

The Crystal Robot: Implementation and Demonstration

Robert Fitch and Zack Butler and Daniela Rus

Department of Computer Science
Dartmouth College
Hanover, NH 03755
{rfitch,zackb,rus}@cs.dartmouth.edu

Abstract

Effective algorithms for modular self-reconfiguring robots should be distributed and parallel. In previous work, we explored general algorithms for locomotion and self-replication and explained their instantiations to systems in which modules move over the surface of the robot. In this paper, we describe the instantiation of one such locomotion algorithm to the Crystal robot: a distributed locomotion algorithm designed specifically for unit-compressible actuation. We also present the integration of this algorithm with obstacle avoidance for a demonstration at the 2002 AAAI conference.

Introduction

Self-reconfiguring (SR) robots are robots that can change shape to match the task at hand. These robots comprise many discrete modules, often all identical, with simple functionality such as connecting to neighbors, limited actuation, computation, communication and power. Orchestrating the behavior of the individual modules allows the robot to approximate, and reconfigure between, arbitrary shapes. This shape-changing ability allows SR robots to respond to unpredictable environments better than fixed architecture robots. Common examples of reconfigurability in action include transforming between snake shapes for moving through holes and legged locomotion for traversing rough terrain, and using reconfiguration for locomotion as shown in Figure 1.

Development of functional self-reconfiguring robots is a significant challenge. Hardware must be designed and built that is capable of self-reconfiguration and autonomous operation, and supporting algorithms must be developed that can confer upon the hardware the ability to change shape and to locomote. In particular, we are interested in systems (hardware and software) that operate without a central controller or common communication channel. In this paper, we describe a design for a self-reconfiguring robot called the *Crystal*, along with distributed algorithms for locomotion (both in the abstract and instantiated to our system) and associated experiments that demonstrate the basic operation of both the robot and the algorithms.

Several types of modules and actuation mechanisms that can support self-reconfiguration have been proposed (Fukuda & Kawakuchi 1990; Pamecha *et al.* 1996; Hosokawa *et al.* 1998; Shen, Will, & Castano 1999; Murata *et al.* 2000; Ünsal & Khosla 2000). In our previous work we described a module capable of 3D self-reconfiguration by using a rotation-based actuation called the *robotic molecule* (Kotay & Rus 1999) and a module capable of self-reconfiguration that uses scaling-based actuation, the Crystal (Rus & Vona 2001). The Crystal robot has gone through two design phases. The original design was described in (Rus & Vona 2001). Based on our experience with this module we refined the design to add an additional degree of freedom, sensing, and distributed systems support through point to point communication.

The key algorithmic question for self-reconfigurable robots is the planning and control problem: how should the modules move relative to each other in order to achieve a static or dynamic goal shape, or to move in a desired way, and how to do it efficiently. We have already developed several centralized planners for self-reconfiguring robots (Kotay & Rus 1999; Rus & Vona 2001). Some of the most interesting applications of this work will employ thousands of modules working together. The off-line planning algorithms proposed above move one module at a time and may be too slow and impractical for controlling lattices made of thousands of modules. In this paper we discuss distributed control algorithms that are scalable, support parallelism, and are better suited for operation in unstructured environments.

Distributed algorithms are naturally suited for controlling self-reconfiguring robots because they take advantage of modularity, allowing the system to be more robust to failures of individual modules and communications, and supporting partitioning of the robot. Several distributed algorithms for self-reconfiguring robots have been proposed, including locomotion for string-type robots (Støy, Shen, & Will 2002) as well as reconfiguration of 2D and 3D robots. Notable 2D examples are algorithms for the Fracta system (Tomita *et al.* 1999) and the system of Hosokawa *et al.* (Hosokawa *et al.* 1998), and the PacMan algorithm for unit-compressible systems (Butler, Byrnes, & Rus 2001). Algorithms for 3D systems include work on the Proteo system (Yim *et al.* 2001) as well as for meta-modules made from unit-compressible modules (Vassilvitskii, Yim, & Suh 2002). Many of these

