Crystalline Robots: 
Self-reconfiguration with Compressible Unit Modules 

Daniela Rus  Marsette Vona 
Department of Computer Science 
Dartmouth College 
Hanover, NH 03755 
rus@cs.dartmouth.edu, vona@ai.mit.edu 

Abstract 
We discuss a robotic system composed of Crystalline modules. Crystalline modules can aggregate together to form distributed robot systems. Crystalline modules can move relative to each other by expanding and contracting. This actuation mechanism permits automated shape metamorphosis. We describe the Crystalline module concept and show the basic motions that enable a Crystalline robot system to self-reconfigure. We present an algorithm for general self-reconfiguration and describe simulation experiments. 

1 Introduction 
We wish to develop versatile and extensible massively-parallel distributed robotic systems. We believe that versatility can be obtained by developing self-reconfigurable systems. Self-reconfiguring robots have the ability to adapt to the operating environment and the required functionality by changing shape. They consist of a set of identical robotic modules that can autonomously and dynamically change their aggregate geometric structure to suit different locomotion, manipulation, and sensing tasks. For example, a self-reconfiguring robot system could self-organize as a snake shape to pass through a narrow tunnel and reorganize as a multi-legged walker upon exit to traverse rough terrain. 

Self-reconfiguring robots can be viewed as a minimalist approach to designing versatile and extensible robots with multiple modalities of locomotion, manipulation, and sensing. A single, architecturally-lean module can aggregate in a variety of shapes with other identical modules. A primary design goal for a self-reconfiguring robot is to allow the robot to assume any geometric shape. This goal is different from that of other types of shape-changing robots, which may only take one of a small number of shapes. For example, Figure 1 illustrates a self-reconfiguring robot that can take an amoebic shape, a square-shape, a chair shape, an “E” shape, etc. 

Self-reconfiguring robots are suited for a range of applications that require the geometric modification of a part and are characterized by incomplete a priori task knowledge. Such a
robot could match its geometric structure to the shape of the surrounding terrain for versatile locomotion. This can be achieved by requiring the robot to metamorphose from one shape to another to best match the shape of the terrain in a statically stable gait, as illustrated in Figure 2. A novel useful application is to realize general self-repair. If an arbitrary part of a fixed-architecture robot fails, the robot can not usually repair itself; a human or a different robot must perform the task. A self-reconfiguring robot carrying some additional modules may excise the failed part and replace it with spare units. Another application domain is in the area of visualization. It is now common to use software to visualize three-dimensional data. A self-reconfiguring robot could serve as a physical prototype for such virtual models. Finally, we envision small scale modules as the basis for smart self-adjusting furniture.

Figure 1: This Figure shows the general design goal for self-reconfiguring robots to realize arbitrary shapes. In this figure, the amoebic shape on the left is transformed into a cubic shape, into a chair, and then into a rotated “E” shape, suggesting that the range of possible shapes is fully general.

Figure 2: This figure demonstrates using shape metamorphosis for locomotion. A statically stable gait is used to translate the robot from left to right.

To create flexible, robust, and autonomous robotic systems capable of self-reconfiguration, a fundamental goal of this research is to develop a science-base for self-reconfiguring robot systems. This is a considerable challenge, which requires (1) new designs for reconfigurable systems and (2) new ideas on algorithmic, task-level planning and control that can confer autonomous reconfigurability. In this paper we describe our contributions to (1) and (2) based on a novel module for self-reconfiguring robots we call the *Crystalline Atom*.

There are two basic types of self-reconfiguring robot systems: *heterogeneous* and *homogeneous*. In a heterogeneous system, the modules may be different. In a homogeneous system all the modules are identical.

In our previous work [6, 7, 10] we discuss a homogeneous robot system based on a small and simple module we call a *Robotic Molecule*. This module has already been prototyped. Our
experiments demonstrate that it is capable of self-reconfiguration in three dimensions. We have also demonstrated how systems composed of robotic Molecules can use self-reconfiguration to increase their locomotive versatility [8, 9]. The robotic Molecule uses 4 rotational degrees of freedom to accomplish motion relative to a structure that consists of identical modules.

In this paper we propose a different approach to homogeneous self-reconfiguring robot systems. The new approach uses a module inspired by muscles and amoebas, and is actuated by expansion and contraction. By expanding and contracting the neighbors in a connected structure, an individual module can be moved in general ways relative to the entire structure. This basic operation leads to new algorithms for global self-reconfiguration planning. Figure 3 shows three snapshots from a simulation of a four-unit robot. This basic operation also allows the system to realize a wide range of geometries; for example Figure 25 shows snapshots from a simulation in which a dog-shaped object transforms itself into a couch-shaped object.

We examine a theoretical basis for self-reconfiguration in homogeneous robot systems. We show that a unit modular robot system is self-reconfiguring if (1) its unit module can be used to assemble arbitrary shapes and (2) in any structure composed of unit modules, some module can be relocated arbitrarily. We then describe a module called the Crystalline Atom (see Figure 5) and show that it is the basis for a self-reconfiguring robot.

The Atom (see Figure 5) has square (cubic in 3D) shape with connectors to other modules in the middle of each face. It is activated by three binary actuators that permit the side length of the square to shrink and expand by a factor of two and to make or break connections. This actuation scheme allows an individual module to relocate to arbitrary positions on the surface of a structure of modules in constant time. Previous systems necessitate linear time in the number of modules on the surface [24, 14, 8, 9], because motion from point $A$ to point $B$ requires traversing a path between $A$ and $B$ on the surface of the structure. The expansion/compression actuation of the Crystalline Atom allows a module to relocate from point $A$ to point $B$ by traveling through the volume of the structure.

We discuss planning algorithms for shape metamorphosis for the class of robot systems consisting of Crystalline Atoms, called Crystals. We present and analyze an algorithm called melt-grow for self-reconfiguration where the initial and the goal structures have the same volume. The melt-grow planner achieves shape morphing by using an intermediate structure, which is a projection of the robot modules into a pool on the ground. We believe his approach will lead to reconfiguration algorithms that maintain stability in real-world environments where gravity is present. The planner runs in $O(n^2)$ time, where $n$ is the number of modules in the Crystal.

We also present a simulator we have built to study Crystalline robot systems, give examples from our experience using this simulator, and describe some physical experiments with Crystals. In our lab, we have built 12 Crystalline Atoms.

