Self-Repair Through Scale Independent Self-Reconfiguration

K. Stoy
The Maersk Mc-Kinney Moller Institute for Production Technology
University of Southern Denmark
Odense, Denmark
Email: kaspers@mip.sdu.dk

R. Nagpal
Division of Engineering and Applied Sciences
Harvard University
Cambridge, USA
Email: rad@eecs.harvard.edu

***Abstract*— Self-reconfigurable robots are built from modules which are autonomously able to change the way they are connected, thus changing the overall shape of the robot. This self-reconfiguration process is difficult to control, because it involves the distributed coordination of large numbers of identical modules connected in time-varying ways.**

We present an approach where a desired shape is grown based on a scalable representation of the desired configuration which is automatically generated from a 3D CAD model. The size of the configuration is adjusted continually to match the number of modules in the system. This has the advantage that if modules are removed or added, the system automatically adjusts its scale and thus self-repair is obtained as a side effect. This capability is achieved by distributed, local rules for module movement that are independent of the goal configuration.

We compare the scale independent approach to one where the desired configuration is grown directly at a fixed scale. We find that the features of the scale independent approach come at the expense of an increased number of moves, messages, and time steps taken to reconfigure.

I. Introduction

Reconfigurable robots are built from modules and can be reconfigured by changing the way the modules are connected. If a robot is able autonomously to change the way the modules are connected, the robot is a self-reconfigurable robot. Self-reconfigurable robots are versatile because they can adapt their shape to fit the task. They are also robust because if a module fails it can be ejected and replaced by a spare module. Potential applications for such robots include search and rescue missions, planetary exploration, building and maintaining structures in space, and entertainment. Challenges exist both in the development of the hardware for the modules, as well as their control. This paper focuses on the challenge of controlling reconfiguration in a robot with many identical modules.

In this paper we present an approach to self-reconfiguration that is *scale independent* — in other words, the size of the goal configuration automatically adjusts to the number of available modules in real-time. This approach relies on a two-step process of self-reconfiguration, first presented in [1]. First, a 3D CAD model representing the desired configuration is transformed into a scalable geometric representation based on overlapping bricks of different sizes. The representation is supplemented with a scaffold structure which removes local minima, hollow, or solid sub-configurations from the configuration. The second step is the self-reconfiguration step which can be viewed as a directed growth process. A user starts the process by choosing an arbitrary initial seed module and assigning it a position in the desired configuration. The seed attracts spare modules by creating a recruitment gradient in the system. Wandering modules climb the gradient to fill positions, and then become seeds themselves and attract more neighbors as needed. Several key features of this approach were highlighted in [1], such as the automatic generation of representation, representation size proportional to the complexity of shape and not the number of modules, and configuration-independent local rules for module movement and connectivity maintainance [2].

In this paper we demonstrate that this approach can also be used to self-reconfigure into a desired configuration, even if the number of modules is not known a priori. Furthermore, modules can be added or removed at run-time, and the configuration automatically readjusts its size. Figure 1 shows an example of this approach. This scale independent reconfiguration is achieved by modifying the local rules used by the modules to grow: A seed module increases the scale of the representation until the seed's position is contained. It then attracts spare modules by creating a recruitment gradient tagged with the scale. Spare modules climb the gradient tagged with the smallest scale and thus fill the corresponding positions first. In this way the configuration is built layer by layer and continually adjusts to the availability of spare modules. This, as a side effect, makes it possible for the system to self-repair, because if modules are removed or lost the system automatically adjusts its size to match the remaining number of modules. The cost of this novel capability is a reduction in performance, in terms of number of moves, time steps, and messages, compared to one where the configuration is grown directly at a fixed scale.

II. Related Work

The self-reconfiguration problem is: given a start configuration, possibly a random one, how to move the modules in order to arrive at the desired final configuration. It is computational intractable to find the optimal solution (see [3] for a discussion). Therefore, self-reconfiguration planning and control are open to heuristic-based methods.

