The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Scalable Self-Assembly and Self-Repair In A Collective Of Robots

Michael Rubenstein, Wei-Min Shen

Abstract—**A collective of robots can together complete a task that is beyond the capabilities of any of its individual robots. One property of a robotic collective that allows it to complete such a task is the shape of the collective.**

In this paper, we present a distributed control method, called DASH, to enable a collective of robots to robustly and consistently form and maintain a pre-defined shape. This control method allows the shape that is formed to be at a scale proportional to the number of robots in the collective. If this collective shape is damaged through the un-controlled movement, removal, or addition of some members of the collective, the existing members will recover the desired shape, proportional to the new number of robots in the collective.

We also analyze this control method in terms of class of acceptable shapes and discuss the convergence to the desired shape.

I. INTRODUCTION

In this paper we present a solution to the general problem of controlling a collective of distributed robots (sometimes referred to as a swarm, group, or ensemble) so that they can robustly and consistently form and maintain a pre-defined shape. By forming a specific shape, a robot collective can complete a goal that cannot be completed by any of the individual robots. For example, in a collective of reconfigurable robot modules, the robots can form the collective shape of a wheel, which will allow them to travel faster and more efficiently, when compared to an individual reconfigurable robot module [1]. Another example is shown in [2], when a single SWARM-BOT is confronted with an obstacle of rough terrain that it cannot cross. It then joins together with other SWARM-BOTs to form a bridge shape, allowing them to traverse the rough terrain as a group.

Due to the distributed nature of the robot collective, the need for scalability, and the desired robustness against single points of failure, the control method should be decentralized. Most robot collective control methods are decentralized; however, some require an initial unique seed to start the shape formation [3,4]. This seed is not an unreasonable requirement, however it does force a centralized decision to be made, reducing robustness against single points of failure.

Along with being distributed, it is also important for the control method to be capable of forming as many different types of shapes as possible. Some methods have a limited class of shapes that they are capable of forming. For example in [5], the collective is only capable of forming polar shapes. In [3,6], the collective cannot form a shape that contains an empty internal volume.

It is also important for the collective to be resistant to damage to the shape. If the shape helps the collective complete a task, then damage to the shape may negatively affect the collective's ability to complete that task. Some control methods for forming a shape [6,7] do not have the ability to recover from most damage. Others [3], can recover from the addition or removal of robots to the collective, but not the un-controlled movement of robots from one location to another.

In the event that the number of robots in the collective changes through the addition or subtraction of robots, there are two options for the collective to adapt (self-heal). The first option, fixed scale self-healing, used in [4,8,9], is to keep the size of the shape the same, but change the density of robots. Due to an upper limit to this robot density (one can fit only so many robots in a fixed area), there is a maximum number of robots that can fit inside the collective shape. Another drawback to this first option is that, for many collective robotic systems, such as reconfigurable robots [10], the robots require a close physical connection to neighboring robots. This means that in general, the density of robots in the collective should remain the same, irrespective of the size of the collective. The second option for adapting to the change in the number of robots is to scalably self-heal, which adjusts the size of the shape proportional to the number of robots in the collective, keeping the robot density constant. This scalable self-healing is shown in nature [11], where a small invertebrate, the hydra, will reform its original shape after being cut in half, but at half the size. This has been recreated in some robotic collectives [3, 5].

During this process of self-healing as well as self-assembly, robots outside the shape must move to a location within the shape. At the same time, their movement is constrained with the added restriction of avoiding the locations of the other robots in the collective. Robots already inside the shape must take care not to stop in a location that would prevent robots outside the shape from entering. Some approaches [3,6] enact very careful communication exchanges between robots to guide moving robots past neighbors, avoiding disconnection, and locations where they can become trapped. Other approaches [5,8] use random or biased random movement, as well as collisions between robots, to move robots around neighbors, and prevent choosing locations where they can become trapped.