2 Related work

Related work in self-reconfiguring robots includes that by Cohen et al, in which modules are reconfigurable using external intervention [3]. In [2], Fukuda et al propose a cellular robotic system to coordinate a set of specialized modules. Several specialized modules and ways of composing them were proposed. In [22] Yim studies multiple modes of locomotion that are achieved physically by manually composing a few basic elements in different ways. This work also presents extensive examples of locomotion and self-reconfiguration in simulation. In [11, 24,
21, 12], Murata et al consider a system of modules that can achieve planar motion by walking over one another. The reconfiguration motion is actuated by varying the polarity of electromagnets that are embedded in each module. More recently [13] this group developed a twelve DOF module capable of three-dimensional motion. In [14] Chirikjian et al describe metamorphic robots that can aggregate as two-dimensional structures with varying geometry. The modules are deformable hexagons. This work also examines theoretical bounds for planning the self-reconfiguring motion of such modules. In [10] we have shown a constant-time reduction between robotic molecule structures our group has designed to support self-reconfiguration [8, 9] and metamorphic robots [14].

The robot proposed in this paper is different than the previously proposed modules in its actuation capabilities, which lead to new types of self-reconfiguration planning algorithms. The high-level idea of a shrinkable module that can be a cell in a reconfigurable system has been presented by Tanie et al as the patent [20].

3 The Crystalline Module
3.1 Concept

![Figure 3: Three snapshots from a simulation of locomotion using Crystalline robots. The left image shows the initial state. The middle image shows the robot after shrinking two modules in the direction of motion. The right image shows the robot after relaxing the shrunken modules in the direction of motion. Notice that the entire structure moved forward one unit, in an inchworm-like fashion.](image)

The Crystalline module is a mechanism that has some of the motive properties of muscles, that can be closely packed in 3D space, and that can attach itself to similar units. We chose a design based on cubes with connectors to other modules in the middle of each face. The idea is to build a cube that can contract by a factor of two and expand to the original size (see Figures 3 and 25). We wish to effect compression along all three principal directions (e.g., x, y, z) individually or in parallel. We call the module an Atom, and each connector a bond. Figure 4 shows a design for the mechanics of a two-dimensional (square rather than cubic) implementation of the Atom and Figure 5 shows the physical prototype. We use complimentary rack and pinion mechanisms to implement the contraction and expansion actuation for the two-dimensional prototype. In three dimensions, the rack and pinion mechanisms could be replaced with lead screws. Because squares and cubes are highly regular, most planning algorithms developed in two dimensions can be easily generalized to three. Similarly, algorithms developed in three dimensions can be easily specialized to two dimensional structures.
3.2 A Physical Implementation

Crystalline Atomic modules can be constructed in both two and three dimensions. In two dimensions, Atoms are square; in three dimensions they are cubic. In this section we describe the two-dimensional version of the module we constructed.

The two-dimensional version of the Crystalline Atomic module (see Figure 5) was created based on the CAD designs shown in Figure 4 (left and middle). The module has an expansion/contraction ration of 2. The faces share a common expansion mechanism, so all have to simultaneously be fully extended or fully contracted. Each face of the module has part of a connection mechanism. Two out of the four faces have active mechanisms (see Figure 4 (right)), and the other two have passive channels. Together, these provide a lock-and-key arrangement for forming rigid connections with adjacent modules.

Thus, the entire unit can be realized with three degrees of freedom: one to expand/contract the faces of the Atom, and two for the active connectors. All three degrees of freedom can be implemented with binary actuators. Since Atoms can never rotate relative to one another, the use of two rather than four connectivity degrees of freedom leads to no mechanical limitations. Every inter-Atomic interface of a structure will have one active connection mechanism. The module has on-board electronics and four 3V 2/3A size Lithium batteries, so that it can function untethered.

An alternative design choice would be to allow each face to be fully extended or fully contracted independent of the other faces. This approach would require three additional degrees of freedom and would result in a higher degree of controllability. However, we believe the versatility of such systems with respect to two-dimensional self-reconfiguration is not significantly greater than our
proposed more minimalist design.

The expansion/contraction degree of freedom has been implemented with a rack-and-pinion mechanism (see Figure 6 (Left)). A vertically mounted pinion at the center of the core mates simultaneously all racks. Gear racks are rigidly mounted to the rear of each face. Racks from opposing faces are mounted off-center and racks from adjacent faces are staggered vertically so that they do not intersect at the center of the core. The pinion is driven by a gear motor. Spinning it in one direction extends the racks, which causes the Atom to expand. Spinning in the other direction retracts the racks which causes the Atom to contract. The motor used for these motions is a Lego toy Mini-Motor, which was selected for its small size (5/8 inch cube), useful torque (2oz-in at 10rpm, 12V and 80mA), and low-cost ($11). Two hall-effect sensors serve as bi-level position sensors and are used to control the expansion/contraction movements of the Atoms. One sensor is used to determine if the faces have moved close enough to the core during a contraction movement; the other sensor indicates whether the faces have moved far enough outward during an expansion.

The connection mechanisms are based on a lock-and-key concept (see Figure 6 (Right)). The
passive face contains a deep horizontal channel on its outer surface. Pockets are built into the upper and lower inside surfaces of this channel at the center of the face. The active face contains a gear-motor (the same model Lego mini-motor used to actuate expansion/contraction). A bar (the key) is attached to the output shaft of the motor. At one angle, the bar can move unobstructed through the channel of the passive face and the connector is freed. At another angle, the bar is rotated so that it extends into the pockets of the passive face and the connector is bonded.

When fully contracted, the Atom occupies a square with a 2 inch side. When fully expanded, the Atom occupies a square with a 4 inch side. The height of the Atom is 7 inches and its weight is 12 ounces.

A Crystalline Atom can connect with identical modules to create Crystalline robot systems. Only lattices whose faces are normal to the $x$, $y$, and $z$ axes can be created using Crystalline robots. By manipulating the size of the Atom, it is possible to approximate any finite solid shape to an arbitrary precision using Crystalline modules\(^1\).

### 3.3 Fabrication

We have built 12 prototype Atoms and an environment of connectors to demonstrate the feasibility of our approach. The parts cost of the prototype Atom is about $300, and it takes approximately three days to fabricate and assemble all parts. We have utilized several technologies to reduce the design iteration time, improve the mechanical precision, and reduce the cost of the prototype. Our design begins with a fully detailed and dimensioned 3-D CAD model of

\(^1\)The aliasing error for any shape on a raster display can be arbitrarily reduced by increasing the resolution of the display.
the prototype developed with Pro/Engineer 20.0 from Parametric Technology Corp. We then
fabricate most of the structure using an FDM2000 Fused Deposition Modeling (FDM) Rapid
Prototyping machine manufactured by Stratasys, Inc. This machine and its associated software
converts the data from our CAD model directly into high-strength, light weight ABS plastic
parts. These ABS parts are assembled together with the gears, motors, sensors, batteries, and a
few other parts to create a full module.