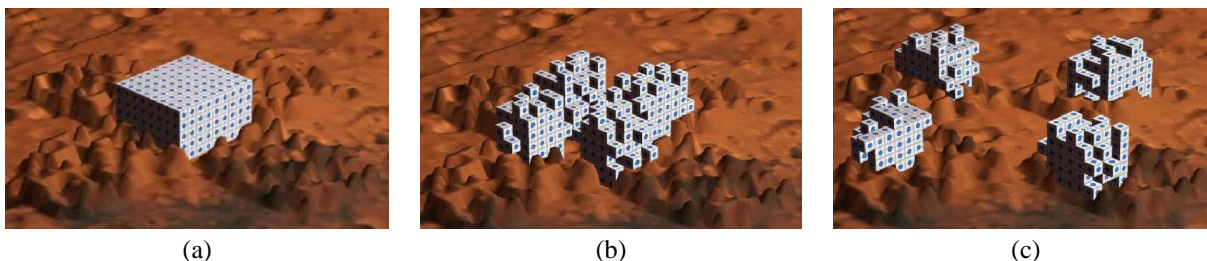


Figure 1: Simulation of SR robot on Mars terrain. The robot begins as a single cube in (a), but divides into four in (b) and (c) for parallel exploration using a distributed, waterfall-like locomotion algorithm inspired by cellular-automata.

works have also included hardware implementation.

In our recent work (Butler *et al.* 2002; Butler, Murata, & Rus 2002) we proposed distributed algorithms for several tasks: (1) *locomotion*, where the modular robot can implement a tumbling gait that conforms to the terrain geometry; (2) *self-replication*, where the robot can divide itself into smaller autonomous robots, for example to explore the terrain in parallel, and (3) *merging*, where two autonomous modular robots can connect into a larger robot, for example in order to climb taller obstacles. These algorithms use local information only and are inspired by a cellular automata approach. For each task we designed a set of geometric rules, so that each module tests the same rules with respect to its neighborhood to decide what action to take. The resulting control algorithms are distributed, efficient, and provably correct (Butler *et al.* 2002; Butler, Murata, & Rus 2002). These algorithms are also generic in the sense that they apply to an abstract model for self-reconfiguring robots, where individual modules on the robot have the ability to traverse a planar surface composed of identical modules and to make convex and concave transitions between surfaces. Many existing self-reconfiguring robots (Hosokawa *et al.* 1998; Kotay & Rus 1999; Murata *et al.* 2000; Pamecha *et al.* 1996; Ünsal & Khosla 2000) fit this model.

In this paper we start from the algorithms developed in (Butler *et al.* 2002; Butler, Murata, & Rus 2002) and show how we instantiated them to unit-compressible systems such as the Crystal robot. This instantiation is challenging because the actuation mechanism of unit-compressible robots more naturally supports movement through the volume of the robot rather than on its surface. We discuss in detail the control algorithms for distributed robot locomotion; each module can sense its local neighborhood structure, communicate with its neighbors, and perform some simple computations to evaluate the control rules. We also discuss an implementation of these algorithms on the Crystal robot and a task recently demonstrated at the 2002 SIGGRAPH and AAAI conferences where a five-unit Crystal robot executed our locomotion algorithm while sensing obstacles.

Crystal Robot

The Crystal Robot is a unit-compressible self-reconfiguring modular robot. It actuates by expansion and contraction of individual modules, which together with connection and disconnection allows the robot to change shape as well as lo-

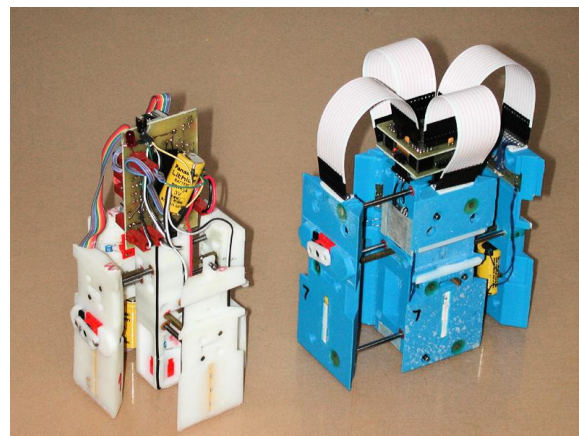


Figure 2: The first (left) and second (right) version prototypes of the Crystal robot. The first version is fully expanded while the second is contracted along one axis and expanded along the other.

comote. Each module consists of a central core and four faces that move in and out relative to the core to perform expansion. An expanded module is exactly twice the size of a compressed module, which aids in reconfiguration and planning.