In the work presented here, we use a fully distributed control method, with no single point of failure, to enable a

Manuscript received March 1, 2009. Michael Rubenstein and Wei-Min Shen are with the Information Sciences Institute and Computer Science Department at the University of Southern California, Marina del Rey, CA 90292, USA. (website: www.isi.edu/robots phone: 310-448-8710; fax: 310-822-0751; e-mail: mrubenst@usc.edu , shen@isi.edu).

collective of robots to scalably self-assemble and self-repair a large class of shapes. This control method is resilient to the removal, addition, or un-controlled movement of robots in the collective, always returning the collective to the desired shape, proportional to the number of robots present. Section 2 will provide details of this control method; section 3 will discuss the convergence of the group to the desired shape; section 4 will provide experimental results of this control method running on a simulated robot collective.

II. Methods for Self-Assembly and Self-Healing

The following section describes DASH (Distributed Assembly and Self-Healing), a distributed method to control each robot in a robotic collective. This control method uses an identical controller that runs in each robot. Each robot controller also includes a full description of the desired collective shape. When DASH is run on each robot in the collective, the robots will self-assemble to form the desired shape (some examples shown in Fig. 1), and self-repair if the shape is damaged, as shown in Fig 5.

A. Assumptions

The robots are simple and homogeneous. Each robot is shaped like a simple 2D circle, with radius R_{robot}. It is capable of moving in its local x direction along a plane, as well as rotating about its center, perpendicular to the plane. A robot cannot share the same space as another robot, and is not capable of pushing any robots. All robots in the collective are identical and indistinguishable from each other in every way, even lacking a unique ID.

Communication between neighboring robots is possible. Each robot can communicate to any of its neighbors who are within a certain distance (R_{com}).

Robots have a consistent coordinate system. The collective has a shared coordinate system that is known by all robots. This enables each robot to precisely know its location in the coordinate system, in terms of (X,Y). This coordinate system could be given from a Global Positioning System (GPS) or developed from a local, distributed method such as trilateration [12], MDS-MAP [13], or robust quadrilaterals [14]. Using movement, the robot can also determine the angle between its x direction and the x direction of the coordinate system, as described in [5].

Robots know the total number of robots in the collective. This number, N_r, represents the total number of robots that are currently part of the collective. If some robots are added or removed, N_r will change accordingly.

B. Class of Shapes

While DASH is extendable to forming 3D shapes, in this paper, we will concentrate on 2D cases. DASH is theoretically capable of forming any connected shape, defined as a shape that for every location in that shape, there is at least one path that is fully contained within the shape to all other points in that shape. However, there are some practical limitations on the details of the shape, as the result of scaling the size of the shape. Namely, these limitations are that for a given shape, at a given scale, the minimum feature size must be greater than $2 \cdot R_{robot}$. This minimum feature size is found by fully decomposing the desired shape using largest possible overlapping circles. The diameter of these circles is the minimum feature size. From the minimum feature size, we can compute the minimum allowable height of these shapes. The minimum allowable height of the shape will be the height that causes the minimum feature size to be $2 \cdot R_{robot}$. An example of some possible shapes, shown as white, is shown in Fig. 1.

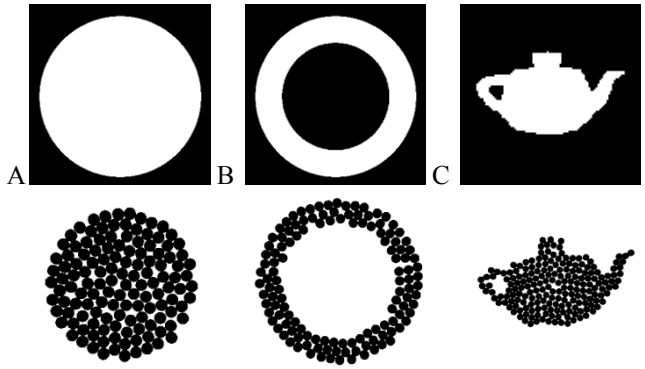


Fig. 1. Three example shapes (upper), and the approximation of those shapes (lower) by a simulated collective of robots (simulator described in detail in section 4).