3.4 On-board Electronics

Each Atom contains an on-board processor (Atmel AT89C2051 microcontroller), power supply
(four 2/3 A Lithium batteries), and support circuitry, which allow both fully untethered and
tethered operations. Atoms are connected by a wired serial link to a host computer to download
programs. For untethered operations, an experiment specific operating program specified as a
state sequence is first downloaded over a tether. When the tether is removed, an on-board IR
receiver is used to detect synchronization beacons from the host.

3.5 Primitive Operations for Crystal Modules

Crystalline robot systems are dynamic structures: (1) they can move using sequences of
re-configurations to implement locomotion gaits; and (2) they can undergo shape metamorphosis.
The dynamic nature of these systems is supported by the ability of individual modules to move
globally relative to the structure.

The basic operations in a Crystalline robot system are:

- (expand <atom> <dimension>) - expand a compressed Atom in the desired dimension (x,
y, or z)
- (contract <atom> <dimension>) - compress an expanded Atom in the desired dimension
- (bond <atom> <dimension>) - activate one of the Atom’s connectors to bond with a
  neighboring Atom in the structure
- (free <atom> <dimension>) - deactivate one of the Atom’s connectors to break a bond
  with a neighboring Atom in the structure

Figure 3 illustrates the use of these primitives for generating a linear locomotion algorithm
called the inchworm gait for Crystals. The robot consists of four connected Crystalline modules.
The modules rest on a substrate of other Crystalline modules\(^2\). We assume that each module can
compress by a factor of 2. In the first phase of the algorithm, the rightmost module attaches to
the substrate and the middle modules compress. This operation causes the leftmost module to
advance by one unit (where the unit is denoted by the size of the module). In the second phase
of the algorithm the leftmost module makes a connection to the substrate, the rightmost module
disconnects and the middle two modules expand. The net effect of this algorithm is a global
translation of one unit for the Crystal. It is possible to describe similar algorithms for effecting
global translations and 90 degree concave and convex transitions about Crystalline structures.
Figure 7 shows two more examples of shape metamorphosis using the expansion/contraction
actuation mechanism.

\(^2\)Note that the substrate is not necessary for all locomotion gaits.
Figure 7: This figure illustrates the use of the expansion and contraction mechanism for movement. The left figure shows a compound movement with several approach trajectories represented as dotted lines. The upper half of the figure shows a configuration of 8 Atoms, with two compressed Atoms preparing to expand. The lower part of the figure shows the resulting structure after the expansion of the Atoms. The right figure shows a compound movement with a 90° approach trajectory. The top half of the figure shows a configuration of four Atoms with two compressed Atoms about to expand. The bottom half shows the structure that results from the expansion of those Atoms.

4 A Basis for Self-reconfiguration

In this section we use the terminology of Yim, who in [22] defined a reconfigurable system to be unit modular if it is homogeneous. Most existing self-reconfiguring robot systems are based on a unit-modular approach. In particular, the system discussed in this paper is unit modular.

The primary design goal for a self-reconfiguring robot is to allow the robot to assume any arbitrary geometric shape in a dynamic fashion. The geometric, mechanic, and kinematic structure of a unit-modular system define its specific approach to self-reconfiguration. We observe that a system composed of a specific unit module is self-reconfigurable if the unit module satisfies two properties as stated by the following theorem.

Theorem 1 A unit-modular robotic system is self-reconfiguring if its unit modules have the following two properties:

1. **structure formation**: groups of unit modules can be assembled into arbitrarily shaped rigid structures.

2. **module relocation**: in every structure composed of unit modules, some unit module can be relocated to each location on the surface of the structure without human intervention.
Proof: The first property ensures that any geometric structure can be aggregated from some collection of modules. In general, the geometric structure of the module will determine a three-dimensional grid pattern that will approximate the desired structure to within some resolution. For example, a cube-shaped module like the one described in this paper can be packed as any geometric shape subject to the approximation resolution given by the size of the cube. In general, this property is not valid for arbitrary geometric shapes. For example, even a simple shape such as a regular tetrahedron fails to satisfy this property because it does not pack in three dimensions.

The second property provides for shape metamorphosis in a general way: given a starting structure $S$ and a goal structure $G$, it makes it possible to construct $G$ from $S$ incrementally. At any point in the process, some module in $S$ can be relocated without human intervention anywhere on the structure—specifically, it can be placed as part of the structure of $G$. Thus, it is possible to relocate modules from places in $S$ to places in $G$ until $G$ has been fully assembled. □

Corollary 2 A robot system based on the Crystalline module is self-reconfiguring.

In the following two sections we prove this result by arguing that the Crystalline Atom satisfies the conditions of Theorem 1; thus the Crystalline Atom can be the basis for a self-reconfiguring robot system.

4.1 Structure Formation with Crystal Modules

Since Crystalline Atoms are cubic in three dimensions and square in two, they can be packed tightly to approximate any three dimensional structure (or two dimensional structure.) In such a packing we can bond all inter-Atom interfaces shared by two Atoms, thus creating a solid structure which we call a Crystal. Figures 8 and 25 show examples of creating three-dimensional structures by packing Atoms. We then have the following result:

Theorem 3 Any three-dimensional (or two-dimensional) solid structure that allows a cell decomposition where the size of the grid is the size of an individual Atom can be approximated using Crystalline Atoms. The approximation error depends on the size of the Atom.

Thus, by manipulating the size of the Crystalline Atom in the mechanical design phase, we can use this module to represent any solid geometric shape to an arbitrary precision. This idea is analogous to reducing the aliasing error on a raster display by increasing the resolution of the display.

4.2 Relocating a Module on a Crystal

Crystalline module motion is controlled by attaching one Atom to a neighboring Atom and actuating the expansion or compression mechanism (as shown in Figure 7.) An individual module can not relocate without help; however, by contracting and expanding a group of modules in a coordinated way, Atoms can move relative to a structure.

