Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations

D.J. Arbuckle · A.A.G. Requicha

Abstract Self-assembly of active, robotic agents, rather than of passive agents such as molecules, is an emerging research field that is attracting increasing attention. Active self-assembly techniques are especially attractive at very small spatial scales, where alternative construction methods are unavailable or have severe limitations. Building nanostructures by using swarms of very simple nanorobots is a promising approach for manufacturing nanoscale devices and systems.

The method described in this paper allows a group of simple, physically identical, identically programmed and reactive (i.e., stateless) agents to construct and repair polygonal approximations to arbitrary structures in the plane. The distributed algorithms presented here are tolerant of robot failures and of externally-induced disturbances. The structures are self-healing, and self-replicating in a weak sense. Their components can be re-used once the structures are no longer needed. A specification of vertices at relative positions, and the edges between them, is translated by a compiler into reactive rules for assembly agents. These rules lead to the construction and repair of the specified shape. Simulation results are presented, which validate the proposed algorithms.

D.J. Arbuckle (✉) · A.A.G. Requicha
Laboratory for Molecular Robotics, University of Southern California, 941 Bloom Walk, Los Angeles, CA 90089-0781, USA
e-mail: arbuckle@alumni.usc.edu

A.A.G. Requicha
e-mail: requicha@usc.edu

1 Introduction

Self-assembly is a process in which autonomous components join themselves to form more complex structures. Examples of self-assembly are all around (and within) us: atoms assemble themselves into molecules, supramolecular structures and crystals; molecules form membranes, organelles and cells; in turn, cells self-assemble into tissues and entire organisms. In contrast to what happens in nature, non-chemical engineered systems have not used self-assembly as a manufacturing process until now. (Chemistry itself is largely based on self-assembly processes.) Interest in self-assembly has been increasing rapidly over the last decade because it is an inherently parallel process that seems well-suited to the fabrication of complex structures from the bottom-up, using micro or nanoscale components. Many manufacturing processes are available at the macroscale, and, in our opinion, self-assembly is not attractive at such scales, except possibly for some niche applications. However, there are few other promising alternatives for the mass production of nanosystems.

The structures built by traditional (passive) self-assembly in the past (Adleman 2000; Winfree et al. 1998) have tended to be symmetric and not very useful from the point of view of such potential applications as nanoelectronics. (However, the more recent work on DNA origami Rothemund 2006 can produce arbitrary structures.) These passively self-assembled structures are built by systems that emulate chemistry: components exist in a medium that imparts on them the necessary motions for them to meet (often by thermal agitation), and complementary components attach to one another when they come into contact. Complementarity may have several forms, such as molecular recognition, hydrophobic/hydrophilic behavior, or Watson-Crick pairing for DNA strands. The work reported in this paper

was initiated before DNA origami techniques were known, and was motivated by a few simple questions such as the following. What if we make the components "active", by giving them a bit of "intelligence", by having sensors, control logic, and autonomous motion? In other words, what if the components are robots, albeit very simple ones? What can be built by such robots? Can they build arbitrary structures? How? What happens when the robots malfunction? What yields does one get? Can noise be rejected by non-global procedures?

A small number of tiny robots is not likely to self-assemble into something useful. Therefore, questions such as those posed above lead naturally to the study of robot swarms. Self-assembly is a special case of self-organization, and, in turn, a robot swarm is a special case of a distributed system. Therefore, the study of self-assembly by robot swarms raises a host of intellectually interesting and challenging problems related to emergent behavior, robustness, tolerance to faults, adaptability in dynamic environments, and the relationship between local and global behavior. Furthermore, we contend that self-assembly without self-repair is not likely to be practically useful, because it is almost certain that components will fail. The system must be able to compensate for component failure on the fly, or it will be down most of the time. The primary goal of this paper is to present algorithms for building self-assembling and self-repairing shapes by a swarm of robots whose assumed capabilities are inspired by those we expect to find in the nanorobots of the future. We present a complete solution to a specific instance of the difficult problem of "compiling" global behavior into local rules. We illustrate the behavior of the algorithms by extensive simulations.

There is a huge amount of literature that can be considered relevant to robotic self-assembly. Constructing a framework in which all this work can be put in perspective is a worthwhile task, but outside the scope of this paper. It is also a non-trivial task because the various systems and algorithms described in the literature make different assumptions, have different models and goals, and therefore are difficult to compare. Here we will simply list representative work in the diverse relevant areas, and provide citations for entries into the literature, without attempting to be complete. The following are related areas: swarm robotics (Dorigo and Şahin 2004; Şahin and Spears 2005); swarm intelligence in social insects and other biological systems (Bonabeau et al. 1999); self-replication (Zykov et al. 2007); passive self-assembly (Winfree et al. 1998; Rothemund 2006; FNANO 2006); modular self-reconfigurable robotics (Shen et al. 2000; Shen and Yim 2002; Rus and Chirikjian 2001); cellular automata (von Neumann 1966; Buchi and Siefkes 1989); and distributed robotics, especially the study of robot formations (Bahceci et al. 2003).

Three main research groups have recently addressed the problem of building arbitrary shapes by self-assembling robot swarms. They are Nagpal's group at Harvard (Nagpal 2002; Kondacs 2003; Stoy and Nagpal 2004; Werfel 2004, 2006), Klavins' at the University of Washington (Klavins 2004; Klavins et al. 2004, 2006; Bishop et al. 2005; Burden et al. 2006), and the robotics group at the University of Southern California (Wawerla et al. 2002; Jones and Matarić 2003; Arbuckle and Requicha 2004, 2005, 2006; Arbuckle 2007).

In this paper we present a novel, communication-based swarm algorithm that has strong self-repairing capabilities and parsimoniously uses resources such as memory and computation. The remainder of the paper is organized as follows. We begin with an overview of the model and algorithms in Sect. 2. Then we present a complete but very simple example in Sect. 3. Sections 4, 5 and 6 are devoted to a general discussion of the edge-building rules, the compiler, and the simulator. Results are presented in Sect. 7 and we draw conclusions in a final section.

2 Overview

We model the problem as follows. We assume that there is a very large number of robots, which are unit squares aligned with the x, y axes in a plane. The robots may move translationally in the plane, and initially execute random walks. All of the robots are identical and identically programmed. When two robots meet, they may exchange messages; while they are exchanging messages they remain attached to each other, but they cannot maintain their grasp if the messages stop. (It suffices for one of two connected robots to stop sending messages for the binding to break.) The robots are programmed by a set of reactive rules; when a robot receives a message, it consults its table of rules and executes the rule that corresponds to the received message, which usually requires sending other messages. Therefore, there is no internal (computational) state stored in the robots—the state of the system is "externalized" in the circulating messages. (Of course, the robots do have a physical state that is used by the simulator—see Sect. 6.) Rule execution may entail some simple arithmetic operations, typically incrementing or decrementing a hop counter. We seek algorithms compatible with minimalistic robots, with very limited memory and computational capabilities.

This model is inspired by the capabilities we expect in the nanorobots of the future. For example, communication is assumed to take place only between adjacent robots in contact, because nanorobots are likely to be able to communicate only chemically, by molecular recognition between molecules presented on their surfaces, or perhaps electromagnetically at very small distances. Assuming that the robots do not rotate may seem unduly restrictive, but we could

instead postulate self-aligning connections when robots become attached. Whether our model is adequate or not for swarm nanorobots must remain in the realm of speculation until such robots are built, but we believe that the assumptions are reasonable. Regardless of our motivation, we think that the model and algorithms are interesting on their own.

The input to our system is a polygon, defined by what is called in geometric modeling a boundary representation (Requicha 1980), i.e., a set of edges, or, equivalently, an ordered set of vertices, with two consecutive vertices defining an edge. Vertices are defined by their x, y coordinates in some arbitrary coordinate system for the plane. The input is fed to a compiler, which runs off-line in a regular computer such as a standard PC. First it performs a few simple geometric computations to deal with such problems as internal holes and sharp acute angles. The result of this first stage of the compiler is another polygon that can be easily approximated by a set of our square robots, forming a so-called spatial enumeration for the polygon (Requicha 1980).