C. DASH Controller Overview

The robot controller operates to achieve the following high level behavior. The controller first determines the desired scale of the shape, based on the number of robots. It then determines if the robot under its control is inside the desired shape. If the robot is inside the desired shape, the robot will move in a way that keeps it within the desired shape, but at the same time does its best to keep from blocking other robots from entering the shape. There are two possibilities if the robot is not in the desired shape. The first possibility is that it is on the outside of the desired shape. In this case, the robot will move along the perimeter until it can find a location to enter the shape. The other possibility is that the robot is not in the desired shape, but is surrounded by the desired shape, for example, the center black region in Fig 1.B. If this is the case, the robot will first try to enter the shape. If this is not possible because it is blocked by other robots already in the shape, then this robot is trapped. This trapped robot will enact a mechanism using robot-to-robot communication that moves some robots within the shape out of its way, allowing the trapped robot to move into the shape.

D. DASH Controller Details

The robot controller has two main sections. The first section, which is run once at robot startup, takes in the desired shape description, and outputs a gradient map (to be described later). The second controller section uses this gradient map, communication with other robots, N_r, and the robot's location in the shared coordinate system, to determine if the robot should move, and in what direction.

1) Processing the Pixel Map

The input to the first section of the DASH controller is a description of the desired collective shape. While other choices are possible, in our implementation, we chose to use a 100x100 pixel map to represent the desired shape. In this pixel map, a pixel is white to represent a location within the shape, and black to represent a location not in the shape, as shown in Fig. 1. We also add a constraint that every pixel on the outside border of this pixel map must be black. The reason for this will be described shortly.

First, the pixel map is segmented into groups of connected pixels with identical colors. Due to the shape constraints given in section II.B. of this paper, there will be only one segment that includes white pixels, which is called the *shape segment* (shown as the white segment in Fig. 2B). There will also be one segment called the *external segment,* (shown as the black segment in Fig. 2B). The external segment includes the pixel (0,0), which is the upper left most pixel in the pixel map. Due to the constraint that every pixel on the outside border of the pixel map must be black, the external segment will completely surround the shape segment. There are further possible segments, called *trapped segments*, if there are black pixels in the pixel map that are completely surrounded by the shape segment. Three examples of trapped segments are shown as vertical, horizontal and diagonal striped line segments in Fig. 2B.

In each segment, there is one "starting pixel", which is used later to generate the gradient map. For all segments except the external segment, the upper left most pixel in that segment is chosen as a starting pixel. For the external segment, the starting pixel is chosen to be the pixel in the external segment that is immediately to the left of the starting pixel for the shape segment.

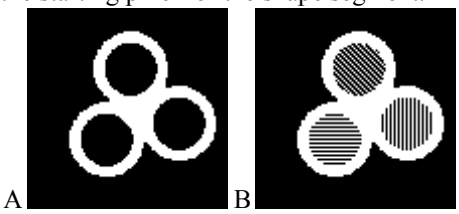
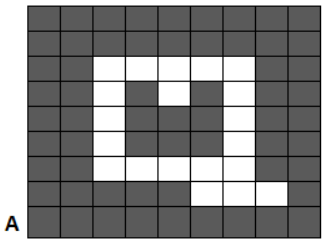


Fig.2. A) the pixel map of the desired shape. B) the pixel map segmented into 5 regions.