The original version of the Crystal (Rus & Vona 2001) had a single degree of freedom for expansion, so that all four faces expanded and contracted together. Each module had its own processor to control actuation, but synchronization was performed through sensing an external beacon — modules had no ability to communicate directly with each other. In the new version of the hardware, the expansion has two degrees of freedom as well as inter-module communications. In fact, in the current version, there is no facility for global communications, so all operations must be performed in a distributed fashion. The old and new prototypes are pictured together for comparison in Fig. 2. In this section, we describe in detail the hardware components, electronics and fabrication of the new modules, the communications infrastructure developed, and our method of initializing the robot's software state.

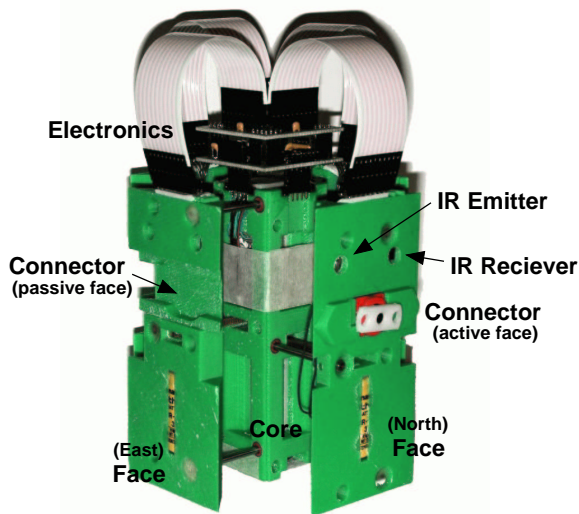


Figure 3: A single module of the Crystal robot.

Hardware

The second-generation Crystal module, shown in Fig. 3, incorporates several important new features including an additional degree of freedom for actuation, inter-module IR communication capability and sensor inputs. A central control beacon is no longer present to coordinate behavior. Thus, the robot control is now done in a purely distributed fashion. The North/South and East/West faces are independently actuated, so the degrees of freedom increase to four (two for expansion/contraction, and two for connectors on active faces). In addition, several features have been improved over the first version, including stiffening of the linear bearings that align the faces during actuation, more powerful motors to perform actuation, and a faster processor with more memory and I/O capability.

Each atom's on-board electronics provide computation, IR communication, sensor inputs and motor control. The processor is a Hitachi HD64F3644H running at 10 MHz, which includes 32KB of EEPROM for program storage. Analog inputs to this chip are brought out to a small connector so that various sensors (analog or digital in nature) can be attached. Digital outputs control motor drivers that perform actuation, and an additional digital output powers an LED for rudimentary debugging.

Communication is implemented with asynchronous serial over IR components on the Crystal faces. Each module face contains an IR emitter and detector that allow modules to communicate at distances of up to 10 cm. These components are connected to a dedicated Maxim Max3100 UART in the core, so a unit can talk with all neighboring units essentially simultaneously. The UARTs communicate at 1200 Baud and have an eight-word hardware FIFO. Synchronous serial communication is used between the processor and the UARTs.

Each face is connected to the core circuit board with a

flexible ribbon cable, so that the cabling exerts minimal force on the face during expansion and contraction. Four 3V Lithium batteries (one in each face) power the unit and enable fully untethered operation. Code is downloaded to the processor through a serial interface, then executes as soon as the unit is powered on.

The expansion/contraction mechanism uses a rack-and-pinion to actuate each pair of module faces. Two MicroMo motors are mounted coaxially in the module core, with pinion gears mounted directly on the motor output shafts. Racks connected to opposing faces mate on opposite sides of a pinion such that each motor drives two faces simultaneously. Shaft encoders built into the MicroMo's housing generate interrupts that allow the processor to detect when the face is fully expanded or contracted.