Conceptually, our algorithm builds a polygon from the outside in, by first constructing its boundary and then filling up the interior. (In practice, boundary construction and interior filling operations proceed in parallel, but since the effects of the interior filling operations are small until the boundary is completed, we will ignore this fact to simplify the exposition.) The boundary is constructed outwards from some arbitrary point in it. Robots may attach to either endpoint of the evolving boundary. When the boundary is complete, we fill the interior by a process that is similar to diffusion across a porous membrane, and schematically shown in Fig. 1. Specifically, when a wandering robot attaches to another one already on the boundary, both move inward by one step, so that the new robot is now on the boundary and the old one is now in the interior of the polygon. The interior robot may simply be released and continue to wander around randomly, but inside the polygon, or a more sophisticated and efficient "guided" process can be used, although it requires more capable robots that can move along and follow the boundary of the polygon (Arbuckle and Requicha 2005). Details of the guided method may be found in Arbuckle (2007). Observe that a robot's ability to move in a specific direction is only needed for the filling procedures. If all that is wanted is a polygonal boundary and the medium imparts a random motion to the robots, these do not require on-board propulsion or steering capabilities. Video 1 shows an example of filling the interior of a structure with the guided method.

In Fig. 1 and similar figures throughout this paper, agents are represented as squares with one V shape and one inverted V shape attached to each side. These shapes represent both the ability to attach to neighboring agents and the ability to exchange messages with them, since both abilities are mutually dependent. When two agents are depicted with
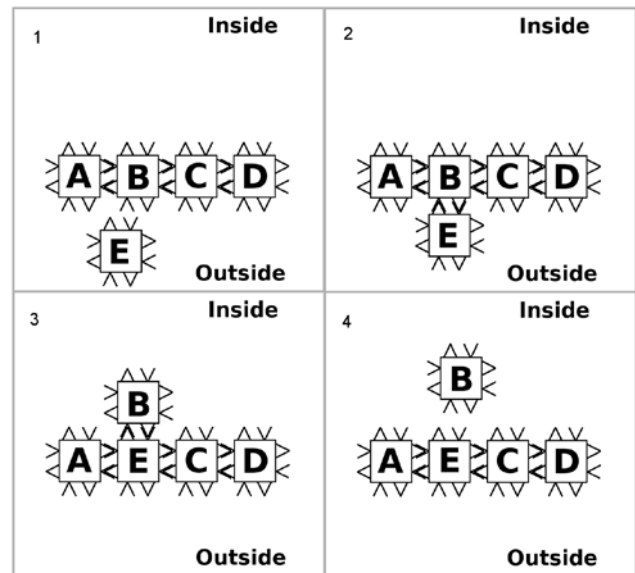

Fig. 1 (**1**) Robots A, B, C and D are taking part in a boundary, while robot E is wandering outside. (**2**) Wandering robot E attaches to the outer face of robot B, which is part of an edge being built. (**3**) Both robots move in by a distance equal to the robot's (edge) size. (**4**) Robot B detaches from E, which is now part of the boundary, and moves around randomly in the interior of the polygon. Agents connected by nested V shapes (e.g. C and D above) are joined together and can exchange messages

their V shapes nested together, this represents agents that are bound together and thus capable of exchanging messages. The messages themselves are represented by arrows in the figures.

The most complicated process is the construction of the polygonal boundary, and we will focus on it in the remainder of this paper. For simplicity of exposition, we consider only simple polygons, i.e., polygons without holes and whose boundaries do not self-intersect (or, mathematically, boundaries that are homeomorphic to a circle) although the system can accommodate arbitrary polygons. The compiler produces a set of parameterized generic rules for growing edges. These are discussed in Sect. 4 below—see also the example of Sect. 3. Suffice it to say here that a single set of *edge rules* is enough, regardless of how many edges a polygon may have. In addition to these generic edge rules, the compiler produces *vertex rules*, one set per each vertex of the polygon. A vertex rule is equivalent to stating *if you were reached from edge with $ID = X$, then emit a "grow edge" message with $ID = X'$ in direction D', and with length L'*. Identifiers for each of the edges are generated by the compiler. Four vertex rules are needed per vertex, two for each of the edges incident with it. It is very easy for the compiler to generate these vertex rules given the polygon's vertex list.

Given the complete set of vertex rules for a polygon, and assuming that the edge growing rules produce the desired result—we will return to this point in later sections—the

polygonal boundary is built correctly as long as robots can reach the current endpoints of the boundary construction. Since we assume that the random walk executed by the robots will eventually bring at least one robot to every point of the plane, the construction can only fail if there are not enough robots, or if an obstacle intersects the desired boundary.

Earlier work such as Arbuckle and Requicha (2004, 2005) showed that arbitrary polygons could be built by robot swarms programmed as finite-state machines (FSMs). Why did we decide to abandon the FSM approach and study reactive robots? Simply because we were unable to achieve the desired self-repair capabilities with the FSM algorithms. Problems arise because a robot's internal state, stored in its memory, can easily become incompatible with the real, physical situation, as a result of faulty operations. For example, a robot's memory may tell it that it is attached to another, when in reality it is not, but the sensor that would update its memory is malfunctioning. These incompatibilities result in errors in the structure being built. As we will see below, the process described here is self-healing. (We use "self-healing" and "self-repair" interchangeably in this paper.) However, this requires another set of messages besides those described so far. We explain these with the example of next section.

3 An example

The principles of operation of our algorithms are best introduced through a simplified example. Consider a large number of agents (robots) which move randomly on a plane. The agents are all identical unit squares (in some arbitrary units), orthogonally oriented, and have identical programs, which consist of sets of purely reactive rules. These rules prescribe actions to be taken when certain messages are received. Actions typically involve sending messages to other agents.

We consider for this example four types of messages, denoted GE, GV, AE and AV. The characters in these message names are mnemonics for grow (or build), G; acknowledge (or provide feedback), A; edge (or line), E; and vertex (or node), V. When a message is received in one of the four sides of the agent, it triggers the emission of other messages, which may be sent forward, backward, left or right, with respect to the direction of the incoming message.

Table 1 shows the ten rules needed for the example. The messages in the table may have up to three parameters: X is an edge identifier, P denotes the position of an agent within an edge, and L is the length of an edge. In the example we build a structure consisting of two vertices which bound an edge made from three agents ($L = 2$) and then start building a second edge, orthogonal to the first. The vertices are not considered part of the edge, and positions within an edge

Table 1 Rules for the example

Rule No.	Input	Output
1	$GE(X, P > 0, L)$	F: $GE(X, P - 1, L)$
2	$GE(X, P = 0, L)$	F: $GV(X)$
3	$GE(X, P < L, L)$	B: $AE(X, P + 1, L)$
4	$GE(X, P = L, L)$	B: $AV(X)$
5	$AE(X, P < L, L)$	F: $AE(X, P + 1, L)$
6	$AE(X, P = L, L)$	F: $AV(X)$
7	$AE(X, P > 0, L)$	B: $GE(X, P - 1, L)$
8	$AE(X, P = 0, L)$	B: $GV(X)$
9	$GV(X)$	LE: $GE(X', L', L')$
		B: $AE(X, 0, L)$
10	$AV(X)$	B: $GE(X, L, L)$

vary from $P = 0$ to L. The second column in the table shows the incoming message, which triggers the effects that appear in the third column. The output messages have a direction, relative to that of the incoming message. There are four such directions, corresponding to the four faces of the agent: F, for forward; B, for backward; R, for right; and LE, for left.

We begin with a seed, which consists of the pair of agents A and B, depicted at the top left of Fig. 2. The agents exchange the messages shown in the figure.

Observe that the message $GE(X, 2, 2)$ sent by A to B, by rule 4, causes B to send back to A the message $AV(X)$. In turn, when A receives this message, by rule 10, it sends back to B the message $GE(X, 2, 2)$. These GE and AV messages constitute a self-reinforcing loop, and the A–B group therefore is stable, i.e., it can exist indefinitely, remaining attached and continuing to exchange messages all along. Furthermore, by rule 1, B will attempt to send the message $GE(X, 1, 2)$ to any agent that may attach to its forward side. Note that this is a grow-edge message similar to that which B received from A, but with the position parameter P decremented by one.

