots the total number of inter-module messages exchanged 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. As expected, the total number of messages scales quadratically $(0.266n^2 + 78.037n)$ with the number of active modules in the system. Even though the n^2 term will dominate past a few hundred modules, we expect the number of messages *per module* to scale as $O(n)$. Figure 8(b) illustrates this: both the average number of messages per module and the maximum number of messages exchanged by any given module scale linearly. It is worth noting that the average number of messages per module is significantly less than the maximum as the size of the cube being duplicated grows. This relationship is better illustrated by Figure 8(c) which shows a histogram of the number of messages exchanged by all modules in the system as it is used to duplicate cubes. When similar plots are generated for the rods and tubes that we also duplicated, we see similar behavior to that in Figure 8.

While theory predicts that the $O(n)$ scaling shown in Figure 8(b) to continue as the number of active modules grows indefinitely, simulations containing more than few thousand modules are difficult. Each module is simulated as an independent process, and the modules use UDP packets to communicate with their neighbors. Several thousand concurrent processes begin to tax all but the most powerful PCs. Additionally, given that each module uses a unique UDP port to communicate with each neighbor, simulations with a few thousand modules quickly exhaust the available supply of port numbers. Neither of these issues will be a concern when running the algorithm on dedicated robotic hardware.

VII. Discussion

We have demonstrated a distributed algorithm capable of autonomously duplicating arbitrary 3D shapes in a modular robot system. The algorithm and implementation are generic and lend themselves to easy hardware implementation. Additionally, we have shown the algorithm running correctly in hundreds of test cases with some experiments using over 1200 simulated modules. The total memory required in each module is fixed while the total number of messages exchanged scales as $O(n^2)$. We hope to reduce this bound so that the number of messages exchanged *per module* is sub-linear.

Our algorithm opens several important problems. It is sensitive to any dropped messages. It is not guaranteed to operate if there are modules missing from the lattice. It will also not work in completely irregular lattices where the modules are randomly arranged and may have more than six neighbors. In the future, we hope to develop extensions that solve both these challenges. Additionally, we hope to eliminate the need for the user to specify in which cardinal direction, with respect to the original object, the duplicate should be placed. Our work raises the question of how to create scaled duplicates or multiple copies of a single original. Finally, if the object to be duplicated cannot be fit into the initial block of material, or if a scaled duplicate is too big to be created in one pass, we need to find a way to break the object into subcomponents that can be produced independently and then joined. This paper is

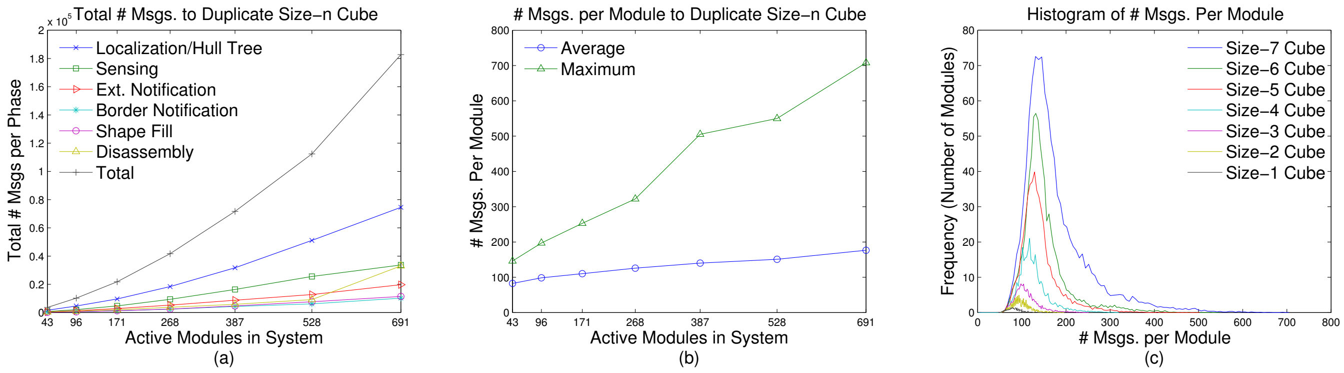


Fig. 8. When duplicating cubes with edges lengths 1—7, the *total* number of messages exchanged by all modules scales quadratically (a). The number of messages *per module* scale linearly (b). Furthermore, only a few modules exchange a relatively high number of messages, the numbers of messages exchanged by most modules are clustered near the average (c). In both subfigures (a) and (b), the 7 x-axis tick marks correspond to cubes with edge lengths 1—7.

a first step, and with additional research, we hope to make distributed duplication a practical reality.