Unlike all other proposed unit modules [22, 11, 13, 14, 8, 9] which can relocate only by traveling on the surface of a structure, Crystalline Atoms can be relocated by traveling through the volume of a Crystal. This interesting property of Crystalline robots is illustrated in Figure 8. The goal is to move the Atom on the surface of the cubic Crystal to some other location on the surface.
Instead of propagating the module along the surface of the large cube (which would require a number of operations linear in the size of the cube) it is possible to reach the goal using a constant number of internal operations. That is, the number of operations remains constant independent of the size of the cube and no matter how the start and goal locations are oriented relative to each other.

Figure 8: A Crystalline module can be pushed into the large cube and popped out at any location on the surface of the cube in constant time. The three images are snapshots from a simulation. The left image shows the initial configuration (with the extra cube located on the side face) and the right image shows the final configuration (where the extra cube is on the top face). The middle image shows the base cube where two internal modules are compressed (not visible in the figure).

The algorithm for module relocation is as follows. First, the module is pulled inside the cube by contracting two internal Atoms (see Figure 9). The two contracted Atoms are selected to be on the supporting line for the start location and adjacent to the intersection of that line and the supporting line for the goal location. Next, two more Atoms are contracted. These Atoms are selected to be on the supporting line for the goal location and adjacent to the intersection of that line and the supporting line for the starting location. The four contracted Atoms are selected so that a void is created at the intersection of the two supporting lines. At this point, the first pair of contracted Atoms are expanded into the void. Finally, the second pair of contracted Atoms are expanded in the direction of the goal, pushing along an entire column of Atoms. The end result is an Atom at the location of the goal. Note that the Atom popped at G is not the same as the Atom that originated at $S^3$. Since all units are identical, it does not matter which actual unit pops out at the goal; what is important is the overall shape transformation. Thus, using this algorithm, an Atom can be relocated in constant time on any convex Crystalline structure$^4$.

When the Crystalline robot structure is non-convex, a similar algorithm effects the module relocation operation in $O(k)$-time, where $k$ is the number of concave angles in the structure. The intuition behind the generalized algorithm is to iterate the algorithm for convex substrates by making transitions at each concave angle between the starting location and the goal location. To make the presentation of the general algorithm cleaner, we now introduce some definitions and supporting results.

\[^3\text{If the supporting lines for the start and goal location in a Crystal do not intersect, two transitions will be required instead of one.}\]

\[^4\text{This result assumes that the actuators are strong enough to push or pull any number of Atoms during these operations.}\]
Figure 9: An internal view of the Algorithm for module relocation on a convex Crystal. Bow-ties mark two Atoms about to be contracted. Small dark diamonds mark two compressed Atoms that are about to be expanded. Dashed lines mark disconnected Atomic interfaces. The supporting lines for the start and goal locations are marked with dotted lines. The leftmost figure shows the initial state, where the Atom to be relocated is on the bottom surface. The rightmost figure shows the final state. The intermediate sequence shows the formation of a void at the intersecting lines between start and goal, and the expansion of an Atom into that void.

**Definition 4.1** A scrunch consists of two adjacent and connected Atoms that are contracted in the dimension normal to their connected face.

**Definition 4.2** An axis is a connected string of at least two Atoms along one dimension. Two axes intersect if they have one Atom in common.

**Definition 4.3** The Atom Connectivity Graph for Crystal $C$ denoted by $ACG(C)$ is an undirected graph whose vertices represent the Atoms in $C$ and whose edges represent bounded inter-Atomic interfaces in $C$.

**Theorem 4** If an axis in a Crystal contains a scrunch, that scrunch can be moved to any position on the axis by the inchworm propagation algorithm illustrated in Figure 3. If one of two intersecting Axes in a Crystal $C$ contains a scrunch, a transition can be performed to transfer the scrunch to the other axis, provided that there is sufficient surrounding structure to maintain connectedness throughout the operation.

**Proof:** Let $i$ be the axis containing the scrunch and $f$ be an intersecting axis to which the scrunch should be transferred (see Figure 10.) Without loss of generality, suppose the scrunch is adjacent to $i \cap f$.

The following algorithm transfers the scrunch (see Figure 10.) A void can be created at $i \cap f$ by compressing two Atoms along $f$. Next, the scrunch along $i$ can be expanded into the void. The scrunch has thus been transferred from $i$ to $f$.

The surrounding structure required for these two steps must be such that $ACG(C)$ remains a connected graph, given that the Atomic interfaces between the compressed modules and the rest of the structure and between the void and the rest of the structure are not present. One way to ensure that this condition is met is to consider axes $i$ and $f$ that are surrounded by collar of modules (see Figure 10 (right)). $\Box$
Figure 10: The scrunch transfer algorithm. The left figure shows a Crystal with axis \( i \) containing a scrunch. The second figure shows the Crystal after the creation of a scrunch along axis \( f \). The third figure shows the Crystal after the transfer of the scrunch from \( i \) to \( f \). The right-most figures show axes \( i \) and \( f \) in a structure that pads them with a collar of Atoms on the outside (\( i' \) and \( f' \)).

Note that the construction in Theorem 4 holds for planar Crystals but it can be generalized easily to three-dimensional structures.

Notation: \( C \) is a concave Crystal.
\( S \) is the start location on the surface of \( C \).
\( G \) is the goal location on the surface of \( C \).

Input: \( C, S, G \).

Atom Relocation: Build \( ACG(C) \).
Find a path \( p \) from \( S \) to \( G \) in \( ACG(C) \).
Create a scrunch \( s \) along the supporting axis for \( S \) by pulling in the Atom that was in the starting location.
Drive \( s \) along \( p \) using transitions from each segment of the path to the next.
Relax \( s \) along the supporting axis for \( G \), popping an Atom out onto \( G \).

Figure 11: The algorithm for relocating an Atom on a concave Crystal. The Atom starts at location \( S \) on the surface of the Crystal and is relocated to \( G \).

Figure 11 shows the algorithm for relocating an Atom on a concave structure. Figure 12 shows an instantiation of this algorithm.