Suppose now that an agent C in its random walk attaches to B as shown in Fig. 2. By rule 1, C will send forward a GE message with $P = 0$, and, by rule 3, it will send back to B an AE message with $P = L = 2$. Note that any other agent that attempts to attach to another face of A or B will eventually get detached because there are no messages being sent to it.

Continuing, D will attach to C and send forward, by rule 2, a grow-vertex message, and, by rule 3, send back to C an AE message with $P = 1$. Now, when E attaches to D it will, by rule 9, send an AE message back to D, and also start another edge, by sending $GE(X', L', L')$ to the left. The identity of this new edge, X', and its length, L', are preprogrammed in the agents' rule set.

The structure shown at the bottom of Fig. 2 reveals an interesting message pattern. Specifically, we have a set of grow-edge messages moving forward and decrementing a position counter until they reach a vertex, and also a second set of acknowledge messages moving backward and incre-
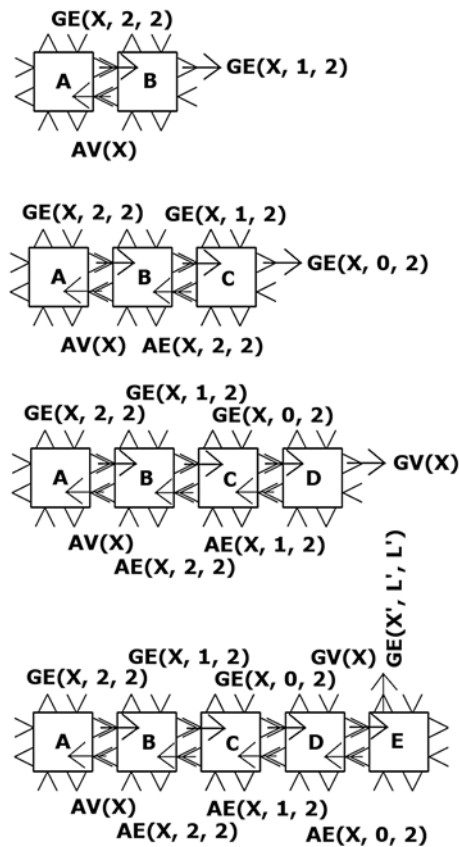
Fig. 2 Sequence of attachments and messages for the construction of a complete edge and the beginning of a second one, starting at *the top* and progressing towards *the bottom*

Fig. 3 Modified Bresenham's algorithm

menting the position counter. It is easy to see that any pair of contiguous agents forms a stable loop, much like A and B, as discussed above. Furthermore, any connected piece of the structure is also stable. This has important consequences that we will discuss now.

Suppose that agents A, B and E get detached due to exterior forces or because they malfunctioned and stopped sending messages. C and D will continue to exchange GE(X, 0, 2) and AE(X, 1, 2) messages indefinitely, per rules 3 and 7. They will also attempt to communicate backward and forward to other agents that may attach themselves to the group. Thus, if B attaches to C, by rule 7, it will send a GE message to C, and, by rule 6, it will send an AV message to any potential agent that may attach to it. Continuing this exercise and comparing with the bottom image of Fig. 2 shows that the original pattern is completely rebuilt. It follows that the self-assembly process introduced in this paper is self-repairing. It is also self-reproducing in the following sense. Suppose we take the final pattern of Fig. 2, break it into connected pieces that are at least two agents long, and move them apart. Each piece will repair itself and construct a whole pattern.

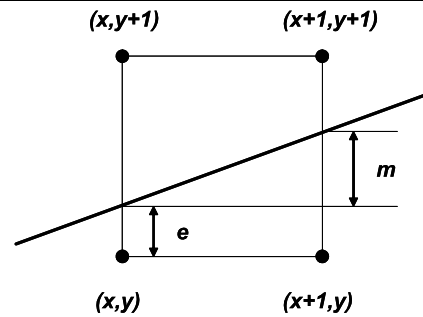It is interesting to note that a robot has no explicit localization mechanism, but its position relative to the other

robots in the structure is encoded in the parameter P of the grow edge messages that it receives. Hop counter incrementing and decrementing operations serve to keep track of positions in the structure, and to signal the end of edges.

4 Building edges

The "edge rules" 1–8 of Table 1 suffice to build any horizontal (i.e., in the x direction) or vertical (i.e., in the y direction) edge. These rules are parameterized by edge ID and length, L. The values of these parameters are set in "vertex rules" such as 9–10 in the table. When a robot receives a GE or AE message, it reads the values of ID and L in the message and instantiates the appropriate rule with those values bound to the parameters. If the generic edge rules 1–8 are placed in every robot's rule set, all that the compiler needs to do is to generate vertex rules for each vertex of the structure, ensuring that such rules pass the correct parameters to the generic edge rules.

What about edges that are not orthogonally oriented? This requires a more elaborate rule set. We approximate line segments with sets of connected agents by using a modified version of Bresenham's algorithm for scan conversion (Bresenham 1965). Bresenham's algorithm is well known in the computer graphics field. For completeness we review it briefly here, and explain how to modify it for our purposes.

Figure 3 illustrates the principles underlying the algorithm. We cover the plane with a uniform grid and associate the center of an agent's position to each node of the grid. For simplicity, we describe the algorithm for line segments in the first octant, i.e., with slopes not exceeding unity. Other octants are easily handled by using symmetry, and here and in the remainder of the paper we omit the details needed to deal with lines that are not in the first octant. The goal of Bresenham's algorithm is to compute an approximation to the line as a set of grid points.

We assume that we are processing the line from left to right, and that at a certain stage in the computation the line is being approximated by a grid node with coordinates (x, y) as shown in the figure. We continue to assume throughout this paper that the robots are oriented along the x, y axes.

Fig. 4 Two-step change of scan lines in the modified Bresenham's algorithm
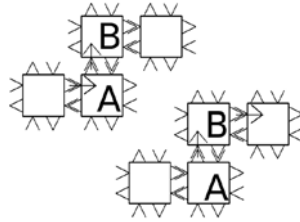
The error e between the line and its approximation is measured in the y direction. Let dx and dy be the lengths of the x and y projections of the entire line segment on the two principal axes, and let $m = dy/dx$ be the slope of the line. The lengths dx and dy are measured in grid nodes and therefore are integers; m is usually not an integer. The slope constraint implies that $dy \le dx$.

At point (x, y) we must decide whether to move along the same "scanline" (in the graphics terminology) to $(x + 1, y)$, or to change scanlines. In the original Bresenham algorithm, a scanline change is done by moving to $(x + 1, y + 1)$. For assembly agents, though, changing to a different scanline is a two-step process, because agents have no direct way to send a message (or attach themselves) diagonally. Instead, one agent has to send a message sideways to the direction of propagation of the received message, to $(x, y + 1)$, and then the agent that receives the message has to resend it sideways again, but this time at an angle that results in the message moving in the same direction as it was originally, i.e., to $(x + 1, y + 1)$. This results in the message reaching the correct position, but produces results that differ from Bresenham's line approximations by having an agent where the Bresenham algorithm would not produce a filled pixel. Figure 4 illustrates the process of changing to a new scanline. First, agent A receives a message, which its rules determine should be sent left. Then agent B receives that message and its rules determine that the message should be sent right. In both cases left and right are relative to the direction in which the message is propagating at the time the rules are applied to it, and so the net result is that the message is shifted to a different scanline and continues propagation in the same direction that it was originally moving.

