6-8-1999

A Two Dimensional Crystalline Atomic Unit Modular Self-reconfigurable Robot

Marsette Arthur Vona III
Dartmouth College

Dartmouth College Computer Science Technical Report PCS-TR99-348

A Two Dimensional Crystalline Atomic Unit Modular Self-reconfigurable Robot

By
MARSETTE ARTHUR VONA, III

HONORS THESIS

Department of Computer Science
Dartmouth College
Hanover, New Hampshire

June 8, 1999

Approved:_____
(Advisor's Signature)

_____
(Author's Signature)

Department of Computer Science
Dartmouth College

"A Two Dimensional Crystalline Atomic Unit Modular Self-reconfigurable Robot"

Marsette Arthur Vona, III

HONORS THESIS

ABSTRACT

Self-reconfigurable robots are designed so that they can change their external shape without human intervention. One general way to achieve such functionality is to build a robot composed of multiple, identical *unit modules*. If the modules are designed so that they can be assembled into rigid structures, and so that individual units within such structures can be relocated within and about the structure, then self-reconfiguration is possible.

We propose the Crystalline Atomic unit modular self-reconfigurable robot, where each unit is called an *Atom*. In two dimensions, an Atom is square. Connectors at the faces of each Atom support structure formation (such structures are called *Crystals*). Centrally placed prismatic degrees of freedom give Atoms the ability to contract their outer side-length by a constant factor. By contracting and expanding groups of Atoms in a coordinated way, Atoms can relocate within and about Crystals. Thus Atoms are shown to satisfy the two properties necessary to function as modules of a self-reconfigurable robot.

A powerful software simulator for Crystalline Atomic robots in two and three dimensions, called *xtalsim*, is presented. Xtalsim includes a high-level language interface for specifying reconfigurations, an engine which expands implicit reconfiguration plans into explicit Crystal state sequences, and an interactive animator which displays the results in a virtual environment.

An automated planning algorithm for generating reconfigurations, called the *Melt-Grow* planner, is described. The Melt-Grow planner is fast ($O(n^2)$ for Crystals of n Atoms) and complete for a fully general subset of Crystals. The Melt-Grow planner is implemented and interfaced to xtalsim, and an automatically planned reconfiguration is simulated.

Finally, the mechanics, electronics, and software for an Atom implementation are developed. Two Atoms are constructed and experiments are performed which indicate that, with some hardware improvements, an interesting self-reconfiguration could be demonstrated by a group of Atoms.

i Table of Contents

ii List of Tables

iii List of Illustrations

iv Preface

This preface includes preliminary material and information. It is divided into three sections:

- **A project history**, which documents the origins of this thesis and which describes how this work fits in with related research at the Dartmouth Robotics Laboratory.
- **A description of the dual nature of this work:** parts of this project serve in fulfillment of the requirements of the Honors Program in Engineering Sciences, other parts serve in fulfillment of the requirements of the Honors Program in Computer Science.
- **Acknowledgements**, which give credit to persons who have helped in the development of this work.

iv.1 Project History

The Dartmouth Robotics Laboratory (DRL), under the direction of Professor Daniela Rus, has been actively pursuing research in self-reconfigurable robotics since the fall of 1997. The author is a member of this research group and was involved in the design of DRL's first self-reconfigurable robot, called the Molecule [13, 14, 15, 16, 18, 27].

The original concept for the Crystalline Atomic self-reconfigurable robot, the central topic of this thesis and DRL's second self-reconfigurable robot, was formed by our group in the Spring of 1998. Xtalsim, a software simulator for Crystalline Atomic robots described in Chapter 2, was developed during the following Summer. The Melt-Grow planner described in Chapter 3 was developed in the Fall and implemented in the subsequent Winter. Recently, this planner has been presented at *the 1999 International Conference on Robotics and Automation* as [32]. Also during that Winter the hardware design presented in Chapter 4 for a two-dimensional Atom was developed. Finally, two Atoms were constructed in the Spring of 1999 and hardware experiments were performed.

iv.2 Nature of Dual Thesis

Some parts of this project serve in fulfillment of the requirements of the Honors Program in Engineering Sciences, and other parts serve in fulfillment of the requirements of the Honors Program in Computer Science. This arrangement has been officially accepted by Ursula Gibson, Undergraduate Advisor in Engineering Sciences, and by Thomas Cormen, Undergraduate Advisor in Computer Science.