2) Creating the Gradient Map

The gradient map is a 100x100 array that has one integer entry for every pixel in the pixel map. Each location in the gradient map corresponds to the pixel in the same location of the pixel map. The values in the gradient map are set as follows. For the external and trapped segments in the pixel map, do the following. For each pixel in that segment, calculate the "Manhattan" distance of the shortest path between that pixel and the segment's starting pixel. This shortest path is constrained to be fully within the corresponding segment. The entry for this pixel in the gradient map is then set to -(path_length + 1). For the shape segment, a similar approach is used. For each pixel in the shape segment, calculate the "Manhattan" distance of the shortest path between that pixel and the shape segment's starting pixel. This shortest path is constrained to be fully within the shape segment. The entry for this pixel in the gradient map is then set to this path length.

When a gradient map is generated in this manner, any location in the gradient map with a negative value is located in a trapped or external segment. If the gradient value is positive or zero, then it is within the shape segment. When a location has a gradient map value of -1, then it is located at a starting pixel for an external or trapped segment. An example gradient map generated from a pixel map is shown in Fig. 3.



B

-4	-3	-4	-5	-6	-7	-8	-9	-10
-3	-2	-3	-4	-5	-6	-7	-8	-9
-2	-1	0	1	2	3	4	-9	-10
-3	-2	1	-1	3	-5	5	-10	-11
-4	-3	2	-2	-3	-4	6	-11	-12
-5	-4	3	-3	-4	-5	7	-12	-13
-6	-5	4	5	6	7	8	-13	-14
-7	-6	-7	-8	-9	8	9	10	-15
-8	-7	-8	-9	-10	-11	-12	-13	-14

Fig. 3. A) An example of a 9x9 pixel map. B) The gradient map generated from that pixel map.

The second part of the DASH controller is run continuously on each robot, and uses the gradient map built from the pixel map, communication with other robots, N_r, and the robot's location in the shared coordinate system, to determine if the robot should move, and if so, in what direction.

3) Finding and Using Scale_Factor

DASH first determines at what scale the shape should be formed, called the Scale_Factor. The controller uses an experimentally determined value, called the packing efficiency (P_e), as well as N_r, to find this scale. The P_e value represents the following: if the scale of the pixel map is made so that each pixel in the pixel map has the size $R_{robot} \times R_{robot}$, on average, P_e robots can fit in a pixel's worth of space. The Scale_Factor is determined by (1), which gives the size of a pixel, in terms of R_{robot}, where N_{pixels} is the number of white pixels in the pixel map.

$$Scale\_Factor = \sqrt{\frac{N_r}{(P_e \times N_{pixels})}} \quad (1)$$

Once Scale_Factor is known, the DASH controller can virtually overlay the gradient map at the appropriate scale onto the shared coordinate system. This will allow the DASH controller to determine, for any location in the shared coordinate system, what entry in the gradient map it corresponds to. When overlaying the gradient map on the shared coordinate system, we choose to place the center entry of the gradient map, (49,49), to correspond to the center of the shared coordinate system (0,0). To find the entry in the gradient map that corresponds to the robot's current location in the shared coordinate system, (2) is used, where x_{index} and y_{index} are the location in the gradient map, and x_{scs} and y_{scs} are the location of the robot in the shared coordinate system. The gradient map requires integer index values, so the floor function is used to change the real value within the parentheses of (2) to an integer value.

$$x_{index} = floor\left(\frac{x_{scs}}{Scale_{Factor}}\right) + 49$$
$$y_{index} = floor\left(\frac{y_{scs}}{Scale\_Factor}\right) + 49 \quad (2)$$

4) Gradient Maximization Movement

Once the controller has determined where the robot is located (x_{index}, y_{index}) in the gradient map, it uses that location, as well as its four neighboring grid locations in the gradient map (up, down, left, right), to determine how to move. Those four neighboring grid locations are used to determine the maximum gradient direction of the gradient map, around the robots current location in the gradient map. The angle of this gradient is then set as the desired direction of movement, θ_{move}, for the robot. The computation of this direction is shown in (3), where gm(x,y) returns the gradient map entry for location (x,y).

$$\theta_{move} = atan2(A, B) \qquad where, \quad (3)$$
$$A = gm\left(x_{index}, y_{index} + 1\right) - gm\left(x_{index}, y_{index} - 1\right)$$
$$B = gm(x_{index} + 1, y_{index}) - gm(x_{index} - 1, y_{index})$$

If the robot has an x_{index} or y_{index} that is not between 0 and 99, it will not have a valid entry in the gradient map. If that is the case, the robot will move in the direction towards (0,0) in the shared coordinate system, until it has a valid entry in the gradient map.