Observe from Fig. 3 that between x and $x + 1$ the error increases by the value of the slope, to $e + m$. If the new error is less than 0.5, point $(x + 1, y)$ is the closest node, and we should remain in the same scanline. Otherwise, we should change scanlines, by the two-step procedure outlined above. In step 1 we move to $(x, y + 1)$ and the error becomes $e' = e - 1$. In step 2 we move to $(x + 1, y + 1)$ and the error is updated to $e'' = e' + m = e + m - 1$. These rules can be summarized as follows:

Same Scanline:

Test: $e + m < 0.5$
Update: $e \leftarrow e + m$

New Scanline:

Test: $e + m \ge 0.5$
Step 1: Update: $e \leftarrow e - 1$
Step 2: Update: $e \leftarrow e + m$

Computationally it is more efficient to work with an integer version of the algorithm. To do this we define another error measure as $\varepsilon = 2(e \cdot dx + dy)$ and rewrite the above rules in terms of it. Simple algebraic manipulations, taking into account that $m = dy/dx$, yield:

Same Scanline:

Test: $\varepsilon < dx$
Update: $\varepsilon \leftarrow \varepsilon + 2dy$

New Scanline:

Test: $\varepsilon \ge dx$
Step 1: Update: $\varepsilon \leftarrow \varepsilon - 2dx$
Step 2: Update: $\varepsilon \leftarrow \varepsilon + 2dy$

These rules use only integer operations. They are correct but insufficient. Comparing the error to the "high" threshold $H = dx$ is enough to determine if one should stay in the same scanline or move to a new one, and to execute the first step of the scanline change. Thus, in Fig. 4, agent A decides to send its message left by testing the error against H. But how is agent B to know that it should send a message to the right? We solve this problem by introducing a second, or "low" threshold $L = -dx + 2dy - 1$, which is not present in Bresenham's original algorithm. B will know that it is the result of step 1 of a change scanline operation if the error is below the low threshold. We prove in Appendix that this method is correct.

In summary, we compare the error ε with the two thresholds, H and L. If $L < \varepsilon < H$ we continue in the same scanline; if $\varepsilon \ge H$ we go left, i.e., we initiate step 1 of the change scanline protocol; and if $\varepsilon \le L$ we go right, i.e., execute step 2.

We are now ready to discuss a complete edge rule. The grow edge messages have the following parameters, most of which have obvious meanings, in light of the previous discussion.

- `edgeId`
- `counter`: this is analogous to the P parameter of Sect. 3 and counts the number of hops until the next vertex.
- `edgeLength`
- `epsilon`
- `highThreshold`
- `lowThreshold`
- `twoDx`
- `twoDy`

All of these parameters are constant while growing an edge, except for `counter` and `epsilon`. Therefore, we denote

below a grow edge message simply by GE (counter, epsilon). The messages sent in the opposite direction to the grow edge messages, called AE for "acknowledge edge" in the previous example, have the same parameters as the grow edge messages. Note that we could compute the thresholds on the fly and save some bandwidth, but we chose not to do so in the current implementation so as to reduce the number of computational operations required of the robots.

The edge rule is conceptually equivalent to the following self-explanatory pseudo-code inspired by the C++ family of languages. The directions forward, right and left are relative to the direction of the incoming message. The computation of epsilon in the AE messages will be explained shortly.

```
if GE(counter, epsilon) is received then {
  if (epsilon>lowThreshold and epsilon<highThreshold) then {
    if counter == 0 then send forward GV message
    else send forward GE(counter - 1, epsilon + twoDy);
    if counter == edgeLength then send backward AV message
    else send backward AE(counter + 1, epsilon - twoDy);
    }
  else if epsilon>=highThreshold then {
    if counter == 0 then send left GV message
    else send left GE(counter - 1, epsilon - twoDx);
    if counter == edgeLength then send backward AV message
    else send backward AE(counter + 1, epsilon - twoDy);
  }
  else if epsilon<=lowThreshold then {
    if counter == 0 then send right GV message
    else send right GE(counter - 1, epsilon + twoDy);
    if counter == edgeLength then send backward AV message
    else send backward GE(counter + 1, epsilon + twoDx);
  }
}
```

The updating rules for epsilon in the AE messages are justified as follows. Given the current value of epsilon, its previous value must either be $\varepsilon - 2dy$ or $\varepsilon + 2dx$, because of the form of the updating rules. The latter only applies when a step 2 of the change scanline protocol is executed. We argued earlier (and prove in Appendix) that we can tell when this happens simply by checking if $\varepsilon \leq L$.

This pseudocode generalizes rules 1 to 4 of Table 1 to oblique lines. We also need another rule for acknowledge edge (AE) messages, to generalize Rules 5 to 8 of Table 1 to arbitrarily-oriented lines. This AE rule is very similar to the GE rule shown above, can be derived in a similar manner, and will not be discussed further here. In summary, all the edges of a polygonal boundary can be constructed by using only one generic rule for GE messages, and another for AE messages. (Recall that we are here and elsewhere in the paper ignoring the fact that lines in different octants need to be accommodated.)

5 Compilation

The software module that converts a representation of the input polygon into the set of rules that constitutes the program of every robot is called the *compiler*, and runs off-line. In the previous sections we have already discussed many of the processes involved in the compilation, but it is worth consolidating here all the relevant information.

The input to the compiler is an array of $N + 1$ vertices, each defined by its x, y coordinates. Normally, the last and first vertices coincide, to ensure that the polygon closes, but the algorithms would also deal correctly with open polygonal boundaries, polygons with internal holes, and self-intersecting polygons. (For simplicity of exposition, we assume closed polygons without holes or self-intersections in the sequel.) The compiler performs several geometric computations to ensure that sharp acute angles, internal holes, and so on, are handled correctly. (These computations could be avoided if we placed some simple restrictions on the input polygons, and are not interesting from the viewpoint of swarm robotics.) Here we will ignore such details and assume that we have a polygonal boundary that can readily be approximated by a set of our robots. We must ensure that two adjacent robots on the boundary meet at a common face, and not just on vertices, otherwise message passing would be impossible and the structure would fall apart.

Two successive vertices define an edge, for a total of *N* edges. (Here we are simplifying things a bit, to avoid unnecessary and perhaps confusing details. The system can also deal with adjacent vertices, without any edge between them; such configurations are sometimes generated by the geometric pre-processor, e.g. when two edges meet at a very acute angle.) For each edge, the compiler computes its characteristic parameters such as length and slope, plus the thresholds H and L needed by the edge rules. It outputs a generic GE and a generic AE rule that were described in the previous section and generalize rules 1 through 8 of Table 1.

In addition, it outputs two vertex rules *per edge incident on each vertex*. These generalize rules 9 and 10 of Table 1. Therefore, the robot program occupies a constant amount of memory (for the generic edge rules) plus space linear on the number of edges (for the vertex rules). The size of the program depends on the physical dimensions of the polygon only through the number of bits of the integers that are used to encode the length of the edges and associated hop counters.

The compiler can be described by the following pseudo-code.

```
Write the generic edge rules discussed in Sect. 4;
Read array v[i],i=1,N+1, containing the vertex coordinates;
for i=1 to N do {
  e[i] = lineSegment(v[i], v[i+1]);
  edgeLength[i] = distance(v[i], v[i+1]);
  dx = project e[i] on x axis; twoDx[i] = 2 * dx;
  dy = project e[i] on y axis; twoDy[i] = 2 * dy;
  H[i] = dx;
  L[i] = -dx + 2*dy - 1;
  inDir[i] = closest axis to the direction of the edge;
  endEpsilon = execute edge rule and extract final epsilon;
  Write vertex rules for vertex i;
}
```

The distance between vertices is measured as the number of hops between the first and last non-vertices of the line segment. The variable `inDir` indicates the direction of the interior of the polygon, and helps implement the filling operation discussed in Sect. 2. A function `relativeDir` within the compiler uses the `inDir` values for two successive edges to find in which direction to send messages.