The following segments of this work are to be considered especially for the Honors Thesis in Engineering Sciences:

- Mechanical design of the two-dimensional Atom
- Design for the on-board and host control electronics for the Atom
- Development of the on-board and host software for the Atom

These segments are to be considered for the Honors Thesis in Computer Science:

- Design and implementation of xtalsim, a software simulator for Crystalline Atomic robots
- Development of several reconfiguration simulations
- Design and implementation of the Melt-Grow planner, a centralized planner for shape metamorphosis of Crystalline Atomic robots

The basic concept of the Crystalline Atomic robot and the construction and evaluation of two Atoms are to be considered fundamental to both the Engineering Sciences Thesis and the Computer Science Thesis.

iv.3 Acknowledgments

- **Daniela Rus**, director of the DRL and the author's advisor in Computer Science, has provided extensive support for this project in many forms. Professor Rus was involved in the development of the Melt-Grow planner, among other things, and she co-authored

[32]. Funding for this project was provided by Professor Rus, and most of the design, construction, and experimental work for all segments of the project were conducted at the DRL.

- **Laura Ray**, the author's advisor in Engineering Science, provided helpful advice and guidance in the design and construction of the Atom.
- **Keith Kotay**, a graduate student in Computer Science and a member of the research group at DRL, was involved from the start in discussions about the Crystalline Atomic system. Keith helped shape the fundamental design of the Atom. Keith also contributed the central idea behind the design of the fault-tolerant connection mechanism included in the hardware implementation.
- **Michael Shin**, an undergraduate student in Computer Science, was involved in the design of the central simulation algorithm in xtalsim. Michael also developed the original implementation of the 3-D animation component of xtalsim.
- **Brian Locke**, **Pete Fontaine**, and **Leonard Parker**, staff members at the Thayer School of Engineering Machine Shop, have trained the author in the use of the tools used to fabricate the Atom.

1 Introduction

In the first part of this introduction, the concept of a self-reconfigurable robot is presented and motivated. *Crystalline Atomic* self-reconfigurable robots, the specific topic of this research, are introduced next. The scope and extent of this work and the structure of this document are then described. Finally, related research in the field of self-reconfigurable robotics is summarized.

1.1 Self-reconfigurable Robots

We wish to develop versatile, extensible, and robust robotic systems. We believe these goals can be attained by self-reconfigurable robots. *Self-reconfigurable* robots are designed so that they can change their aggregate external shape without human intervention. The field of self-reconfigurable robotics is a sub-field in the more general area of reconfigurable robotics, which also includes systems whose shape can be changed manually. Self-reconfigurable robots can be two-dimensional, in which case their shapes are restricted to the plane, or they can be fully three-dimensional.

A primary design goal for a self-reconfigurable robot is to allow the robot to assume any arbitrary geometric shape. This is different from other types of shape-changing robots, which may only take one of a small number of shapes. Figure 1.1 illustrates this concept.

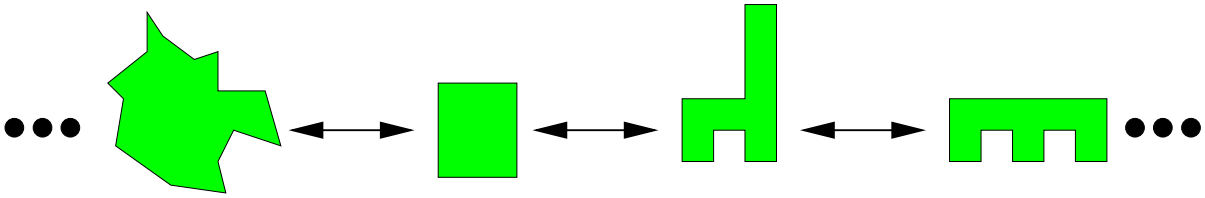


Figure 1.1: This schematic representation of a two-dimensional self-reconfigurable robot illustrates that it can take on any arbitrary shape, including the amoebic form at left, a square shape, a chair shape, and the overturned E shape at right.