Modules attach to each other at their faces, using channel-and-key type connectors. Each module has two faces with active connectors, and two faces with passive connectors. Passive faces simply contain a channel that accepts a bar from an active face. The active face can rotate the bar a quarter-turn, locking the two modules together, and unlock the modules by reversing the rotation. Lego mini-motors are used to actuate the active faces.

Dimensionally, this Crystal prototype is slightly larger than its predecessor. Expanded size is 5.2 inches square, and contracted size is 2.6 inches square. Overall height is 7.4 inches with a weight of 18 ounces. Eighteen modules have been constructed so far.

Software Architecture

Communication is the key component for providing the system support for distributed control in Crystal robots. To this end, we developed a message passing infrastructure on top of the Crystal's communication capabilities. Each unit maintains a message queue, and can post messages to neighbor modules. A module's program is then centered around a message loop, similar to the message loop in modern windowing systems. In each iteration, the processor polls each UART for incoming messages and adds any new messages to the queue. It then takes a message from the queue and processes it according to the appropriate message handler. Since each UART has its own FIFO, the UARTs still can receive data while the processor is busy handling messages. Because the processor speed is much faster than the UART transmission rate, the risk of the UART FIFOs filling up before they get serviced by the processor has not been an issue.

Library functions were developed to handle the synchronous communication between the processor and UARTs in both directions. Polling of the UARTs is done with a single library call, so that the creation of the message loop is trivial. For these library functions, we have assumed that all messages will be two bytes long (although the content within the two bytes is message dependent). This limit was imposed to allow the system to be able to receive four messages from each direction before the UARTs are polled, although future messaging infrastructure (and interrupt-based communication) will allow us to relax this restriction. For all the algorithms and robot sizes presented here, this message size limitation has not been problematic.

sender	data				parent ID	type			
15	13				7			3	0
0	1	1	1	0	0	1	1	0	0

Figure 4: Message format example.

Using this infrastructure we implemented the distributed algorithms by defining a set of message types and creating message handlers for each message type. In general we have maintained a common format for the messages, so that additional common library functions can be used. In particular, the lowest four bits of the message are reserved for the message type, and the highest two bits are reserved for the direction from which the message was received. These data can be extracted from the message with a bit mask, and providing library functions to perform the bit masking leads to less propensity for coding errors. The data can take up the remainder of the message and be in whatever format is necessary for the particular message or algorithm (in many of our algorithms, the data format is consistent over all message types). An example message is given in Fig. 4.

Initialization

While the messages used by any given algorithm are necessarily specific to that algorithm, there is a common boot sequence that is used (with some algorithm-specific adaptations possible). A boot sequence is required since the modules must be manually switched on one at a time, so that when a module starts its program, it does not initially know whether its neighbor units are powered on. To solve this problem, we use a special message called *system_init*. This message is initially generated only after all modules have been powered on, and can be sent at later times to effect a software “reboot.” This message is created by one module that has a switch. The *system_init* message handler propagates the message to all neighbors, and also recognizes each neighbor from which it received a *system_init*. Any initialization of algorithm-specific global variables is also done in this function. If further *system_inits* are received, they are ignored. However, to enable later soft reboots, there is also a *pre_init* message to set a state bit that is cleared by *system_init*. This allows a module to realize when a *system_init* (or *pre_init*) is a duplicate and when it indicates a new reboot of the system.

Demonstration

The development of the new Crystal hardware and communication infrastructure has enabled us to implement and test several distributed algorithms for this system. In fact, since there is no central communication or control capability, all testing of the system must be done in a distributed fashion. This allows us to test the validity of our distributed algorithms in a truly asynchronous context, in which modules are powered up at different times, have different actuation speeds, and have no way to share data except via the pass-

ing of messages (all issues that are easy to forget about in simulation).