The vertex rules for a vertex incident with edges `edgeId` and `edgeId + 1` are as follows.

```
if GV(edgeId) is received then {
  i = edgeId;
  j = edgeId + 1;
  send in relativeDir(inDir[i], inDir[j])
    GE(j, counter=edgeLength[j], edgeLength[j], epsilon=0,
      H[j], L[j], twoDx[j], twoDy[j]);
  send backward AE(i, counter=0, edgeLength[i],
    endEpsilon[i], H[i], L[i], twoDx[i], twoDy[i]);
}
if AV(edgeId + 1) is received then {
i = edgeId;
j = edgeId + 1;
send backward GE(j, counter=edgeLength[j], edgeLength[j],
  epsilon=0, H[j], L[j], twoDx[j], twoDy[j]);
send in relativeDir(-inDir[j],-inDir[i])
  AE(i, counter=0, edgeLength[i], endEpsilon[i], H[i],
    L[i], twoDx[i], twoDy[i]);
}
```

These two vertex rules would suffice if we could guarantee that first `edgeId` would be built continuously in one direction, then the vertex would be reached, and then `edgeId + 1` would be built. Since this is not necessarily the case, we need two other rules similar to those above, and which cater to the opposite building order. We omit the code for these.

6 Simulation

The algorithms described in this paper are interesting when there are large numbers of robots, thousands or millions of them. In all likelihood, this will imply that the robots must be small, possibly nanoscopic. Identical robots in such large numbers, at such small spatial scales (or at any scales, for that matter), and with the capabilities we assumed for our agents, do not exist today. Therefore the algorithms cannot yet be validated experimentally. It should be clear that the algorithms build the desired structures correctly, unless there is an obstacle that intersects the polygonal boundary being built, or the number of robots is insufficient. (Obstacles in the interior of the polygon simply become surrounded by the structure and do not cause failure Arbuckle and Requicha 2005.) However, the self-healing capabilities of our scheme are not as obvious. Simulation is a valuable tool to experiment with the system for large numbers of (simulated) robots, and to understand its behavior.

The straightforward approach to building a simulator involves discretizing time and updating the swarm state at each clock tick. (Note that although the robots do not internally store state, in the Computer Science sense, each physical robot—and hence each simulated robot—has a physical state that characterizes it, and the swarm itself has a physical state as well; robot and swarm physical states must be known to the simulator, which manipulates them.) This type of simulation quickly becomes impractical as the swarm size increases beyond the hundreds. Therefore, we opted for a different approach. Our simulation is driven by a prioritized event queue. Each event has a time associated with it, and the highest priority in the queue corresponds to the lowest value of time. The event with highest priority is retrieved from the queue and processed, which normally results in other events being scheduled and added to the queue. After an event is processed, we advance the clock to the next time something interesting happens, i.e., to the scheduled time of the next event in the queue.

Before we embark on a more complete description of the simulator, we need to discuss certain details of our model that we have ignored thus far. We associate with each robot face a connection *strength*, which is zero if no other robot is attached to that face. This strength decays as time goes by. However, if the robot's face receives a message, the strength is increased. If the strength of a connection reaches zero, the robots become disconnected. This ensures that dead robots will be deleted from the structure. Hence, messaging is essential for keeping connectivity. Our connection strengths are integers, and we increase or decrease them by 1 when the strength is updated. There is a maximal value, M, at which strength saturates, but we will ignore it here for simplicity of exposition. Connection strengths in our model are a property of the hardware, and their behavior requires no software action.

Robots operate asynchronously. There is no need for a shared clock, which would be difficult to maintain in a distributed, dynamic environment, and in the presence of failures.

The messaging model used in the system is very simple. A message is an n-tuple, or, equivalently, a bit string. In Sect. 4 we described the various fields of a grow message, and others are similar. A robot that is instructed by its reactive rules to send a message makes the appropriate bit string available for reading at a specified face. If there is another robot attached to that face, the receiving robot has the ability to read the transmitted message. However, message processing takes time and the receiving robot may be busy processing a previous message when a new message appears. If so, the incoming message is simply dropped. There are no message buffers or queues. If a reactive rule calls for sending two or more messages, they are sent simultaneously out of different faces. The time that it takes to process a single message is an empirically-determined system parameter. This time implicitly defines a message density in the system.

Robots may become attached when they collide. When a robot motion event is processed, the trajectory of the robot is compared to the positions of other nearby robots. If a collision with a robot that belongs to a structure is detected, the motion stops and the colliding faces are identified. Collision detection is done with the help of a spatial grid, so that each robot only needs to be compared with others in nearby grid cells. (This is a standard speed-up technique in computational geometry.) Each robot moves instantaneously, by itself, while others are assumed stationary. This is physically inaccurate but computationally very convenient, since considering the joint, simultaneous motion of all the robots is very difficult.

The (physical) state of each robot in the simulation is kept in an array `robot[i]`. Each entry in this state array contains the following values:

- `position` in the x, y plane;
- `displacement` in x, y for the next movement;
- `binding`, an array of 4 entries corresponding to the 4 faces of the robot. Each entry indicates which face of another robot, if any, binds with the face under consideration, and with what strength;

- `fault` information describing any simulated failures that the robot is experiencing.

The fault condition stored in the array corresponds to a situation in which a robot ceases to transmit or respond to messages. The prioritized event queue maintains a shared time variable called `now`. Each event in the queue contains the following data:

- `time`;
- `target` robot;
- `action` to take upon the `target` robot;
- `face` associated with the `action` (potentially null);
- `parameter` to apply to the `action` (also potentially null).

The simulation uses several random variables, which are sampled to determine variable values needed by the process. They are the following:

- P: sampled for `position`;
- V: sampled for `displacement`;
- T: sampled for event `time`;
- D: sampled for the time of a binding strength decay event;
- C: sampled to decide whether or not to change `displacement`, i.e., the length and direction of the next motion of a robot.

The first 4 of the variables above are Gaussian and the fifth is binary. When P has a broad distribution, robots' positions are initially almost uniform over the region where the simulation takes place. One can think of T and V as playing roles related to physical temperature and energy. When the increments of `time` are small, the density of events increases, which is also what happens in physical systems at high temperature. The `displacement` values are large when the robots travel fast, i.e., have high kinetic energy. When the random variable C has the value 1, the `displacement` of a robot changes, i.e., the robot's direction and velocity change, which is what would happen if it collided with particles in the medium. A distribution of C skewed towards 1 corresponds to a medium with a high density of non-robot colliding particles.

The initial state of the simulation is determined as follows. For $N - 2$ robots, each component of `position` is chosen by sampling P, each component of `displacement` is chosen by sampling V, `binding` is set to no bindings, and `fault` is set to no faults. Two more robots are created in a similar way, except that they are placed immediately adjacent to one another, and bound together. The event queue is initialized with `now = 0`, a message delivery event targeting one of the bound robots, with time chosen by sampling T, two binding decay events with time chosen by sampling D, each targeting one of the bound robots, and motion events for all the other robots. As we saw in earlier sections of this paper, two correctly functioning robots that form a stable loop are sufficient to build the goal structure.

Simulation execution proceeds thusly:

```
while queue not empty do {
  e = event in the queue with the lowest time;
  now = e.time;
  if (e.action == MESSAGE) then {
    e.target.receiveMessage(e.face, e.parameter);
    e.target.binding[e.face].strength += 1;
    addToQueue(DECAY, e.target.binding[e.face], time = now + sample(D));
  }
  else if (e.action == DECAY) then {
    e.target.binding[e.face].strength -= 1;
    if (e.target.binding[e.face].strength == 0) then {
      otherRobot = robot bound to e.target on e.face;
      otherFace = bound face of otherRobot;
      e.target.binding[e.face] = null;
      otherRobot.binding[otherFace] = null;
      // Disconnect the two robots
    }
  }
  else if (e.action == MOVE) then {
    check for collisions for motion from e.target.position
    to e.target.position + e.target.displacement;
    if (no collision) then {
      e.target.position += e.target.displacement;
      if (sample(C) == 1) then
```

```
        e.target.displacement = sample(V);
    addToQueue(MOVE, e.target, time = now + sample(T);
    }
    else {
      e.target.position = collision location;
      // Binding involves a robot alignment process
      e.target.binding[collision face] = collided robot with strength 1;
      addToQueue(DECAY, e.target.binding[collision face], time = now + sample(D));
      }
    }
}
```

Here the `receiveMessage` procedure triggers the execution of the appropriate edge and vertex rules discussed in earlier sections. These executions, in turn, generate more events that get added to the queue.