Theorem 5 Let \( C \) be a Crystal with \( k \) concave angles, \( S \) a starting Atom location on the surface of \( C \), and \( G \) the goal Atom location on the surface of \( C \). A relocation from \( S \) to \( G \) through \( C \) can be executed in \( O(k) \) time.
**Proof:** To analyze the complexity of Atom relocation, note that the first step of the algorithm in Figure 11 would require $O(n^2)$ time to generate an optimal path, where $n$ is the number of Atoms in the structure. However, a non-optimal path between $S$ and $G$ can be found in $O(t)$, where $t$ is the number of turns in the path. Note that $t = O(k)$. The running time of the remaining steps is $O(t)$. □

![Figure 12: This figure illustrates the algorithm for relocating a Crystalline module on a concave substrate of Crystalline modules. The left figure shows the initial configuration. The relocating Atom is in the upper right corner of the structure. The goal location is in the bottom left corner. Large dark diamonds mark two Atoms about to be be compressed. Small dark diamonds mark two compressed Atoms about to be expanded. Dark lines mark compressed pairs. The second figure shows the structure after the compression of the first pair of candidate Atoms, and two Atoms preparing for the next compression. The third figure shows two pairs of compressed Atoms and a hole. The fourth figure shows the first compressed pair expanded into the hole and a candidate pair of Atoms for the next compression. The fifth figure shows the state of the structure after this compression, with the resulting hole. The right-most figure shows the structure after an expansion into the hole. At this point, the remaining compressed pair can be expanded into the goal location.]

5 A Planner for Shape Metamorphosis

In this section we describe a planner for self-reconfiguration in Crystalline robot systems. More precisely, given a pair of Crystals $(S, G)$, each composed of $n$ Atoms, a planner finds a feasible reconfiguration plan $P$ that transforms $S$ into $G$. A reconfiguration plan $P$ is a partially ordered sequence of Atom primitive operations. A reconfiguration plan is feasible iff at no time during the execution of the plan does the Crystal become disconnected or crash into itself.

One key observation for planning is that Crystalline systems consist of identical, interchangeable modules, so it is not necessary to compute goal locations for each element. Thus, self-reconfiguration is different from the related warehouse problem (where modules are assigned unique identifiers and have to be placed at desired locations), which is intractable.

Figure 13 presents a simple centralized planning algorithm that we have developed called the Melt-Grow planner. This planner is complete over a useful subset Grain($4$) of Crystals and can run in $O(n^2)$ time, where $n$ is the number of Atoms in the Crystal.

5.1 The Class of Grain Crystals

Before we present the details of the Melt-Grow algorithm, we'll explain what the Grain($4$) subset is and why we use it. The subset Grain($n$) contains all Crystals that can be tiled by cubic
**Notation:**  $S$ is the starting Crystal.
$G$ is the goal Crystal.
$I$ is an intermediate Crystal.

**Input:**  $S, G$.

**Melt-Grow:**  Melt $S$ into $I$.
Grow $G$ out of $I$.

---

Figure 13: The Melt-Grow algorithm for shape metamorphosis.

blocks of Atoms (or square blocks for two-dimensional systems) of side-length $n$, so that the set
of planes (or edges in 2D) that coincide with all sides of all blocks intersect only at block edges
and corners. We call each block a *Grain*. Figure 14 gives an example of a 2D Crystal in $Grain(4)$
and a 2D Crystal not in $Grain(4)$.

![2D Crystals](image)

Figure 14: (Left): a 2D Crystal in $Grain(4)$. (Right): a 2D Crystal not in $Grain(4)$—because the
bottom grain is not aligned with the top ones.

We say that the subset $Grain(4)$ is useful because we can argue that by manipulating the scale
of the Atom, it is possible to approximate any finite solid shape to an arbitrary precision with
some Crystal in $Grain(4)$. Effectively, we have only decreased the resolution of our system—
given enough Grains we can still represent any shape, and we can regain the resolution we have
lost by building smaller Atoms.

Why do we introduce Grains? The reason is that reconfiguration planning is complicated
by the surrounding structure requirements necessary for Atom relocation (see Figure 12 and
Theorem 4). Atoms always require some helpers in order to move, and it can be difficult to
guarantee that such helpers will always be available in the general case. We introduce Grains in
order to encapsulate these requirements. If we restrict ourselves to Grained Crystals, then we can
design reconfiguration plans without considering the low-level surrounding structure requirements
for each movement.

To simplify the generation of reconfiguration plans for Grained Crystals, we define five *Grain
motion primitives*:

- (scrunch <grain> <dimension> <sense>) - create a planar (linear in 2D) compression
in a mobile Grain at one of its faces

- (relax <grain> <dimension> <sense>) - expand a compression at one face of a Grain
- (transfer <grain> <dimension> <sense>) - move a compression at one face of a Grain into the adjacent neighbor Grain
- (propagate <grain> <dimension> <sense>) - move a compression at one face of a Grain to the opposing face of that Grain
- (convert <grain> <dimension₁> <sense₁> <dimension₂> <sense₂>) - relocate a compression at one face of a Grain to one of the orthogonal faces of that Grain

These five primitives are designed to satisfy two goals: (1) They can be assembled into linear sequences to effect Grain relocation. (2) They are always feasible; that is, if any Grain is in a situation to which any of the motion primitives apply, then that motion primitive can always be performed without disconnecting or crashing the Crystal, no matter what the surrounding structure of Grains happens to be.

We demonstrate graphically how the first design goal is met in Figure 15.

![Diagram of Grain relocation](image)

Figure 15: Seven steps of a Grain relocation sequence built from the five Grain motion primitives, illustrating how the primitives can be combined to effect arbitrary grain relocation. At the top of the figure the desired Grain relocation is indicated. These seven steps are repeated four times to complete the relocation.

To meet the second design goal, we first impose the restriction that the Grains surrounding a Grain undergoing a motion primitive (henceforth simply a moving Grain) that are not directly involved in that operation are all fully expanded (i.e. all the Atoms they contain are fully expanded and connected whenever possible throughout the operation). We show later that the reconfiguration plans generated by the Melt-Grow planner always satisfy this restriction. Given this, we can meet the second design goal by building each primitive to maintain three invariants at all times during its execution: (1) The moving Grain remains internally connected. (2) The Atoms in the moving Grain never crash. (3) There is some connected path from some Atom in
every neighboring Grain, through the moving Grain, to some Atom in every other neighboring Grain.

With one exception, it’s trivial to design all of the Grain motion primitives to maintain each of these invariants. The exception is convert. After some development, we came up with the routine in Figure 16.

![Diagram of Grain motion primitives](image)

Figure 16: The convert operation can be carried out in 12 steps. Note that the algorithm is symmetric; the Grain state after the sixth step is a rotationally symmetric midpoint, and the remaining six steps are simply the reverse of the first six applied in the normal dimension.

The difficulty of the convert operation is also the reason we use Grains of size four (rather than something smaller). We do not have a design for the convert operation for any smaller Grain.