In preparing a demonstration for the 2002 AAAI conference, we wished to choose a task that illustrated the basic capabilities of the Crystal robot and the style of algorithms we have developed. The task also needed to be one that could be run reliably, as the demonstration would run for hours at a time, and had to operate for a reasonable period without human intervention. The task we chose was for the robot to locomote on a table until it detected an obstacle, then reverse direction and repeat. This task is simple, yet presents three challenges: a locomotion algorithm, obstacle detection, and a reliable implementation. We addressed these challenges by adding simple contact sensors to the robot, and modifying our inchworm locomotion algorithm to make use of sensory data in controlling direction. In this section we describe the algorithm, the implementation, and observed results.

Algorithm 1 Distributed Stand-Alone locomotion.

State:
neighbors[], array of neighbors
heading, direction robot is moving: N,S,E,W

Messages:
inch (*direction d*), sent to move robot in direction *d*
 Action: set *heading* state to *d*, execute TryRules()
state (*state s*), announces state changes to neighbors
 Action: execute TryRules()

Procedures:
 TryRules()
 position ← FindPosition()
 if position = head **then**
 if neighbor[opposite(*heading*)] is contracted **then**
 contract, send state
 expand, poll touch sensor, send state
 if touch sensor is triggered **then**
 send inch in reverse direction
 else
 send inch
 if position = body **then**
 if neighbors[opposite(*heading*)] is contracted **then**
 contract, send state
 expand, send state
 if position = tail and responding to *inch* message **then**
 contract, send state
 expand

FindPosition()
 if rear neighbor but no forward neighbor **then**
 return head
 else if forward neighbor but no rear neighbor **then**
 return tail
 else
 return body

Inchworm Locomotion Algorithm

One of the fundamental tasks of any self-reconfigurable robot system is locomotion. On most lattice-based systems, locomotion can be performed by having individual modules

move over the surface of the group from the back to the front in a tank-tread-like pattern. In unit-compressible systems such as the Crystal, however, no single module can move relative to the group without help from other modules, and so a different specialized technique is required. In previous work, we have described various locomotion techniques for the Crystal that operate in a centralized fashion to coordinate the modules. Here we present distributed algorithms which achieve similar performance with only local information available to each module.

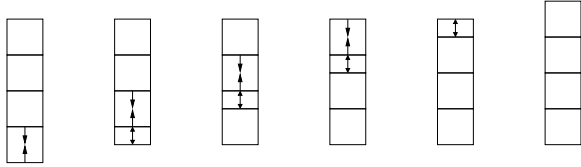


Figure 5: Schematic of module action under Stand-Alone locomotion, in which the group is heading upward, and the series (left to right) represents progress of a single inchworm “step”.

To perform locomotion, we have developed several algorithms for unit-compressible systems. One such algorithm produces inchworm-like motion to perform locomotion on a stand-alone group of modules, taking advantage of friction with the ground to move the group forward. It is based on a set of rules that test the module’s relative geometry and generate expansions and contractions as well as messages that modules send to their neighbors. When a module receives a message from a neighbor indicating a change of state, it tests the neighborhood against all the rules, and if any rule applies, executes the commands associated with the rule. The algorithm is designed to mimic inchworm-like locomotion: compressions are created and propagated from the back of the group to the front, producing overall motion.

The Stand-Alone algorithm is presented as Algorithm 1. In our algorithm listings, we give a module’s global state variables, the message types it can send and receive, and the procedures that are called from the message handlers (including the rules of the algorithm). The overall idea behind Stand-Alone locomotion is that at any given moment, the majority of the modules are stationary, so that the remaining modules will move relative to the majority. In addition, the motion is specified such that two adjacent modules will move together, minimizing the net force to the other modules. A schematic storyboard of this algorithm is given in Fig. 5. The “tail” module contracts first and signals its forward neighbor to contract. Each module expands after contraction, so that the contraction propagates through the robot. When the contraction has reached the front of the group, the group will have moved half a unit forward. Depending on context, once the leader of the group has contracted and expanded, it can then send a message back to the tail to initiate another step. In our demonstration, the head module polls its touch sensor as it expands. If it senses an obstacle, it stops the expansion and sends a message to initiate the algorithm with the direction reversed.

Analysis/Extensions The Stand-Alone locomotion algorithms can be proven correct (i.e. it produces locomotion in the intended direction). This is done by noting that only the tail can contract at first, followed by each other module in turn. Since each contraction must be triggered by a *state* message, no module will contract until it has the proper information, but once it does contract and sends a message forward to that effect, the contraction will always propagate.