Acknowledgments

Many thanks to our excellent reviewers. This work is supported by the US Army Research Office under grant number W911NF-08-1-0228, the NSF through Grants 0735953 and 1138967, and the NDSEG fellowship program.

Supplementary Material

http://www.youtube.com/watch?v=XmtXIHCRN_c

References

[1] 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 *IROS*, pages 3684–3691, August 2005.

[2] C. Chiang and G. S. Chirikjian. Modular Robot Motion Planning Using Similarity Metrics. *Autonomous Robots*, 10:91–106, 2001.

[3] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein. Distributed Localization of Modular Robot Ensembles. *IJRR*, 28(8):946–961, 2009.

[4] K. Gilpin, K. Kotay, D. Rus, and I. Vasilescu. Miche: Modular Shape Formation by Self-Disassembly. *IJRR*, 27:345–372, 2008.

[5] Kyle Gilpin and Daniela Rus. A distributed algorithm for 2d shape formation with smart pebble robots. In *ICRA*, page In Press, 2012.

[6] Kyle W. Gilpin. *Shape Formation by Self-Disassembly in Programmable Matter Systems*. PhD thesis, MIT, 2012.

[7] S. C. Goldstein, J. Campbell, and T. Mowry. Programmable Matter. *IEEE Computer*, 38(6):99–101, 2005.

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

[9] 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.

[10] M. E. Karagozler, S. C. Goldstein, and J. R. Reid. Stress-driven MEMS Assembly + Electrostatic Forces = 1mm Diameter Robot. In *IROS*, pages 2763–2769, 2009.

[11] B. T. Kirby, B. Aksak, J. D. Campbell, J. F. Hoburg, T. C. Mowry, P. Pillai, and S. C. Goldstein. A Modular Robotic System Using Magnetic Force Effectors. In *IROS*, pages 2787–2793, 2007.

[12] B. Leong, B. Liskov, and R. Morris. Geographic Routing without Planarization. In *Networked Systems Design and Implementation*, 2006.

[13] V. J. Lumelski and A. A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *Trans. on Automatic Control*, 31(11): 1058–1063, 1986.

[14] P. Pillai, J. Campbell, G. Kedia, S. Moudgal, and K. Sheth. A 3D Fax Machine based on Claytronics. In *IROS*, pages 4728–4735, 2006.

[15] Michael Rubenstein and Wei-Min Shen. Automatic Scalable Size Selection for the Shape of a Distributed Robotic Collective. In *IROS*, pages 508–513, Oct 2010.

[16] Mike Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *ICRA*, page In Press, May 2012.

[17] M. T. Tolley, M. Kalontarov, J. Neubert, D. Erickson, and H. Lipson. Stochastic Modular Robotic Systems: A Study of Fluidic Assembly Strategies. *Trans. on Robotics*, 26(3):518–530, June 2010.

[18] J. Walter, E. Tsai, and N. Amato. Algorithms for Fast Concurrent Reconfiguration of Hexagonal Metamorphic Robots. *Trans. on Robotics*, 21(4):621–631, 2005.

[19] P. White, V. Zykov, J. Bongard, and H. Lipson. Three Dimensional Stochastic Reconfiguration of Modular Robots. In *RSS*, pages 161–168, June 2005.

[20] P. J. White, M. L. Posner, and M. Yim. Strength Analysis of Miniature Folded Right Angle Tetrahedron Chain Programmable Matter. In *ICRA*, pages 2785–2790, 2010.

[21] M. Yim, W.M. 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*, 14(1):43–52, 2007.