5.2 The Melt-Grow Planner

In this section we examine the details of the Melt-Grow planner. As presented above, there are two major components: the Melt algorithm and the Grow algorithm. At a high level, Melt works by finding a mobile Grain $g$ in $S$, transporting $g$ to a place in $I$, and repeating until all Grains are in $I$. Similarly, Grow works by selecting mobile Grains from $I$ and transporting them to locations in $G$ until all Grains are in $G$. A Grain is mobile iff it can be removed without disconnecting the Crystal.

We use the intermediate Crystal $I$ for two reasons: (1) To help maintain stability during reconfiguration in situations where gravity is present. (2) To simplify the selection of mobile Grains.

The intermediate Crystal can help us satisfy (1) if we arrange $S$, $I$, and $G$ to allow Melting to happen from the top down and Growing to happen from the bottom up. If $S$ and $G$ are
three-dimensional, then we project them onto a two-dimensional $I$. If they are two-dimensional, then we project them onto a linear $I$.

To see how the intermediate Crystal satisfies (2), observe that another planner might generate Crystal states in which no out-of-place Grain is mobile. This would require the sacrificial selection of a mobile Grain that is already in-place (by “in-place” we mean “occupying a position in $G$”). For simplicity, we choose to avoid dealing with such situations. By designing $I$ to be as disjoint as possible from $S$ and $G$ (of course, they must all contain at least one Grain in common), we can guarantee that there will always be at least one obvious mobile Grain while Melting and Growing$^5$.

Many types of intermediate Crystal are possible; for simplicity, we use a planar spiral of Grains for 3D systems and a one-dimensional line of Grains for 2D systems. Such intermediate Crystal designs make locating mobile Grains in $I$ trivial.

Before we formalize the details of the Melt-Grow planner, we define the Grain Connectivity Graph of a Grained Crystal $C$, $GCG(C)$, to be an undirected graph whose vertices represent Grains in $C$ and whose edges represent active connections between neighboring Grains in $C$.

Figure 17 presents the top-level details of the Melt-Grow algorithm, and Figure 18 illustrates the algorithm with a concrete example. The first step is to compute the location and structure of the intermediate Crystal $I$. This is done by the function LocateStem. The function DesignIntermediate generates an intermediate Crystal with suitable volume. The second step of the algorithm is to Melt $S$ into $I$. The third step is to Grow $I$ into $G$. These steps require locating a mobile Grain, locating a good destination (a parent Grain), and finding a path for the mobile Grain to the destination. Mobile Grains in a Crystal can be located by searching for vertices which are not articulation points in the Crystal’s $GCG$. While melting, any mobile Grain still in $S$ is a suitable mover, and while growing, any mobile Grain still in $I$ is a suitable mover. Parent Grains can be located by searching for Grains that are adjacent to yet-to-be-filled vacancies. While melting, any such Grain in $I$ is a suitable parent, and while growing, any such Grain in $G$ is a suitable parent.

After locating a mobile Grain and a parent, the mobile Grain is transported to a space adjacent to the parent by (1) Finding a route through the Crystal. (2) Decomposing the route into a sequence of Grain motion primitives. (3) Executing the Grain motion primitives, as in Figure 15.

5.3 Analysis of the Melt-Grow Planner

In this section we analyze the completeness, complexity and optimality of the Melt-Grow planning algorithm. We start by proving a supporting lemma.

**Lemma 6** All finite connected graphs with at least two vertices contain at least two vertices which are not articulation points.

**Proof:** We can prove this Lemma by induction on the number of vertices in the graph. In the base case we have a graph with only two vertices. Trivially, neither of these is an articulation point, so we start with a non-articulation point count of two. For the inductive step, we show that

$^5$As suggested by Chirikjian et al in [1], another way to avoid backtracking is to start with a maximal connected overlap of $S$ and $G$. This may be faster than melting to and growing from $I$, however this scheme does not seem to offer any gravitational stability. Note that in [1] a simply connected overlap is required, while in our system any connected overlap will do because our modules can effectively pass through each other.
\textbf{Melt-Grow}(Crystal }S, \text{ Crystal }G\)

\begin{itemize}
    \item Grain \texttt{stem} $\leftarrow$ \texttt{LocateStem}(\textit{S})
    \item Crystal \textit{I} $\leftarrow$ \texttt{Melt}(\textit{S, stem})
    \item \texttt{Grow}(\textit{I, stem, G})
\end{itemize}

\textbf{Melt}(Crystal }S, \text{ Grain } stem\)

\begin{itemize}
    \item Crystal \textit{I} $\leftarrow$ \{\textit{stem}\}
    \item Crystal \textit{C} $\leftarrow$ \texttt{DesignIntermediate}(\texttt{Volume}(\textit{S}), \textit{stem})
    \item While \textit{S} $\neq$ \{\textit{stem}\}
 \begin{itemize}
        \item Grain \texttt{mover} $\leftarrow$ \texttt{FindMobile}(\textit{S})
        \item Grain \texttt{parent} $\leftarrow$ \texttt{FindParent}(\textit{I, C})
        \item \texttt{Transport}(\textit{mover, parent, S \cup I})
    \end{itemize}
    \item Return \textit{I}
\end{itemize}

\textbf{Grow}(Crystal }\textit{I}, \text{ Grain } stem, \text{ Crystal } G\)

\begin{itemize}
    \item Crystal \textit{C} $\leftarrow$ \{\textit{stem}\}
    \item While \textit{I} $\neq$ \{\textit{stem}\}
        \begin{itemize}
            \item Grain \texttt{mover} $\leftarrow$ \texttt{FindMobile}(\textit{I})
            \item Grain \texttt{parent} $\leftarrow$ \texttt{FindParent}(\textit{C, G})
            \item \texttt{Transport}(\textit{mover, parent, C \cup I})
        \end{itemize}
\end{itemize}

\begin{flushright}
Figure 17: The implementation of the Melt-Grow algorithm.
\end{flushright}
Figure 18: Reconfiguring a two-dimensional table shape to a chair shape using the Melt-Grow algorithm. Each square represents one 4x4 Grain (16 Atoms). Grains marked □ are mobile, the stem Grain is marked □, and candidate parent Grains are marked □. This figure represents schematically the output of a simulation where the table and chair are composed of 176 Atoms each.

the non-articulation point count cannot decrease as we add vertices to the graph. We consider two cases as we add each vertex:

Case 1: If we only connect the new vertex to one existing vertex, then we have made that existing vertex an articulation point, possibly decreasing the total non-articulation point count by one. But the new vertex itself is not an articulation point, so such a decrease will always be counterbalanced.