5) Trapped Robot Movement

If a robot finds itself at a location where the corresponding gradient map indicates it is at a starting pixel for an external or trapped segment, and its location in the gradient map corresponds to a pixel in the pixel map that is in a trapped segment, then the robot considers itself trapped, and initiates a procedure to become un-trapped. This un-trapping procedure uses communication between neighboring robots, supersedes the previous gradient following movement, and works as follows. First, the trapped robot generates a trapped robot message. This message contains the shared coordinate system location of the trapped robot $(x_{trapped}, y_{trapped})$. It is sent to all neighboring robots which have an x_{scs} less then $x_{trapped}$, and a y_{scs} that is within the range $(y_{trapped} - T_{width}) < y_{scs} < (y_{trapped} + T_{width})$, where T_{width} is a predefined constant. This message is further propagated since, every time a robot receives the trapped robot message, the receiving robot will send the message to all of its neighbors that have a x_{scs} less than that of the receiving robot, and a y_{scs} that is in the range $(y_{trapped} - T_{width}) < y_{scs} < (y_{trapped} + T_{width})$. When a robot receives a trapped robot message, it commands a movement in the negative x direction of the shared coordinate system. This movement has priority over the previously described gradient movement. The movement of these robots will create a "corridor" for the trapped robot to eventually enter.

As long as the trapped robot remains trapped, it will continuously send out the trapped robot message. Once it is no longer trapped, the message will stop. The robots that received the trapped robot message directly or indirectly will stop moving in the negative x direction, and revert to the gradient following behavior.

6) Random Robot Movement

If at any time a robot is unable to move in the commanded direction, (determined by not detecting a change in the robot's coordinates after a commanded movement) then it assumes it has bumped into another robot. When this occurs, there is a possibility that the robot can get stuck in a local minimum. In an example of this local minimum, shown in Fig. 4, the robot in location 1 tries to move in the direction to place it in location 2; however, it is prevented from doing so when it bumps into a robot in location 3. To prevent a robot from getting stuck in this minimum, when a bump is detected, a robot will move in a random direction, far enough to get out of the local minimum. This random movement will only occur if it will not take the robot from a location inside the shape segment to a location outside the shape segment, according to values of the gradient map. This random movement has the highest priority, and will occur instead of the gradient maximization movement, or the movement responding to a trapped robot message.

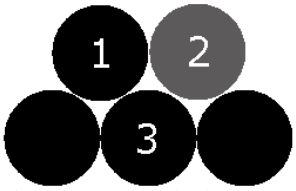


Fig. 4. Robot in local minimum. Black circles are robots, and the grey circle labeled 2 is a desired location for the robot labeled 1.

III. ANALYSIS

The goal of DASH is for the collective to scalably form and heal a desired shape. For the robot collective to have fully formed or healed the desired shape, every robot must be in a location that is within the shape, where the size of the shape is determined from the current N_r. This means that each robot is in a location that corresponds to an entry in the gradient map greater than -1.

To show that all robots will move into the shape, we will look at two possible cases. First, when a robot is on its own, there is no possibility of collisions or blocking from other robots. In this case, a single robot is capable of moving into the shape from any location outside the shape, whether external or trapped segments, by moving to maximize its gradient map entry. This works because if a robot is at the starting seed of an external or trapped segment, then gradient maximization will take it into the shape. If the robot is not at the starting seed, then gradient maximization will either take it into the shape, or to a starting seed. Within the external or trapped segment, the simple gradient maximization movement will not get stuck in local maxima, because there are no local maxima in the gradient map for these segments, which can be proven as follows.