We call a reconfiguration from one shape to another *shape metamorphosis.* The capacity for shape metamorphosis is the defining characteristic for a self-reconfigurable

robot. Other useful and important capabilities are attained through shape metamorphosis or by composing sequences of shape metamorphoses.

1.1.1 Locomotion and Manipulation

A very important example which illustrates a useful composition of shape metamorphoses is locomotion: self-reconfigurable robots can move relative to their environment. One way this might be accomplished is to have the robot metamorphose from one state to another in a statically stable gait, as illustrated in Figure 1.2.

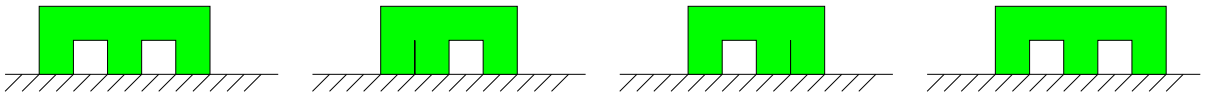


Figure 1.2: A schematic representation of one way a self-reconfigurable robot might locomote. A statically stable gait is used to translate the robot to the right.

It is important to note that manipulation, since it can be considered the dual of locomotion in a general sense, can also be accomplished through the composition of shape metamorphoses.

1.1.2 Self-Repair

Another potentially useful application for shape metamorphoses is to realize general *self-repair*. If an arbitrary part of a fixed-architecture robot fails, the robot cannot usually repair itself. A human (or perhaps another robot) must perform the repair. This is not necessarily the case for a self-reconfigurable robot: If we have constructed the robot to contain some extra material in an unobtrusive location, it may be possible to compose shape metamorphoses to excise any failed part of the robot and to replace it with the spare material. This concept is illustrated in Figure 1.3.

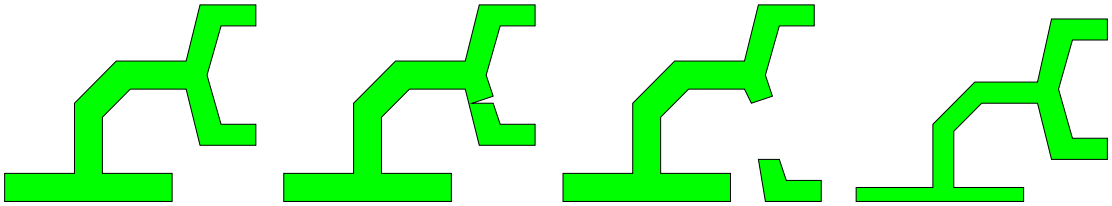


Figure 1.3: A self-reconfigurable robot undergoing self-repair. The robot contains a gripper which has been partly damaged. The damaged part is ejected and replaced with spare material from the body of the robot.

1.1.3 The Unit Modular Approach

In the previous section we defined the concept of self-reconfigurable robots and explored some of their capabilities, we now examine a fundamental approach which is basic to the realization of practical self-reconfigurable systems. This approach is to develop systems composed of multiple, identical *unit modules*. The term "unit module" was first popularized in [30] as a descriptive term for the homogeneous mechanical units that compose a manually-reconfigurable robotic system therein introduced. As summarized below, every major self-reconfigurable robotic system that has been developed has been based on a unit modular approach. A system composed of a specific unit module is self-reconfigurable if the unit module satisfies two properties. These are stated as the following theorem:

Theorem 1.1 *A unit modular robotic system is self-reconfigurable if its unit modules possess the following two properties:*
(1) ***structure assembly****: a group of the unit modules can be mechanically assembled into arbitrarily shaped rigid structures*
(2) ***module relocation****: in any structure composed of the unit modules, some unit module can be relocated arbitrarily within and/or about the structure without human intervention*

Proof: (1) satisfies the design goal of allowing any arbitrary shape to be assumed by the robot. (2) provides for shape metamorphosis in a general way: Given a starting structure ***S*** and a goal structure ***G***, relocate modules from places in ***S*** to places in ***G*** until ***G*** has been fully assembled.