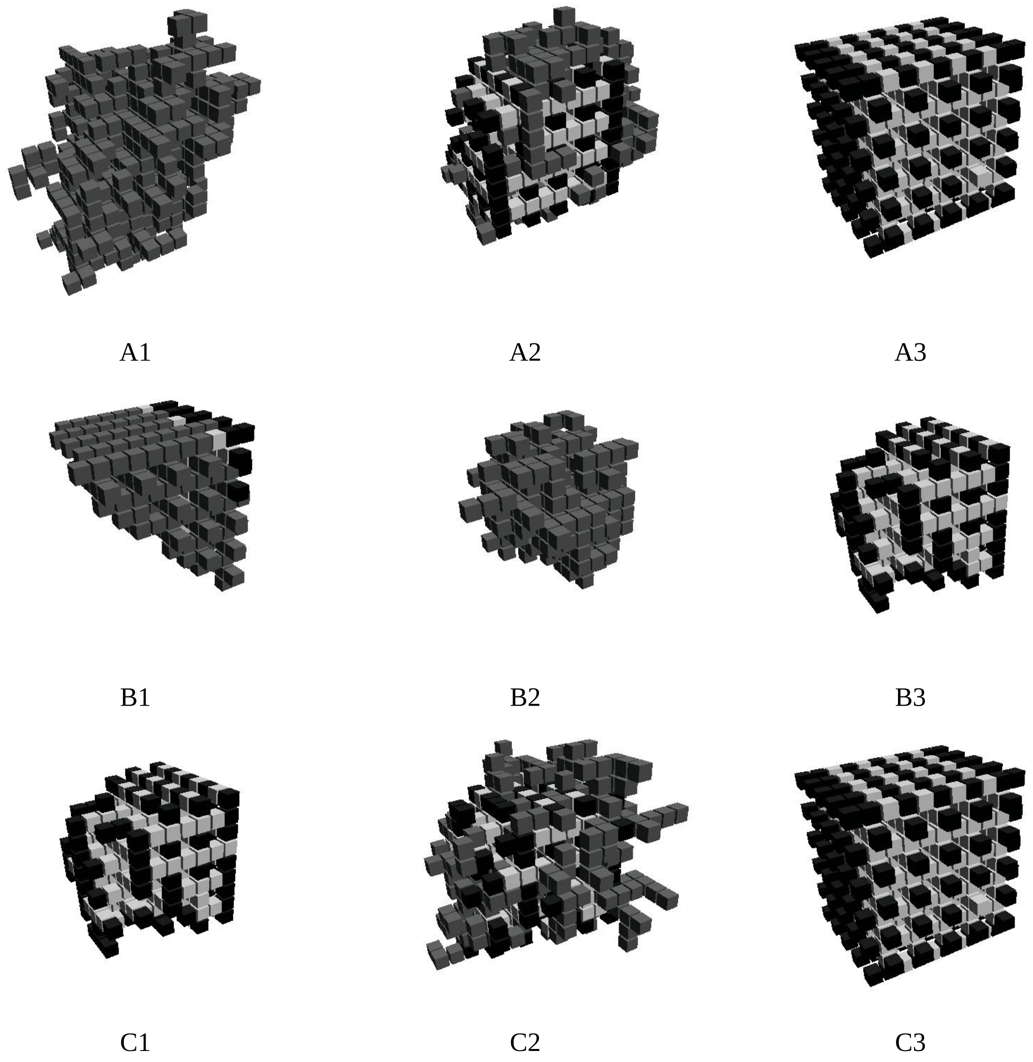


Fig. 1. Reconfiguration and self-repair. **A1**: Modules start in a random configuration. **A2-A3**: The modules self-reconfigure into the desired shape, which is a porous cube. **B1**: Approximately half of the modules are removed from the system. **B2-B3**: The remaining modules automatically reconfigure into a smaller approximation of the desired shape. Note that several modules are left over and remain connected to the structure **C1-C3**: new modules are inserted into the system and again the system adjusts its scale to match the new number of modules. Medium gray modules are wandering modules, black modules are seeds where growth potentially can continue, and light grey modules represent module from which further growth is not possible.