Proposition: There is no local maxima in the gradient map for trapped or external segments.

Proof: The starting seed, S, for the external or trapped segment is not a local maximum, because it is always

immediately adjacent to a location in the shape segment which has a higher value in the gradient map. For any other location, L, in the segment there is a shortest path $L \rightarrow L_1 \rightarrow L_2 \rightarrow \cdots \rightarrow S$ from L to the starting seed S, where L_1 is an immediate neighbor of L. Due to the fact that every sub-path of a shortest path is also a shortest path for its respective start and finish points, the shortest path from L_1 is $L_1 \rightarrow L_2 \rightarrow \cdots \rightarrow S$, which is 1 less than the shortest path from L. The values in the gradient map for the external or trapped segment are set to be - (the length of the shortest path from that location to the starting seed+1), so every value in the gradient map for this segment must have an immediate neighbor in the gradient map with a higher value, and therefore is not a local maximum. ■

Once the single robot is inside the shape segment, the gradient maximization movement will not move it out of the shape. This is because the gradient map value of any location inside the shape segment is greater than the gradient map value of any location outside the shape segment.

The second possible case to look at which shows that every robot will move into the shape, is when more than one robot is trying to move into the desired shape. In this case, there is the possibility of neighboring robots blocking entrance into the shape. This blocking, called *blockade starvation,* can prevent the collective from fully forming the desired shape. Blockade starvation is when an area in the shape segment cannot be filled by a robot because the behavior of some robots inside the shape prevents robots from reaching this area. There are two types of blockade starvation: internal and external. An example of external blockade starvation is shown in Fig. 5A. In this form of blockade starvation, a robot in the external segment cannot move inside the shape, thus preventing the shape from being fully formed. An example of internal blockade starvation is shown in Fig. 5B. Here, a robot in a trapped segment is not capable of entering the desired shape, also preventing the shape from fully forming.

When external blockade starvation occurs, the empty area in the desired shape does not include any location that corresponds to a local minimum of the gradient map. This is because the generation of the gradient map in the shape segment guarantees that there is no local minimum, which can be proven in a similar fashion to the previous proof, however is excluded from this paper for brevity. If a robot immediately borders this empty area, and that robot has a corresponding gradient map entry lower than that its neighboring empty space (which is part of the empty area), then by the gradient movement rule, the robot will move into the empty space. Robots will continue to move into the empty area until either the empty area is filled with robots, or there are no robots next to the empty area that have a gradient map value less than the gradient map value corresponding to any of its empty neighbor spaces. If the latter is true, then, because there are no local minima, the empty area must include the starting pixel for the shape segment. This starting pixel for the shape segment is on the outside of the shape segment, where a robot in the external segment can reach. If this is the case, this empty space is no longer an external blockade starvation.

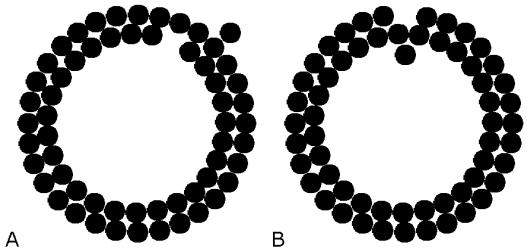


Fig. 5. Examples of A) external and B) internal blockade starvation of the desired shape shown in Fig. 1B.

When internal blockade starvation occurs, an empty area in the desired shape must become available to the trapped robot. This empty location is made available to the trapped robot using the trapped robot behavior. Similar to [7], but with less coordination between robots, the trapped robot behavior will create one or more empty spaces in the shape directly below (lower x value) the trapped robot. Other robots within the shape, but above this newly created space, will move into it, in effect propagating the empty space upward. Eventually, this empty space will reach the trapped robot, allowing it to move into the desired shape, and stopping the trapped robot behavior.