Theorem 1.1 will be used shortly to argue that systems composed of the unit module proposed in this research, the Atom, can self-reconfigure. But before we introduce the

Atom, we will consider applications and research issues which motivate the study of self-reconfigurable robotics.

1.1.4 Applications

Self-reconfigurable robotic systems can be applied in situations where few, if any, existing technologies are suited. Four such applications are: situations with incomplete a-priori task knowledge, situations requiring robustness in inaccessible environments, visualization, and entertainment.

- **Situations With Incomplete A-priori Task Knowledge** It is often difficult to produce fixed-architecture robots which will reliably perform in situations where the operating task is not well specified at design time. General examples are real-world locomotion and manipulation, where the robot may be called upon to traverse or manipulate environments which the designers did not consider. A more concrete example is disaster site reconnaissance: a self-reconfigurable robot might be deployed into the rubble of a large building that has been destroyed by an earthquake to search for survivors. If the robot encounters some boulders which prevent its passage except through a narrow gap, then it could reconfigure into a snake form to pass through the gap. If the robot then comes across a pool of water, it might reconfigure into a flagella structure and swim across. When the robot locates a survivor it might even change into a structural shape in order to stabilize the surrounding rubble.
- **Situations Requiring Robustness in Inaccessible Environments** A major application for all types of robots is to replace humans in situations where it is inordinately expensive or dangerous to send people. Examples include planetary and undersea exploration, nuclear reactor operations, and some industrial situations. The fixed-architecture robots which are currently employed in these situations are often adequate, but reliability and

robustness are almost always major complicating issues in their design and operation. Failure of some part of a fixed-architecture robot is often devastating, and the repair of such failures usually requires human intervention (or the intervention of another robot in a few cases). Self-reconfigurable robots, however, can be made to self-repair as described above. If any part of a self-reconfigurable robot breaks, the robot could potentially eject that part and replace it, all without human intervention.

- **Visualization** It is now common to use software to visualize three dimensional data. The use of special-purpose output hardware, called rapid prototypers, to automatically construct real three dimensional objects from such data is also becoming more widespread. Though current rapid prototyping systems can be quite speedy, requiring only a few hours to build geometries that might take days or weeks to produce by other means, there may be a better way for some applications. An appropriate self-reconfigurable robot can be viewed as a sort of rapid prototyper, where the hardware itself becomes the prototype. Such prototypes are probably best considered temporary geometry representations, but for some applications this exactly what is required. In these cases, this approach might be both faster and more economical (as no consumables are required) than conventional rapid prototyping systems.
- **Entertainment** Living room furniture that incorporates self-reconfigurable robotics could be the home re-decorator's dream! A self-reconfigurable robot might also be a fun toy for technically-minded youngsters.

1.1.5 Research Issues

On a more serious note than reconfigurable furniture, it is important to point out that the study of self-reconfigurable robots raises some important research issues. Self-reconfiguration poses new challenges in designing and controlling distributed robot systems.

Because the connectivity topology of unit modular self-reconfigurable robot systems changes dynamically during shape metamorphosis, new models of synchronization, communication, control, and planning are needed. Solutions to these problems will impact more broadly the research field of computation in distributed systems. For example, new models of distributed computing are emerging due to the proliferation of wireless computers. As with self-reconfiguring robot systems, topology and reachability changes dynamically in wireless networks of mobile computers, requiring new solutions to communication and routing.

Self-reconfigurable robots also pose new engineering challenges. New mechanical connection mechanisms are needed that are strong, low-power, fault-tolerant, and misalignment-correcting. New power storage and distribution schemes are needed in order to supply energy to all the unit modules of a self-reconfigurable robot in a reliable and efficient way while still allowing the modules to reconfigure.