Algorithm 1 is specified (and analyzed above) for a single column, but can be extended to convex shapes by selecting one column as a master column. When a module in the master column actuates, a message is passed across its row that causes all modules in the row to actuate simultaneously. This is effective since communication is much faster than actuation, and the modules not in the master column have no other responsibilities that could cause communications lag. This allows for correct locomotion for any convex shape, as shown in the experiments below.

Implementation

The ability to perform distributed locomotion depends on the modules passing their state to their immediate neighbors. The communication infrastructure enables this by allowing us to define a *state* message, which indicates whether the module is expanded in the direction of motion and (for the Attaching algorithm) whether it is connected to a fixed module. This information (along with the message type) easily fits within the two-byte limit. The other message required is the *inch* message, which tells the robot to begin locomotion, and includes the desired direction of travel. The *inch* message is sent from an external source to initiate the locomotion, and is also sent by the head module to trigger another step (when desired). Together with the soft-boot sequence described previously, these algorithms therefore use only four message types to perform locomotion.

Results

We implemented this algorithm and performed experiments with various shapes, one of which is shown in Figure 6. The experiments successfully demonstrated reliable locomotion in the configurations we tested. See Butler, Fitch and Rus (Butler, Fitch, & Rus 2002) for further discussion.

This locomotion gait is significant first in that it exemplifies the style of distributed, scalable algorithms we wish to develop and implement in proposed work. It also provides on possible gait to use in an application that chooses between various gaits, such as wheeled locomotion versus the inchworm, in response to the environment. For demonstrations at the SIGGRAPH and AAAI conferences, the touch sensors we added to the modules allowed the head module to detect obstacles and reverse the direction of the inchworm. The result was that the robot “walked” back and forth between two obstacles on a table. The algorithm ran for over 65 hours in total at the SIGGRAPH and AAAI conferences.

Conclusion

These experiments were the first significant repeated demonstrations of untethered unit-compressible modular robots in

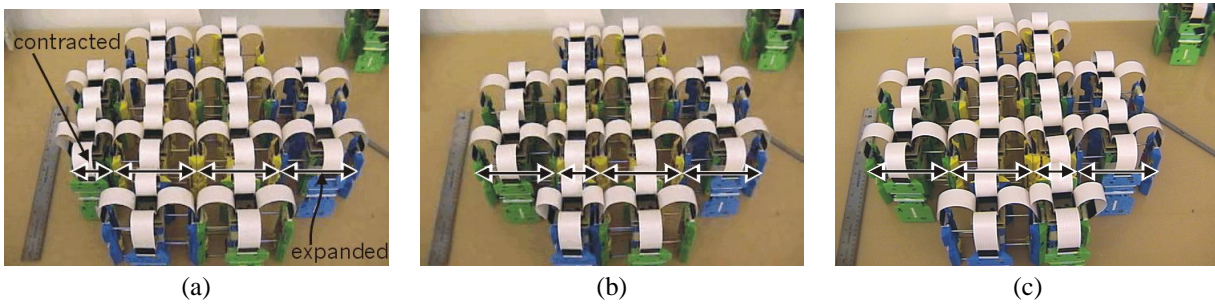


Figure 6: Photos of locomotion experiment for a “blob” shape. In (a), the leftmost column is contracted, and in (b) and (c) the following columns contract to make the group walk to the right.

public. They were largely successful, in that we were able to successfully perform distributed communication and actuation to achieve specific tasks without any central control using on-board processing and power supply. During the course of the experimentation, we learned several things which we plan to incorporate into future versions of the system. In particular, the connectors are not rigid enough to support some types of reconfigurations and often jam during initialization, the power consumption could be greatly reduced by utilizing the processor’s “sleep” modes when possible, and a more general communications protocol would allow for greater algorithmic capability and robustness against ambient IR. Cameras, camcorders, and other photographic equipment emit enough IR to trigger the communications hardware. This creates false messages that confuse the robot.