7 Results and discussion

We have tested the algorithms extensively, by running on the order of 200 simulations, each involving from 4,000 to 10,000 robots. Figure 5 shows a few examples of structures constructed by the system. All of them have complete boundaries, but their interiors are at varying degrees of completeness, which depend on when the process was (manually) stopped. Note that the nature of the algorithms implies that interior completion can only be ensured in an asymptotic and probabilistic sense. Robot colors in the figures are assigned by the simulator: red robots are receiving vertex messages, blue robots are receiving edge messages, and black robots have not recently received any message. Figure 6 illustrates the construction process by showing a few frames produced by the simulator while building the boundary of an asymmetric star.

The self-repair capabilities of the algorithms were tested by simulating communication faults. All of the simulations, including those shown in the figures, have a probability of message dropping and message corruption. For every message transmission we assume a 5% probability that the message will be dropped, and (independently) a 10% probability that it will be corrupted. Message corruption is accomplished by randomly flipping one bit of the message. These errors have never prevented successful completion of polygonal boundaries in any of our experiments, except during the vulnerable period when only the original two robots are exchanging messages, before any others have joined the structure. When either of these two probabilities reaches a structure-dependent higher value, typically on the order of 20%, the structure tends to fall apart. In one experiment, an 8×8 square tolerated a 20% drop rate.

We also simulated another type of fault, in which robots stop sending or receiving messages. This is done by setting fault conditions on the robots' (physical) state array, as explained in the Simulation section. Figure 7 shows a few frames of a simulation in which a key-like structure is being built when a group of (yellow) robots near the lower end of the key become faulty. The faulty robots are detached from the others because their connection strengths decrease to zero. Then they wander away under random motion, and

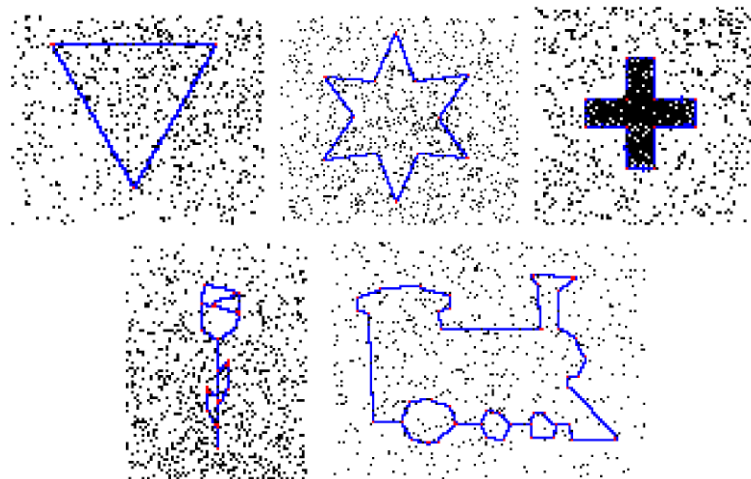**Fig. 5** Examples of structures built by the system

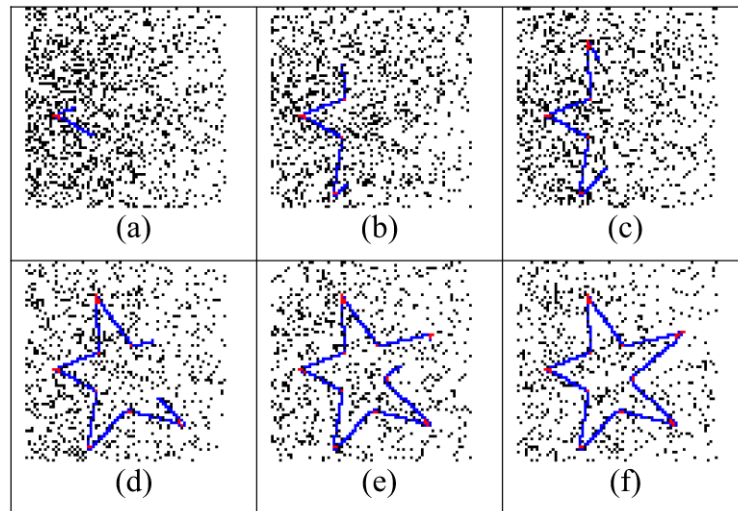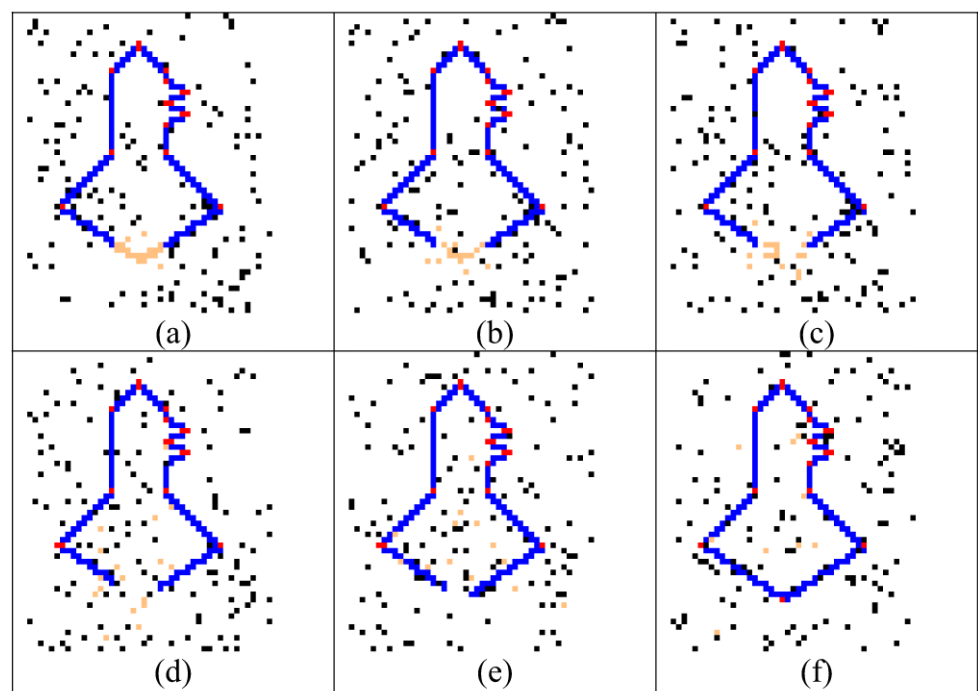**Fig. 6** Construction of a complex asymmetric shape

Fig. 7 A completed key-like boundary is damaged by forcing the robots around the structure's lower point to fail. The structure recovers from the damage and self-repairs

are replaced by new, well-functioning robots. This results in a correct, self-repaired structure.

Finally, we tested the system's ability to self-replicate. This is a weak form of self-replication since an external action is required to break a structure being built. Figure 8 and Video 2 illustrate this behavior. The key-like structure of Fig. 7 is being built when an external force (imposed manually by the experimenter) breaks the structure into two. Each of these regenerates the goal structure.