One type of approach is planning based, where a path is determined for each module in the original configuration. Chirikjian and others use this approach and propose heuristic-based methods where the idea is to find a suboptimal sequence of moves from initial to final configuration, which is then optimized by using local searches [3], [4]. Rus et al. simplify the planning problem by using an intermediate chain configuration, which is easy to configure into and out of [5]. Several papers propose hierarchical planners, where at the highest level some of the hardware motion constraints are abstracted away to facilitate efficient planning. Based on the high-level plans, the lower level then produces the detailed sequence of actions [6], [7]. Another approach is to use meta-modules consisting of a small number of modules [6]. By planning at the meta-module level there are no or few motion constraints; on the other hand, meta-modules make the granularity of the robot larger. A related approach is to maintain a uniform scaffolding structure, facilitating planning [8]. Butler implemented the distributed Pacman algorithm on the Crystalline robot, which has very few motion constraints making the planning problem easier [9], [10]. The advantage of the planning approach is that it can accommodate motion constraints and minimize unnecessary moves; the disadvantage is that plans are often comparable in size to the number of modules and depend on knowing the initial configuration.

A different approach is to rely on common local rules as far as possible and then add randomness to deal with the problems that cannot be solved using local rules. This was true in early work such as the work on Fracta [11] and also later work [12], [13]. The problem tends to be that even though the robot often ends up in the desired configuration, it did not always converge. This problem was also present in the work of Yim et al [14], however local communication is used to increase the probability of converging to the final shape. One solution to convergence, proposed by Bojinov et al. [15], is not to focus on a specific configuration. Instead, the idea is to build something with the right functionality. Using this approach it is acceptable if a few modules are stuck as long as the structure maintains its functionality. Alternatively, Jones et al. insist on a specific configuration, but achieve convergence by enforcing a specific sequence of construction [16]. In the work presented here, scaffolding is used to guarantee convergence by making sure that the configurations do not contain local minima, hollow, or solid sub-configurations.

Our system can be thought of as combining two approaches: the global configuration representation is a plan for constructing a shape from simpler shapes (bricks), while the local rules allow modules to recruit nearby modules to form bricks. This approach is similar to approaches for self-assembly used in Amorphous Computing, such as [17], [18]. There a global goal is specified as a construction which is then compiled into biologically-inspired local rules for agents, resulting in self-assembly that is scale-independent, robust and space efficient. The representation we use is inspired by the circle-network proposed by Kondacs for 2D self-assembly, however the agent model and local rules are completely different [19]. Instead we use local rules proposed by Støy [2] to control module movement.

III. Simulated Robot Model

In our simulation, we use modules which are more powerful than any existing hardware platforms but do fall within the definition of a Proteo module put forward by Yim et al. [14]. The modules are cubical and when connected they form a lattice structure. They have six hermaphrodite connectors and can connect to six other modules in the directions: east, west, north, south, up, and down. Modules directly connected to a module are referred to as neighbours. A module can sense whether there are modules in neighbouring lattice cells. In this implementation we do not control the actuator of the connection mechanism, but assume that neighbour modules are connected and disconnected appropriately. A module can only communicate with its neighbours. It is able to rotate around neighbours and to slide along the surface of a layer of modules. Finally, we assume that coordinate systems can be transformed uniquely from one module to another. This is necessary to propagate the gradients and the coordinates used to guide the growth process.

The simulator is programmed in Java3D. The simulation uses discrete time steps. In each time step all the modules are picked in a random sequence and are allowed: 1) to process the messages they have received since last time step, 2) to send messages to neighbours (but not wait for reply), and 3) to move if possible.

IV. From CAD Model to Representation

It is difficult and time consuming to hand-code local rules which result in a desired configuration being assembled. Therefore, we need an automatic way of transforming a human-understandable description of a desired configuration into a representation we can use for control.

In our system, the desired configuration is specified using a connected three-dimensional volume in the VRML 1997 or Wavefront .obj file format, which are industry standards produced by most CAD programs. We transform the model into a representation consisting of a set of overlapping bricks of different sizes which approximates the input shape. The choice to use bricks is fairly arbitrary and other basic geometric shapes, such as spheres or cones, could be used as well. The key features of the representation are: (1) the size scales with the complexity of the three-dimensional model (2) it is independent of the initial configuration (3) it does not require recompilation if the number of modules changes.

The representation is automatically generated as follows: the user specifies a point inside a CAD model. The algorithm then fits as large a brick as possible which contains this point and does not intersect the CAD model. This is done recursively for all points just outside this brick, but inside the CAD model. This process continues until the volume has been filled with overlapping bricks. Figure 2 shows a simple example of a shape and its brick representation. The fewer bricks needed, the more concise the representation.