The connector issues are more challenging. In the first prototype of the Crystal, the faces themselves were not particularly rigid, and so compliance in the connectors was not noticed in the overall performance. However, the new Crystal uses better bushings to keep the faces more rigidly aligned during expansion. This means that alignment tolerances are now limited by the connector stiffness. The lock-and-key style of connector is advantageous in that it can fit in a very small space (the faces of the Crystal are only 8 mm thick), allows two module faces to slide along each other when not connected, and uses very simple parts (the key can be directly driven by a gearmotor). However, it does not have much tolerance for misalignment (≈ 3 mm laterally and ≈ 1 mm of separation, and virtually no vertical misalignment). In addition, in order to decrease the potential for jamming, the keys must be slightly undersized relative to the slot, but this allows for them to wobble. We are currently exploring alternative connector designs.

Acknowledgements

This paper describes research done in the Dartmouth Robotics Laboratory. Support for this work was provided through the NSF CAREER award IRI-9624286, NSF award IRI-9714332, NSF award EIA-9901589, NSF award IIS-98-18299, and NSF IIS 9912193.

References

Butler, Z.; Kotay, K.; Rus, D.; and Tomita, K. 2002. Generic decentralized control for a class of self-reconfigurable robots. In

Proc of IEEE ICRA.

Butler, Z.; Byrnes, S.; and Rus, D. 2001. Distributed motion planning for modular robots with unit-compressible modules. In *Proc. of the Int’l Conf. on Intelligent Robots and Systems*.

Butler, Z.; Fitch, R.; and Rus, D. 2002. Experiments in distributed control for modular robots. In *Proc. of the Int’l Conf. on Intelligent Robots and Systems, to appear*.

Butler, Z.; Murata, S.; and Rus, D. 2002. Distributed replication algorithms for self-reconfiguring modular robots. In *Distributed Autonomous Robotic Systems 5*.

Fukuda, T., and Kawakuchi, Y. 1990. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. In *Proc. of IEEE ICRA*, 662–7.

Hosokawa, K.; Tsujimori, T.; Fujii, T.; Kaetsu, H.; Asama, H.; Koruda, Y.; and Endo, I. 1998. Self-organizing collective robots with morphogenesis in a vertical plane. In *Proc. of IEEE ICRA*, 2858–63.

Kotay, K., and Rus, D. 1999. Locomotion versatility through self-reconfiguration. *Robotics and Autonomous Systems* 26:217–32.

Murata, S.; Yoshida, E.; Tomita, K.; Kurokawa, H.; Kamimura, A.; and Kokaji, S. 2000. Hardware design of modular robotic system. In *Proc. of the Int’l Conf. on Intelligent Robots and Systems*, 2210–7.

Pamecha, A.; Chiang, C.-J.; Stein, D.; and Chirikjian, G. 1996. Design and implementation of metamorphic robots. In *Proc. of the 1996 ASME Design Engineering Technical Conf. and Computers in Engineering Conf.*

Rus, D., and Vona, M. 2001. Crystalline robots: Self-reconfiguration with unit-compressible modules. *Autonomous Robots* 10(1):107–24.

Shen, W.-M.; Will, P.; and Castano, A. 1999. Robot modularity for self-reconfiguration. In *SPIE Conf. on Sensor Fusion and Decentralized Control in Robotic Systems 2*.

Støy, K.; Shen, W.-M.; and Will, P. 2002. Global locomotion from local interaction in self-reconfigurable robots. In *Proc. of IAS-7*.

Tomita, K.; Murata, S.; Kurokawa, H.; Yoshida, E.; and Kokaji, S. 1999. Self-assembly and self-repair method for a distributed mechanical system. *IEEE Trans. on Robotics and Automation* 15(6):1035–45.

Ünsal, C., and Khosla, P. 2000. Mechatronic design of a modular self-reconfiguring robotic system. In *Proc. of IEEE ICRA*, 1742–7.

Vassilvitskii, S.; Yim, M.; and Suh, J. 2002. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of IEEE ICRA*.

Yim, M.; Zhang, Y.; Lamping, J.; and Mao, E. 2001. Distributed control for 3D shape metamorphosis. *Autonomous Robots* 10(1):41–56.