Case 2: If we connect the new vertex to more than one existing vertex, then the new vertex cannot be an articulation point because there already existed paths from each of its neighbors to every other neighbor. Also, addition of the new vertex cannot force any of its neighbors to become an articulation point because there are paths to the new vertex from other vertices.

□

**Theorem 7** The Melt-Grow algorithm is complete for the subset Grain(4) of Crystals. The running time of the Melt-Grow algorithm is $O(n^2)$, where $n$ is the number of Atoms in the Crystal.

**Proof:** We can prove that the Melt-Grow algorithm is complete for all Crystals $S$ and $G$ in Grain(4) in two stages. First, we show that we can always find suitable mobile and parent Grains while Melting and Growing:
1. While Melting there is always at least one mobile Grain left in $S$: As stated above, mobile Grains correspond to vertices in the GCG which are not articulation points. Since, by Lemma 6, all finite connected graphs with at least two vertices have at least two non-articulation points, and since the GCG of the remnants of $S$ will always be such a graph (the stem will be one vertex, as will be all remaining Grains in $S$), there will always be at least one vertex that is not an articulation point, and the corresponding Grain will be mobile.

2. While Melting there is always at least one parent Grain in $I$: Since we build $I$ by filling in a “skeleton” Crystal, which represents the shape $I$ will take when it is complete, there will always be some outer boundary surface of Grains in the current $I$ which are adjacent to vacant Grain locations in the skeleton. We can choose any such Grain in $I$ to be a parent.

3. While Growing there is always at least one mobile Grain left in $I$: The proof of this is the same as for 1.

4. While Growing there is always at least one parent Grain in $G$: The proof of this is the same as for 2.

The second stage of the proof demonstrates that we can always transport mobile Grains to their destinations adjacent to parent Grains. More formally, we show that it is always feasible to transport any mobile Grain to any parent Grain in any Crystal:

1. There is always a decomposable path from the mobile Grain to the parent Grain: Since the GCG is always a connected graph, there will always be a path from every Grain to every other Grain. We can decompose any path into a sequence of Grain motion primitives by converting all linear segments in the path to sequences of propagate and transfer operations and all turns in the path to convert and transfer operations.

2. Any such decomposition from a mobile Grain to a parent Grain can be feasibly executed: As long as all surrounding Grains are fully expanded, the Grain motion primitives are all independently feasible by design. Since the Melt-Grow planner only executes one Grain transport at a time, this surrounding structure requirement will always be met. Thus, the execution of any serial composition of the Grain motion primitives, which constitutes the execution of any decomposed path, is feasible because only one primitive is executed at a time.

We can prove that the running time of the Melt-Grow algorithm is $O(n^2)$ from the bottom up. First, we make a few observations:

1. The GCG has maximum fan-out of 4 in 2D (6 in 3D). Thus, with suitable data structures, the planner can DFS the GCG in $O(n)$.

2. With suitable data structures, the planner can access (i.e. query existence, change state, etc.) any Grain in $O(1)$. 
The most expensive operation in **Transport** is to find a path from the mobile Grain to the parent Grain. By our first observation, we can find such a path by a DFS in the GCG in $O(n)$ (that is, if we don’t require an optimal path, see below). Thus **Transport** is $O(n)$.

The running time for **FindParent** is $O(n)$, because it can be as simple as a linear search through the Crystal-in-progress. **FindMobile** is also $O(n)$ because the articulation points of any finite connected graph $G = (V, E)$ can be found in $O(E)$ time, and in the GCG $E$ is $O(V)$.

**DesignIntermediate** can construct a plan for the intermediate Crystal in $O(n)$. **FindStem** can be just a linear search through $S$, so it is also $O(n)$.

**Melt** and **Grow** are each $O(n^2)$, because they perform several $O(n)$ operations for each Grain in the Crystal. Thus, **Melt-Grow** is $O(n^2)$.

□

An optimal planner would find the shortest reconfiguration plan for any specified $S$ and $G$. The Melt-Grow planner sacrifices optimality for simplicity. Also, during any Grain relocation there may be several paths through the Crystal from the mobile Grain to the parent Grain. In the running time analysis, we suggested that some such path can be found by DFS through the GCG in $O(n)$. Paths found this way are not guaranteed to be optimal. If optimal paths are desired, then Dijkstra’s algorithm can be used, but this will raise the overall running time of Melt-Grow to $O(n^3)$.

### 5.4 Implementation of the Melt-Grow Planner

A two-dimensional implementation of the Melt-Grow planner was developed and interfaced to the *xtalsim* Crystalline Atomic robot simulator described in Section 6.2. The table-to-chair reconfiguration in Figure 18 was planned automatically. This reconfiguration of 11 Grains (176 Atoms) requires less than 1 second for the planning stage and about 1 minute for the simulation stage on our workstations.

The implementation of the Melt-Grow planner accepts descriptions of $S$ and $G$ as lists of Grain centroid coordinates. All further action is completely automated.

### 6 Experiments with Self-reconfiguring Crystals

#### 6.1 Physical Experiments

We have constructed ten prototype modules and used them to perform experiments to evaluate the feasibility of using multiple Atoms to demonstrate reconfiguration.

To facilitate experimentation, a row of 8 fixed passive connectors was constructed to simulate the surface of a Crystal. This arrangement not only frees us from having to construct many units at the outset, but it also allows us to perform experiments that are focused narrowly on the specific activities under study. The fixed connectors are placed as they would be for a flat Crystal surface composed of 8 contracted Atoms. In the descriptions that follow, we will refer to two of the prototype Atoms as $a$ and $b$, and we will number the fixed connectors 0–7. The North and West faces of $a$ and $b$ (those that contain active connection mechanisms) will be referred to as $a.n/b.n$ and $a.w/b.w$, respectively, and the South and East faces will be similarly named. $a$ and $b$ are always oriented so that $a.n$ and $b.n$ are facing the row of fixed connectors.

The first experiment was designed to determine if an Atom could reliably expand and then connect with a neighbor. Initially, $a$ was expanded and affixed to 0 (at $a.n$). $b$ was contracted
and affixed to 2 at (b.n). The following state sequence was then executed on the Atoms, as shown in Figure 19:

1. expand b
2. connect b.w

![Diagram showing state sequence](image)

Figure 19: In the first experiment, a was affixed to 0, b to 2. b was programmed to expand and connect with a.

The first experiment yielded positive results. It demonstrated that b could reliably expand and connect with a in most cases. One situation where it was observed to fail was when an especially low-friction environment surface was used. In this case, step 2 usually does not complete because play in the mechanics allows Atom a to be too easily pushed away from Atom b as it expands in step 1.