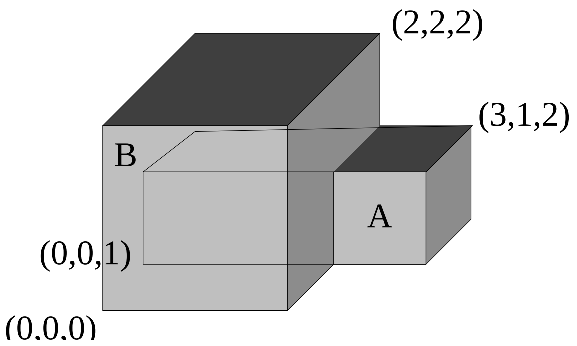
In order to control the resolution of the approximation, a parameter r is supplied. The points and the corners of the bricks are then constrained to be positioned at coordinates equaling an integer times r. Table I shows the number of bricks needed to approximate a model of a Boeing 747 at different resolutions; higher resolutions increase the size of the representation. The resolution r is supplied a priori, and cannot be changed at run-time. However, the number of modules used to approximate the shape can vary at run-time. The shape can be approximated using any number of modules, however in general it takes Mi^3 modules to complete a shape, where M is the minimum number of modules required (i.e. the volume of the brick representation in unit cubes) and i is an integer.

V. From Representation to Self-Reconfiguration Algorithm

Starting from a random configuration the robot needs to reconfigure into the desired configuration as described by the representation. The self-reconfiguration algorithm consists of three components: a coordinate propagation mechanism, a mechanism to create gradients in the system, and a mechanism the modules use to move without disconnecting from the structure. We will look at these in turn.

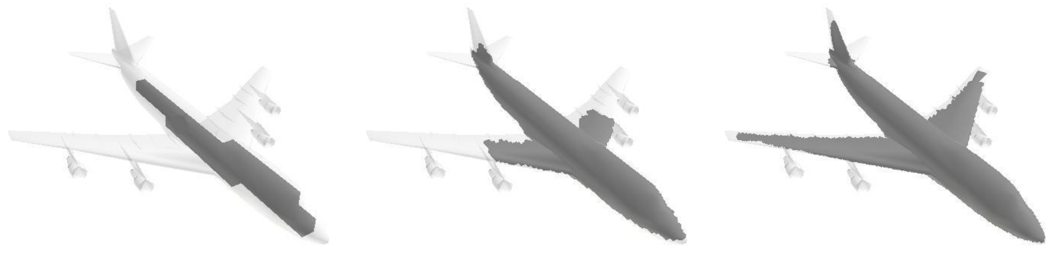
A. Coordinate Propagation

All the modules are initially connected in a random configuration, have a copy of the representation of the desired configuration, and start in the wandering state. An arbitrary module is chosen as the seed and given an arbitrary coordinate. The idea is to grow the configuration from this seed module. The seed detects whether a module is



Brick:	lower-left → upper-right
B:	(0,0,0) → (2,2,2)
A:	(0,0,1) → (3,1,2)

Fig. 2. This figure shows a simple shape approximated by two overlapping bricks, and the associated representation given to the modules. This representation requires a minimum $M = 9$ modules, but can also be fully made with Mi^3 modules, where i is an integer.



Resolution	low	medium	high
Modules	32	4512	34493
Bricks	3	168	1152

TABLE I
THIS TABLE SHOWS THE NUMBER OF MODULES AND BRICKS NEEDED TO APPROXIMATE A CAD MODEL OF A BOING 747 AT THREE DIFFERENT RESOLUTIONS.

needed in a neighbour position based on its coordinate and the representation. If a neighboring module is present, then it is given the appropriate coordinate (the seed's coordinate plus a unit vector in the direction of the neighbour) by the seed. If not, then the seed attracts a wandering module to the unfilled position, using a recruitment gradient. When a module has reached an unfilled position and is given its coordinate, it also may act as a seed. A module stops acting as a seed and becomes finalized when all neighbour modules, specified by the representation and the seed's coordinate, are in place. Notice that if a neighbour disappears (e.g. is removed), then a module will start to attract wandering modules again. Thus there is an inbuilt local self-repair response.