There are two side effects from this trapped robot behavior. The first is that it may introduce multiple empty locations inside the shape, below where the trapped robot was located; however, these locations are easily filled with robots using the gradient maximization movement. The second side effect is that while some robots were moving to create space for the trapped robot, they may have moved into an external or trapped segment. If they moved into the external segment, the gradient maximization movement will direct them back into the desired shape. If they moved into a trapped segment, and cannot move back into the desired shape, they will also need to use the trapped robot behavior. It is important to note that while the trapped robot behavior may create more trapped robots, it only creates them in locations below the original trapped robot. This means that there is no cycle where a robot trapped in a specific segment will cause more trapped robots in that segment. Without this problem of feedback, the trapped robot behavior will quickly cause the number of trapped robots to reduce to zero.

With the ability for each robot to move into the desired shape both by itself and when considering possible interference from other robots, the control method described in section 2 should always form the desired collective shape.

IV. Experimental verification

To verify that the DASH controller can indeed form a desired shape, we tested it on a simulated collective of robots. Each of these simulated robots was given the capabilities described in section 2.A, with R_{com} set to $6 \times R_{robot}$. To start with, the robots were distributed

randomly in the simulated world, and given the pixel map of the desired shape. In each simulated time step, the controller for each robot would run once, commanding a movement for the robot. At the end of each time step, the robots would make the commanded movement, if possible. At the end of each time step, the robots can also send messages to their neighbors, which were received by the receiving robot at the beginning of the next time step.

The collective was tested on 75 shapes, where 50 of them were chosen at random from [15], which is a library of real world objects, and the remaining 25 were drawn by students unaffiliated with our lab. Each image was given to a simulated collective at the beginning of the simulation run for a three simulation run series. Each of these three runs would first wait T_{damage} simulation steps, where the collective would form the desired shape, shown in Fig. 6A→B, and then apply the following damage: For the first run, the collective would be cut in half, where half the robots are removed (Fig. 6B→C). For the second run, the collective would be cut in half, and the upper half would be moved below the lower half, shown in Fig. 6B→D. For the final run in a series, more robots would be added near the collective, as shown in Fig. 6B→E. In each case, after damage was applied, the collective would reform the original shape until it was complete at a scale proportional to the new number of robots (Fig. 6F).

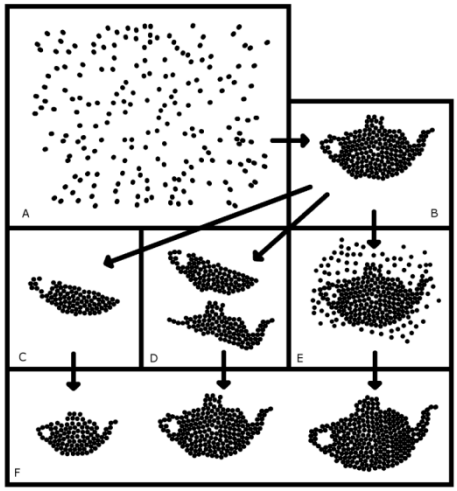


Fig. 6. Self assembly of the desired shape shown in Fig. 1C, the application of various forms of damage (C,D,E), and the scalable self-healing of the collective (F).

During each simulation run, at every time step we would measure the percentage of robots that were inside the desired shape, scaled appropriately based on N_r. The average of this value for all 225 simulation runs, for every time step is shown in Fig. 7. These results show that the DASH control method can self-assemble a simulated collective from a random starting configuration to a configuration where over 99% of the robots are within the desired shape. Furthermore, the control method can fully recover after various forms of damage back to a desired shape with over 99% of the robots in that shape.