Message dropping also occurs naturally in the system when the robots cannot keep up with the incoming messages, and simply ignore them. The tolerance of our method to message corruption and dropping comes from the depen-

dence of swarm behavior on reinforcement. A randomly-corrupted message may cause a robot to attempt a binding on one of its faces, but it is very unlikely that the same corrupted message will be transmitted with enough frequency to keep that binding request active. Similarly, random message dropping tends not to matter as long at it is moderate, because it is unlikely that enough messages will be dropped for a binding to break. (Complete cessation of message transmission does cause bindings to break, as shown in the example of Fig. 7, but the structures are later healed, as long as there are enough well-functioning robots.) The system is robust in the presence of transmission errors, but is not entirely immune to such errors. The probability of
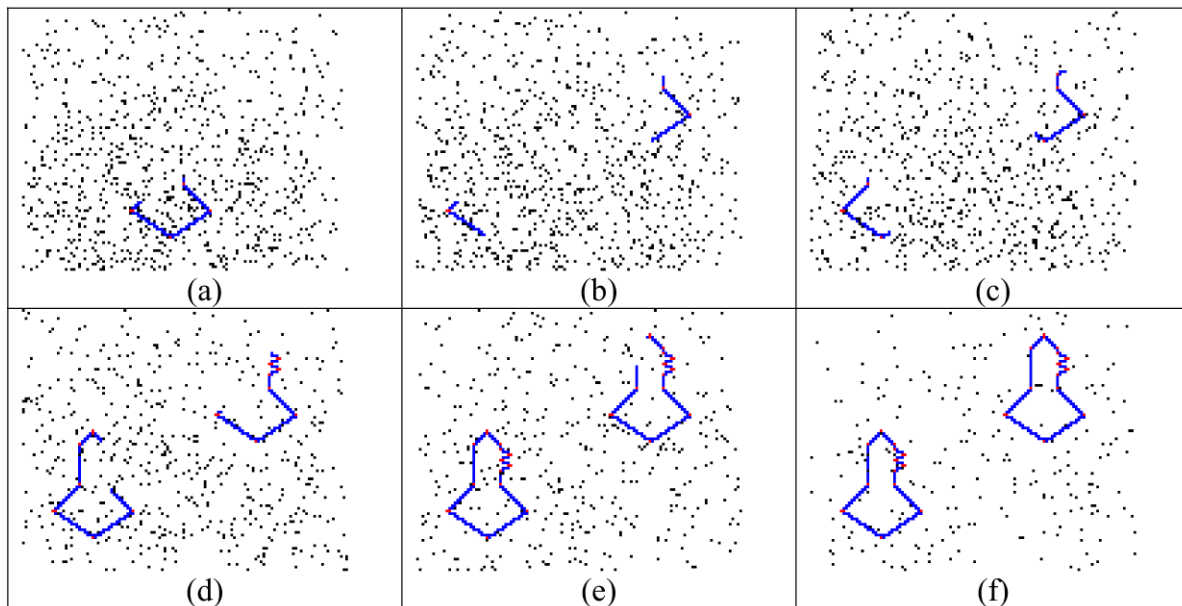
Fig. 8 A partially completed key-like shape is broken in two, with the result that two complete copies of the structure are created

corrupted messages building an erroneous reinforcing loop is very small but not zero, and we have observed rare examples of faulty constructions due to message corruption. These happened when we let the process run for a long time after the boundary was completed.

The efficiency of the algorithms needs further, more systematic study. Recent work (Tangchoopong and Requicha 2009) shows that the time required to complete the boundary of a square is linear in the size of the square, measured by the number of robots needed to form an edge. These results have numerous qualifiers; for example, they depend strongly on the assumed statistics of the initial robot distribution. The problem of meaningfully evaluating the performance of algorithms such as ours remains largely open.

The memory requirements of the agents are quite low, consistent with our desire to use vast numbers of simple robots for construction tasks. (Note that, although the robots do not store their computational state, they need memory to store their rules, or programs.) The number of fields per message is constant, with 11 being sufficient, and so the number of fields per rule is also constant. (For example, the edge rules of Sect. 4 have 8 parameters, plus a message type and two additional parameters needed for capabilities not described in this paper.) The sizes of fields are either constant or grow with the log of the maximum line length in the structure. The number of rules that must be stored in memory grows linearly with the number of vertices. Thus the memory requirements for even very complex structures on scales much larger than the individual agents are not taxing. For example, the set of specific rules for building the polygonal boundary of Fig. 7 occupies a space of 812 bytes.

If we expand the maximum line length to 256 agents the program grows to 1044 bytes, and if we expand the maximum line length to approximately 2×10^9 agents, the program grows to only 4814 bytes.

If the robots' memory is writable by external means, then the robots are recyclable. The same group of robots can be used to form one structure, then tossed back into the mix and reprogrammed for a different structure at any time, and as many times as desired. There is no physical difference between robots that construct one structure and agents that construct a different one.

8 Conclusion

This paper describes a method by which swarms of simple and identical reactive agents (i.e., robots) can be programmed to build and repair structures from a large class of geometric forms. As a side effect of the behavior of the agents, the structures built from them exhibit the ability to heal, and the ability to reproduce when they are broken by an external force. The assembly agents may be re-used when the structures are no longer needed. The algorithms use very limited computational resources and are suitable for implementation in minimalistic robots.

We present a complete solution of the global-to-local compilation problem for a shape-building task. The compiler we designed and implemented generates automatically the reactive rules used by the robots to construct any polygon in the plane.

The algorithms discussed in this paper are most attractive at small scales and with large numbers of agents. Unfortunately, micro or nanoscale artificial physical agents with the characteristics assumed here do not yet exist. This paper serves as motivation to build such agents, since the arbitrary structures constructed by them could find applications ranging from scaffolds for nanoelectronic component placement, to cell repair and tissue regeneration, to construction of macroscale goods.

We believe, but have not yet demonstrated, that the techniques introduced here may be extended to three dimensions either by working on successive slices of the object to be produced (as in rapid prototyping systems), or by first building faces as two dimensional structures and then filling in the three dimensional interior of the resulting polyhedral approximation.

Several interesting issues remain open, some specific to the algorithms presented here, others of a more general nature. Here are some examples. What are the completion rates for the algorithms? How do they depend on the shapes being built? What are the effects of positional uncertainties, and how can these be mitigated? How are self-repair capabilities to be assessed quantitatively? How is the performance of self-organizing algorithms, including robustness and adaptability, to be measured in the presence of a dynamic environment? How is parallelism related to the shapes being built and the algorithms being used to build them? What is the interplay between state, communications, and self-repair?

Appendix: Proof of the modified Bresenham's algorithm

First note that $H \geq L$. This follows from $dx \geq dy$ by adding $-dx + dy - 1$, which is a negative number, to the right side of the previous inequality. Suppose now that we are at point (x, y) where the decision is to continue in the same scanline. Then we must have $\varepsilon \leq dx - 1$, because otherwise we would need to change lines since the error would reach the high threshold. Moving now to the next pixel along the same scanline, the error will increase by $2dy$. Adding this quantity to both sides of the inequality just above yields $\varepsilon + 2dy = \varepsilon' \leq dx + 2dy - 1$. Since we are analyzing step 1 of the scanline change protocol, we also know that the error must now be above the high threshold, i.e., $\varepsilon' \geq dx$. Therefore, before step 1 of the scanline change protocol we have the following inequalities: $dx \leq \varepsilon' \leq dx + 2dy - 1$. Now, executing step 1 will decrease the error by $2dx$. Subtracting this quantity from the previous inequality yields $-dx \leq \varepsilon'' \leq -dx + 2dy - 1 = L$. This proves that the error is below

the low threshold after step 1 of a scanline change. Executing step 2 increases the error by $2dy$. Adding this quantity to both sides of the left inequality just above reveals that $-dx + 2dy \leq \varepsilon'''$, and *a fortiori* also $-dx + 2dy - 1 \leq \varepsilon'''$. This proves that after step 2 of a change of scanline the error is always larger than the low threshold. It remains to be shown that the error is also above L when we move along the same scanline. This is trivial since, after the first pixel, the error always increases by $2dy$ when we move to the next pixel, and $2dy > L$. Therefore an agent can tell that it is the result of step 1 of a change scanline operation by checking that its error value is below L.

References

Adleman, L. M. (2000). *Toward a mathematical theory of self-assembly* (Technical Report 00-722). University of Southern California, Department of Computer Science.

Arbuckle, D. J. (2007). *Self-assembly and self-repair by robot swarms*. Ph.D. Dissertation, University of Southern California, Los Angeles, CA.

Arbuckle, D. J., & Requicha, A. A. G. (2004). Active self-assembly. In *Proc. IEEE int'l conf. on robotics & automation (ICRA'04)*, New Orleans, LA (pp. 896–901), 25–30 April 2004.

Arbuckle, D. J., & Requicha, A. A. G. (2005). Shape restoration by active self-assembly. *Applied Bionics and Biomechanics*, 2(2), 125–130. (An earlier version appeared in *Proc. int'l symp. on robotics & automation (ISRA'04)*, Querétaro, Mexico (pp. 173–177), 25–27 August 2004).