The above system creates a shape at a given scale, as described in [1]. In order to automatically adjust the scale, the local behavior is modified as follows. A seed module scales the representation of the desired shape to contain its own coordinates. Based on this scaled representation and its coordinate, the seed detects whether a neighboring module is needed and attracts a wandering module to the unfilled position with a priority *inversely proportional to the scale* of its representation. As described in the next section, lower scales inhibit higher scales from attracting wandering modules, and thus all lower scale positions get filled first. After all modules in the lowest scale get filled, then the next higher scale becomes attractive to modules. A finalized module or seed may also return to the wandering state if there is an unfilled position which belongs to a lower scale than the module's own position. This allows a system to adjust its scale when modules are removed. Figure 4 shows a simplified version of a module's behavior.

B. Creating a Recruitment Gradient Using Local Communication

In this section we will describe how seed modules attract wandering modules by creating a gradient in the system. A seed module acts as a source and sends out an integer, which represents the concentration of an artificial chemical, to all neighbours. A non-source module calculates the concentration of the artificial chemical at its position by taking the maximum received value and subtracting one. This concentration is then propagated to all neighbours and so on. When the concentration reaches zero it is not further propagated. A module can locate the source by climbing the gradient of concentration. There may be many seeds recruiting simultaneously, in which case the module sees a combined gradient such that moving towards higher concentrations results in movement towards the closest source. The gradient always follows the structure and therefore local minima in the configuration do not exist. For instance, modules will not be trapped at one end of a C-shaped configuration, as shown in Figure 3,

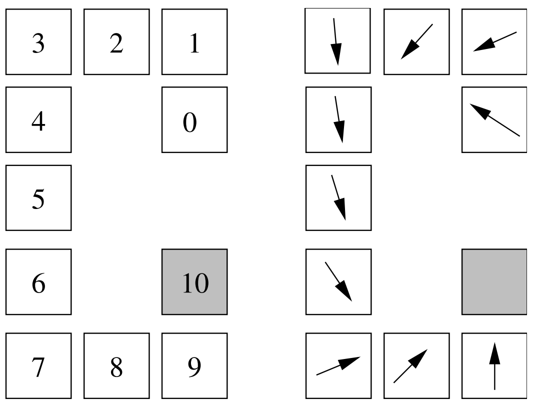


Fig. 3. This figure shows how a gradient is propaged in a simple 2D c-shaped configuration. The boxes represent modules, the numbers the concentration, and the grey box the source module which created the gradient. To the right it can be seen that the gradient follows the structure which means that modules climbing the gradient will not encounter local minima.

because the concentration is not propagated across the gap. Furthermore, the porous scaffold structure ensures that the configuration internally does not contain local minima. Since messages take one time step to travel between neighbours, it can take many time steps for gradients to be propagated in the system. After the unfilled position gets filled, the seed stops sending out the gradient message. Again, it may take many time steps for the gradient to disappear

If wandering modules have to rely on the basic integer based gradient to locate the source, they would have to move around randomly for a while to detect the direction of the gradient. Instead we introduce a vector gradient which makes direction information available locally, thereby eliminating unnecessary moves. The basic gradient implementation is extended with a vector indicating the local direction of the gradient. This vector is updated by taking the vector from the neighbour with the highest concentration, adding a unit vector in the direction of this neighbour and renormalising the result.

In order to achieve scale independent reconfiguration, seeds tag gradients with the scale to which the unfilled position belongs. A gradient inhibits other gradients if its scale tag is lowest. In this way, only the lowest scale gradient is propagated in the structure and its corresponding positions are filled first. Only after all unfilled positions at this scale have been filled, does the gradient disappear. This automatically allows the next higher scale gradient to be propagated in the system. Thus the scale adjustment is achieved purely through distributed and local behavior of the modules. The disadvantage is that a delay is introduced when the system transitions from one scale to the next, because gradients take time to disappear.