The second experiment was designed to evaluate whether Atoms could work together to effect a reconfiguration. Initially, both a and b were contracted. a was connected to 0 (at a.n) and b was connected to 1 (at b.n). a and b were connected together at b.w. The Atoms were programmed with state sequences designed to perform a variant of inchworm translation along the fixed connectors:

1. free b.n from 1
2. expand a
3. expand b
4. connect b.n to 2
5. disconnect a.n from 0
6. contract a and b
7. connect a.n to 1
8. repeat

This sequence is illustrated in Figure 20, and Figure 21 presents several photographs of the Atom prototype hardware performing the experiment.

In the third experiment, we explored the motion primitives for Crystal modules in a structure constructed out of 10 modules. We created a 3 × 3 square structure with an additional Crystal module on its surface. We developed a program that allowed this tenth Crystal module to be propagated across a row in the 3 × 3 Crystal. Figure 22 shows snapshots from this experiment. We are currently using the 10 modules to experiment with more extensive shape morphing.
Figure 20: The second experiment tests an inchworm propagation algorithm.

Figure 21: Several snapshots of the Atom prototype hardware performing the inchworm experiment.

6.2 Simulation of Large Structures

For reconfigurations that include more than a few Atoms, and for three-dimensional reconfigurations, it becomes difficult to keep track of all aspects of the system by hand. In these cases, a software simulation system is highly desirable. Such a simulator facilitates the design and debugging of reconfiguration algorithms.

We have developed a software simulator for Crystalline Atomic robots called xtalsim. Unlike the hardware prototype that we have been developing, which is two-dimensional, xtalsim is designed to simulate Crystalline Atomic robots that are fully three-dimensional. Of course, xtalsim can also be used for simulations of two-dimensional systems.

Xtalsim was developed as two components, each of which is implemented as a separate program: the simulation engine xtalexpr, and the interactive display animator xtalanim. Xtalexpr accepts a simulation script called a \textit{relative deformation}, specified in a simple but powerful input language, and produces a list of explicit Crystal states called an \textit{absolute deformation}. Xtalanim accepts absolute deformations and displays them as interactive three-dimensional animations. Splitting the simulator into these two components simplifies development because it neatly partitions the two main functions of the simulator: xtalexpr is entirely concerned with the logical model of the Crystalline Atomic system, and xtalanim is focused on the user-interface and rendering operations.
The relative deformation input script that is presented to xtalsim is human-readable and operator friendly. It can either be manually generated or obtained as the product of an automated planning algorithm. Only the essential expand, contract, bond, and free actions are included in the relative deformation. Xtalexp efficiently extrapolates all the effects of the specified actions and fills in the implied Atom motions automatically. Additionally, xtalexp performs careful feasibility checks to ensure that only self-consistent\(^6\) reconfigurations are executed.

Every relative deformation script follows the same basic format. First, a list of commands specifying the initial Crystal state is presented. Next, a list of commands that specify the sequential updates which effect the reconfiguration under study is included. C-style /* comments */ are allowed. Figure 23 illustrates the relative deformation script used to specify the inchworm simulation depicted in Figure 3 (a line-by-line commentary is included in lieu of a full description of the relative deformation input language).

xtalanim presents a quick-and-dirty GUI for examining the output from xtalexp (Figure 24).

A more involved simulated reconfiguration than what we have seen so far is presented in Figure 25, which shows several snapshots from a simulation in which a three-dimensional “dog” shape is metamorphosed into a “couch” shape.

The reconfiguration in Figure 25 is one of several manually planned reconfigurations that we have simulated with Xtalsim. Other such reconfigurations include the inchworm locomotion algorithm shown in Figure 3. Additionally, xtalsim been interfaced to our implementation of the Melt-Grow automated planner, as described in Section 5.

So far, we have used xtalsim to simulate about a dozen different reconfigurations. These range in complexity from the inchworm, which took only several minutes to implement, to the dogcouch, which required about 4 days of development, to the automatically planned table-to-chair reconfiguration illustrated in Figure 18, which contains 176 Atoms and was planned and simulated in several minutes.

7 Discussion and Future Work

We presented the Crystalline Atom and proved that this module is the basis for a homogeneous, unit-modular, self-reconfigurable robot system. The module is inspired by a muscle-like actuation mechanism. The Atom has 3 DOFs that allow it to make and break connections with identical

\(^6\)An example of a reconfiguration which is not self-consistent would be one in which some Atom collides with some other Atom.
/* initial crystal */
((atom (0 0 0) "head" bbf eee) /* create an Atom named "head" at (0,0,0) */
 (atom (-4 0 0) "m1" bff eee) /* similar for "m1" */
 (atom (-8 0 0) "m2" bff eee) /* similar for "m2" */
 (atom (-12 0 0) "tail" fbf eee)) /* similar for "tail" */

/* updates */
(((free "tail" y)) /* disconnect tail from the base */
 ((contract "m1" x) (contract "m2" x)) /* contract m1 and m2 in x */
 ((bond "tail" y)(free "head" y)) /* attach tail to the base and free head */
 ((expand "m1" x)(expand "m2" x)) /* expand m1 and m2 in x */
 ((bond "head" y) ))) /* attach head to the substrate */

Figure 23: The relative deformation script used to specify the inchworm simulation depicted graphically in Figure 3.

Figure 24: A screenshot of the xtalanim GUI.
Figure 25: Five snapshots from a simulation using crystalline robots. The initial configuration (on the left) is a dog-shaped object. The final configuration (on the right) is a couch-shaped object. The middle images show intermediate steps in the transformation from dog to couch. The planning for this transformation was done manually. Note that some atoms are left in a compressed state so that the volume of the final shape is less than the volume of the initial shape.

modules and to expand and contract by a factor of two. This actuation mechanism supports very fast algorithms for relocating one module on the surface of a Crystal, which leads to an efficient $O(n^2)$ planner for shape metamorphoses. We have described our first physical prototypes for the Atom and discussed our experimental results. We have also presented a simulation system we created to study the self-reconfiguration of Crystals consisting of hundreds of Atoms.

Acknowledgements

We are grateful to Keith Kotay for extensive discussions about the Crystalline module. We are also grateful to Michael Shin for his help in implementing the simulator.

This paper describes research done in the Dartmouth Robotics Laboratory. Support for this work was provided through the NSF CAREER award IRI-9624286 and the NSF award IRI-9714332.

References