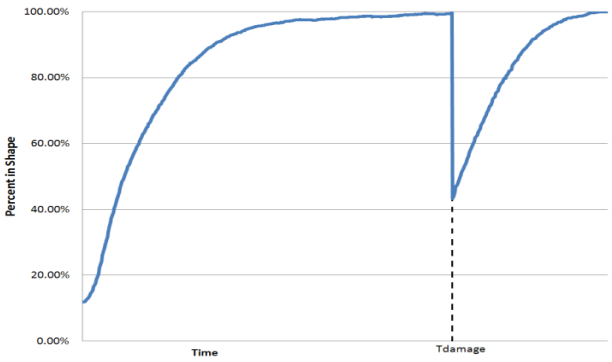


Fig. 7. The percent of robots in the desired shape, averaged over 75 test shapes. Damage was applied at the time of T_{damage}.

V. Conclusion

In this paper we have shown that the described distributed control method DASH can allow a collective of robots to scalably form many shapes. In the event of damage, the same method can reform the desired shape, but at a new scale proportional to the number of robots remaining. For videos of DASH running on a simulated collective, please visit www.isi.edu/robots/media.html

References

[1] H. Chiu, M. Rubenstein, W. Shen. *Deformable Wheel, A Self-Recovering Modular Rolling Track.* Intl. symposium on Distributed Robotic Systems, November 2008.

[2] R. Grady, R. Grob, A. Christensen, F. Mondada, M. Bonani, M. Dorigo. *Performance Benefits of Self-Assembly in a Swarm-Bot*. IROS 2007.

[3] K. Stoy, R. Nagpal. *Self-Repair Through Scale Independent Self-Reconfiguration*. IROS Sendai, Japan. 2004.

[4] D. Arbuckle, *Self-Assembly and Self-Repair by Robot Swarms*, Dissertation, University of Southern California, August 2007.

[5] M. Rubenstein, W. Shen. *A Scalable and Distributed Approach for Self-assembly and Self-Healing of a Differentiated Shape.* IROS 2008.

[6] M. Yim, J. Lamping, E. Mao, J. Chase. *Rhombic Dodecahedron Shape for Self-Assembling Robots*. SPL Technical Report P9710277. Palo Alto CA: Xerox PARC; 1997.

[7] M. Rosa, S. Goldstein, P. Lee, J. Campbell, P. Pillai. Scalable Shape *Sculpting Via Hole Motion : Motion Planning in Lattice-Constrained Modular Robots*. ICRA 2006.

[8] J. Cheng, W. Cheng,R. Nagpal, *Robust and Self-repairing Formation Control for Swarms of Mobile Agents*, AAAI july 2005 .

[9] A. Kondacs, *Biologically-inspired Self-Assembly of 2D Shapes, Using Global-to-local Compilation*, IJCAI 2003.

[10] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, G. Chirikjian. *Modular Self-Reconfigurable Robot Systems -- Challenges and Opportunities for the Future*. IEEE Robotics and Autonomation Magazine, March():43–53, 2007.

[11] H. Bode. *Head Regeneration in a Hydra*, Developmental Dynamics 226:225-236, 2003.

[12] P. Maxim,S. Hettiarachchi, W. Spears, D. Spears, J. Hamman, T. Kunkel, C. Speiser. *Trilateration localization for multi-robot teams.* Sixth International Conference on Informatics in Control, Automation and Robotics, Special Session on Multi-Agent Robotic Systems. 2008.

[13] Y. Shang, W. Ruml, Y. Zhang, M. Fromherz. *Localization Connectivity in Sensor Networks*. IEEE Transactions on Parallel and Distributed Systems, vol. 15 October 2004.

[14] D. Moore, J. Leonard, D. Rus, S. Teller. *Robust distributed network localization with noisy range measurements.* SenSys 2004.

[15] J. M. Geusebroek, G. J. Burghouts, A. W. M. Smeulders, The *Amsterdam library of object images*, Int. J. Comput. Vision, 61(1), 103-112, January, 2005.