C. Staying Connected

Wandering modules climb the vector gradient to reach unfilled positions. Unfortunately, the wandering modules cannot move independently of each other, because they

```
shape = <<brick representation of shape>>;
myscale = 1;
state = WANDERING;
position = <<UNKNOWN>>;

while(true) {
  switch( state ) {
    case WANDERING:
      if ( <<position message received>> ) {
        position = <<message position>>;
        myscale = <<scale that includes position>>;
        state = SEED;
      }
      else {
        <<follow recruitment gradient>>
      }
      break;

    case SEED:
      if ( myscale > <<recruitment gradient scale>> ) {
        position = <<UNKNOWN>>;
        state = WANDERING;
        break;
      }

      foreach <<neighbour position>> {
        if ( <<shape contains neighbor position>> &&
             myscale <= <<recruitment gradient scale>> ) {
          if ( <<neighbor position occupied>> ) {
            <<send position message>>;
          }
          else {
            <<create gradient with myscale>>;
            break;
          }
        }
      }

      if ( <<no neighbours are needed at this scale>> )
        myscale = <<scale that includes more neighbours if possible>>;
      break;
  }
}
```

Fig. 4. Simplified pseudocode for the behavior of a module.

depend on each other for connection to the robot. The problem is to keep the system connected while allowing wandering modules to move. In our solution finalized modules in the configuration emit a *connection gradient* and wandering modules only move or change to the wandering state if they do not perturb the gradient. Detailed rules for movement and proofs were presented in [2].

VI. Experiments

In this section we present preliminary results comparing two strategies for achieving the desired configuration. In the first approach the desired configuration is grown directly at a scale which matches the number of modules. This scenario simulates a situation where the number of modules is known a priori. In the second approach the configuration is grown at increasingly higher scales until all modules have been used. We compare the two approaches based on three criteria: time taken to reconfigure, number of moves, and number of messages.

The task is to self-reconfigure from a random connected configuration of 605 modules to a configuration in the shape of a cube. The representation of the configuration is built by the generator based on a CAD model. The representation is then downloaded into the modules of the simulation and the self-reconfiguration process is started.

In Table II, we can see that to grow the configuration using the scale independent approach requires significantly more resources than growing it directly. For this particular shape, approximately three times as many moves and time steps are used, and approximately five times as many messages. This is not surprising since a time delay is introduced when the system makes a transition between scales. The

	Scale independent	Scale dependent
Moves	29270±5300	9761±1455
Time steps	817±87	302±46
Messages	2801907±395257	616922±73984

TABLE II

THE PERFORMANCE OF THE SCALE DEPENDENT AND SCALE INDEPENDENT APPROACHES (MEAN AND STD.DEV. OF 10 TRIALS SHOWN).

delay occurs because the gradient corresponding to the old scale has to disappear before the gradient corresponding to the new scale starts to be propagated. During this delay wandering modules move unnecessarily, contributing to the higher cost in moves. In the case of the cube, the shape with scaffold may be created completely at 8 different scales (with 1, 9, 75, 147, 405, and 605 modules respectively), however smaller scales are completed relatively quickly compared with the higher scales. In the future we plan to further investigate empirically, and analytically, the cost increase as a function of number of scales and type of shapes.

VII. Conclusion

We have explored an approach to the control of self-reconfiguration which consists of two steps. In the first step a generator takes as input a 3D CAD model of a desired configuration and outputs a set of overlapping bricks which represent this configuration. In the second step this representation is combined with a local, distributed control algorithm to produce the final self-reconfiguration algorithm. This algorithm controls the self-reconfiguration process through a growth process: seed modules create recruitment gradients in the configuration that wandering modules climb to locate the seed.

In this paper we demonstrate that this approach also can be used to self-reconfigure into a desired configuration even if the number of modules is unknown a priori. Furthermore, modules can, at run-time, be added or removed from the system and the configuration will automatically readjust its size. We have in experiments demonstrated that the price of this self-repair capability in terms of moves, time steps, and communication messages is high. However, the proposed system represents one of the first steps toward scale independent self-reconfiguration.

Acknowledgments

This research is in part funded by the EU contract IST-2001-33060, the Danish Technical Research Council contract 26-01-0088, and NSF grant EIA-0130391.

References

[1] K. Støy and R. Nagpal, “Self-reconfiguration using directed growth,” in *Proc., int. conf. on distributed autonomous robot systems (DARS-04) (to appear)*, Toulouse, France, 2004.