Arbuckle, D. J., & Requicha, A. A. G. (2006). Self-repairing self-assembled structures. In *Proc. IEEE int'l conf. on robotics & automation (ICRA'06)*, Orlando, FL (pp. 4288–4290), 15–19 May 2006.

Bahceci, E., Soysal, O., & Şahin, E. (2003). *A review: pattern formation and adaptation in multirobot systems* (Technical Report CMU-RI-TR-03-43). Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., Napp, N., & Nguyen, T. (2005). Self-organizing programmable parts. In *Proc. IEEE/RSJ int'l conf. on intelligent robots & systems (IROS'05)*.

Bonabeau, E., Dorigo, M., & Théraulaz, G. (1999). *Swarm intelligence—from natural to artificial systems*. London: Oxford University Press.

Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 25–30.

Buchi, J. R., & Siefkes, D. (Eds.) (1989). *Finite automata, their algebras and grammars: towards a theory of formal expressions*. New York: Springer.

Burden, S., Napp, N., & Klavins, E. (2006). The statistical dynamics of programmed robotic self-assembly. In *Proc. int'l conf. on robotics & automation (ICRA'06)*, Orlando, FL (pp. 1469–1476), 15–19 May, 2006.

Dorigo, M., & Şahin, E. (Eds.) (2004). Special issue on swarm robotics, *Autonomous Robots*, 17(2–3).

FNANO (2006). *Proc. foundations of nanoscience 2006—self-assembled architectures and devices*, Snowbird, Utah, 23–27 April 2006.

Jones, C. V., & Matarić, M. J. (2003). From local to global behavior in intelligent self-assembly. In *Proc. IEEE intl. conf. on robotics & automation (ICRA'03)*, Taipei, Taiwan (pp. 721–726), 14–19 Sep 2003.

Klavins, E. (2004). Universal self-replication using graph grammars. In *Proc. int'l. conf. on MEMs, NANO and smart systems*, Banff, Canada (pp. 198–204).

Klavins, E., Ghrist, R., & Lipsky, D. (2004). Graph grammars for self-assembling robotic systems. In *Proc. IEEE int'l conf. on robotics & automation (ICRA'04)*, New Orleans, LA, 25–30 April 2004.

Klavins, E., Ghrist, R., & Lipsky, D. (2006). A grammatical approach to self-organizing robotic systems. *IEEE Transactions on Automatic Control*, *51*(6), 949–962.

Kondacs, A. (2003). Biologically inspired self-assembly of two-dimensional shapes using global-to-local compilation. In *Proc. 18th annual joint conf. on artificial intelligence*, Acapulco, Mexico (pp. 633–638), 9–15 August 2003.

Nagpal, R. (2002). Programmable self-assembly using biologically-inspired multiagent control. In *Proc. 1st int'l joint conf. on autonomous agents and multiagent systems: part 1*, Bologna, Italy (pp. 418–425).

Requicha, A. A. G. (1980). Representations for rigid solids: theory, methods and systems. *ACM Computing Surveys*, *12*(4), 437–464.

Rothemund, D. W. K. (2006). Folding DNA to create nanoscale shapes and patterns. *Nature*, *440*, 297–302.

Rus, D., & Chirikjian, G. S. (Eds.) (2001). Special issue on self-reconfigurable robots. *Autonomous Robots*, *10*(1).

Şahin, E., & Spears, W. M. (Eds.) (2005). *Lecture notes on computer science*: Vol. 3342. *Swarm robotics*. Berlin: Springer.

Shen, W.-M., & Yim, M. (Eds.) (2002). Special issue on self-reconfigurable robots. *IEEE/ASME Transactions on Mechatronics*, *7*(4).

Shen, W.-M., Lu, Y., & Will, P. (2000). Hormone-based control for self-reconfigurable robots. In *Proc. intl. conf. on autonomous agents*, Barcelona, Spain (pp. 1–8), 3–7 June 2000.

Stoy, K., & Nagpal, R. (2004). Self-repair and scale-independent self-reconfiguration (for modular robots). In *Proc. IEEE/RSJ int'l conf. on intelligent robots & systems (IROS'04)*.

Tangchoopong, T., & Requicha, A. A. G. (2009). An empirical study of the performance of active self-assembly. In *Proc. IEEE/RSJ int'l conf. on intelligent robots and systems (IROS'09)*, St. Louis, MO, 11–15 October 2009.

von Neumann, J. (1966). *The theory of self-reproducing automata*. Champaign: University of Illinois Press.

Wawerla, J., Sukhatme, G., & Matarić, M. (2002). Collective construction with multiple robots. In *Proc. IEEE/RSJ int'l conf. on intelligent robots & systems (IROS 2002)*, Lausanne, Switzerland, September 30–October 4 2002.

Werfel, J. (2004). Building blocks for multi-robot construction. In *Proc. int'l symp. on distributed autonomous robotic systems (DARS'04)*, Toulouse, France, 23–25 June 2004.

Werfel, J. (2006). *Anthills built to order: automating construction with artificial swarms*. Ph. D. Dissertation, Massachusetts Institute of Technology.

Winfree, E., Liu, F., Wenzler, L. A., & Seeman, N. C. (1998). Design and self-assembly of two-dimensional DNA nanocrystals. *Nature*, *394*, 539–544.

Zykov, V., Mytilinaios, E., Desnoyer, M., & Lipson, H. (2007). Evolved and designed self-reproducing modular robotics. *IEEE Transactions on Robotics*, *23*(2), 308–319.

D.J. Arbuckle received his Ph.D. in Computer Science from the University of Southern California.
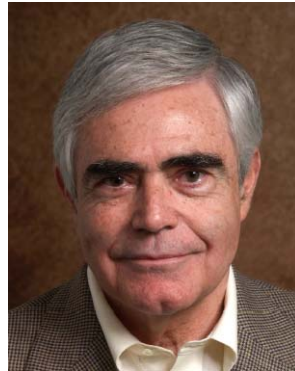
A.A.G. Requicha was born in Monte Estoril, Portugal, in 1939. He received the Engenheiro Electrotécnico degree from the Instituto Superior Técnico, Lisbon, Portugal, in 1962, and the Ph.D. in electrical engineering from the University of Rochester, Rochester, NY in 1970. He was a college and high school Valedictorian, and is listed as a highly cited author in the ISI Web of Science for 1980–1999. He was awarded the Pioneer in Robotics and Automation Prize of the IEEE (Institute of Electrical and Electronic Engineers) Robotics and Automation Society in 2008, the first-ever Pierre Bézier Prize of the Solid Modeling Association in 2007, and the Senior Research Award of the Viterbi School of Engineering of the University of Southern California in 2006. He is a Fellow of the IEEE, of the ACM (Association for Computing Machinery) and of the AANM (American Academy of Nanomedicine), and is a member of Computer Graphics Pioneers and Sigma Xi.

Dr. Requicha holds the Gordon Marshall Chair in Engineering and is a Professor of Computer Science and Electrical Engineering at the University of Southern California, where he also directs the Laboratory for Molecular Robotics. He has served in numerous conference program committees and journal editorial boards. His past research focused on geometric modeling of 3-D solid objects and spatial reasoning for intelligent engineering systems. Currently he is working on robotic manipulation of nanometer-scale objects using scanning probe microscopes; nanorobot components and nanorobotic system integration; fabrication of nanostructures by robotic self-assembly; sensor/actuator networks; and applications in NEMS (nanoelectromechanical systems) and nanobiotechnology. The long-term goals are to build, program, and deploy nanorobots and networks of nanoscale sensors/actuators for applications to the environment and health care.

Dr. Requicha currently is the Editor-in-Chief of the IEEE Transactions on Nanotechnology, and co-chairs the Micro/Nano Robotics and Automation Technical Committee of the IEEE Robotics and Automation Society. He is a member of the AAAI (Association for the Advancement of Artificial Intelligence), AAAS (American Association for the Advancement of Science), AVS (American Vacuum Society) and SME (Society of Manufacturing Engineers). Personal web page: http://www-lmr.usc.edu/~requicha.