[2] K. Støy, “Controlling self-reconfiguration using cellular automata and gradients,” in *Proc., 8th int. conf. on intelligent autonomous systems (IAS-8)*, Amsterdam, The Netherlands, 2004, pp. 693–702.

[3] G. Chirikjian, A. Pamecha, and I. Ebert-Uphoff, “Evaluating efficiency of self-reconfiguration in a class of modular robots,” *Robotics Systems*, vol. 13, pp. 317–338, 1996.

[4] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian, “Useful metrics for modular robot motion planning,” *IEEE Transactions on Robotics and Automation*, vol. 13, no. 4, pp. 531–545, 1997.

[5] D. Rus and M. Vona, “Self-reconfiguration planning with compressible unit modules,” in *Proc., IEEE Int. Conf. on Robotics and Automation (ICRA’99)*, vol. 4, Detroit, Michigan, USA, 1999, pp. 2513–2530.

[6] K. Kotay and D. Rus, “Algorithms for self-reconfiguring molecule motion planning,” in *Proc., IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS’00*, vol. 3, Maui, Hawaii, USA, 2000, pp. 2184–2193.

[7] C. Ünsal, H. Kiliccote, and P. Khosla, “A modular self-reconfigurable bipartite robotic system: Implementation and motion planning,” *Autonomous Robots*, vol. 10, no. 1, pp. 23–40, 2001.

[8] C. Ünsal and P. Khosla, “A multi-layered planner for self-reconfiguration of a uniform group of i-cube modules,” in *Proc., IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS’01)*, vol. 1, Maui, Hawaii, USA, 2001, pp. 598–605.

[9] Z. Butler, S. Byrnes, and D. Rus, “Distributed motion planning for modular robots with unit-compressible modules,” in *Proc., IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS’01)*, vol. 2, Maui, Hawaii, USA, 2001, pp. 790–796.

[10] S. Vassilvitskii, M. Yim, and J. Suh, “A complete, local and parallel reconfiguration algorithm for cube style modular robots,” in *Proc., IEEE Int. Conf. on Robotics and Automation (ICRA’02)*, vol. 1, Washington, DC, USA, 2002, pp. 117–122.

[11] S. Murata, H. Kurokawa, and S. Kokaji, “Self-assembling machine,” in *Proc., IEEE Int. Conf. on Robotics & Automation (ICRA’94)*, San Diego, California, USA, 1994, pp. 441–448.

[12] E. Yoshida, S. Murata, H. Kurokawa, K. Tomita, and S. Kokaji, “A distributed reconfiguration method for 3-d homogeneous structure,” in *Proc., IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS’98)*, vol. 2, Victoria, B.C., Canada, 1998, pp. 852–859.

[13] K. Tomita, S. Murata, H. Kurokawa, E. Yoshida, and S. Kokaji, “A self-assembly and self-repair method for a distributed mechanical system,” *IEEE Transactions on Robotics and Automation*, vol. 15, no. 6, pp. 1035–1045, Dec 1999.

[14] M. Yim, Y. Zhang, J. Lamping, and E. Mao, “Distributed control for 3d metamorphosis,” *Autonomous Robots*, vol. 10, no. 1, pp. 41–56, 2001.

[15] H. Bojinov, A. Casal, and T. Hogg, “Emergent structures in modular self-reconfigurable robots,” in *Proc., IEEE Int. Conf. on Robotics & Automation (ICRA’00)*, vol. 2, San Francisco, California, USA, 2000, pp. 1734 –1741.

[16] C. Jones and M. J. Matarić, “From local to global behavior in intelligent self-assembly,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA’03)*, Taipei, Taiwan, 2003, pp. 721–726.

[17] R. Nagpal, “Programmable self-assembly using biologically-inspired multiagent control,” in *Proc., 1st Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Bologna, Italy, 2002, pp. 418–425.

[18] R. Nagpal, A. Kondacs, and C. Chang, “Programming methodology for biologically-inspired self-assembling systems,” in *Proc., AAAI Spring Symposium on Computational Synthesis: From Basic Building Blocks to High Level Functionality*, 2003.

[19] A. Kondacs, “Biologically-inspired Self-Assembly of 2D Shapes, Using Global-to-local Compilation”, in *Proc., Int. joint conf. on Artificial Intelligence (IJCAI-03)*, 2003.