1.2 Crystalline Atomic Robots

In this section we discuss the design of the specific unit modular self-reconfigurable robot proposed by this research, the *Crystalline Atom*. The idea behind the Crystalline Atom (or just the *Atom*) is to create a mechanism that can volumetrically compress itself and that can attach itself to other units. We choose a design which can be realized in either two or three dimensions. In three dimensions Atoms are cubic, in two dimensions they are square. Atoms contain connectors to other modules at the center of each face and central prismatic degrees of freedom which serve to reduce the Atom side-length by a constant factor. Atoms can be assembled together into structures. We call such structures *Crystals.*

We can formalize the properties of Atoms by defining four *Atom motion primitives* which encapsulate the basic operations of Atoms within a Crystalline Atomic system:

- (***expand*** <atom>, <dimension>) - expand an Atom in the selected dimension
- (***contract*** <atom>, <dimension>) - contract an Atom in the selected dimension
- (***bond*** <atom>, <dimension>, <sense>) - activate a connector at one of the inter-Atomic interfaces
- (***free*** <atom>, <dimension>, <sense>) - free a connector at one of the inter-Atomic interfaces

Figure 1.4 illustrates the use of these primitives for generating a locomotion algorithm called the *inchworm propagation algorithm for Crystals*. The robot consists of four connected Atoms which rest on a substrate of other Atoms (such a substrate is not necessary for all locomotion algorithms). Initially, all four Atoms are fully expanded and bonded to the substrate and to each other. In the first phase of the algorithm, all but the rightmost Atom ***free*** their connection to the substrate. Next, the two middle Atoms ***contract***. This operation causes the leftmost Atom to advance to the right by one Atomic unit (the distance spanned by the size of one Atom). In the second phase of the algorithm, the leftmost Atom ***bond***s to the substrate, the rightmost Atom ***free***s from the substrate, and the two middle Atoms ***expand***. Finally, all Atoms ***bond*** to the substrate. The net effect of this algorithm is a global translation to the right of one Atomic unit for the Crystal.

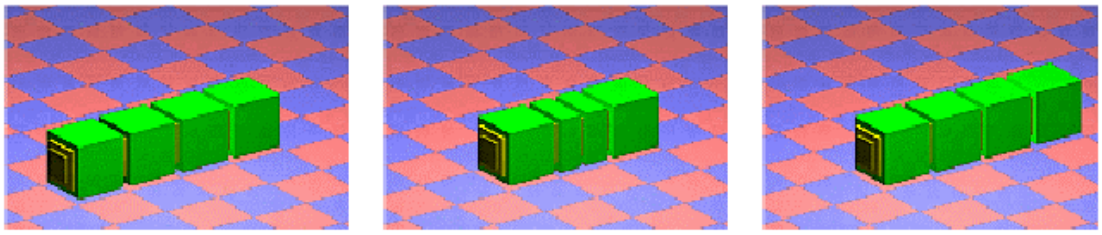


Figure 1.4: Three snapshots from a simulation, illustrating the inchworm propagation algorithm for Crystals. The left image shows the initial state of a four Atom Crystal on a planar environment of Atoms. The middle image shows an intermediate state where the two middle Atoms in the Crystal have ***contract***ed. The right image shows the final state of the Crystal after the middle Atoms have ***expand***ed. Note that the entire Crystal moved a distance of one Atomic unit to the right.

1.2.1 Materials Science Metaphor

We borrow terminology—*Atom*, *Crystal*, *Bond*, and later *Grain*—from the field of Materials Science. Our motivation is a simplified Materials Science model of the mechanism of plastic deformation in crystalline solids. This model is used to understand the Atomic-level interactions which lead to bending in metals. The Atoms in a metal arrange themselves into a regular lattice. In order for the metal to bend macroscopically (i.e. reconfigure), the Atoms from which it is composed must re-organize. Effectively, certain planes of Atoms in the crystalline lattice must slide relative to the adjacent planes. Rather than moving all the Atoms in a given plane simultaneously, a "defect" is formed at one edge of the plane so that several rows of Atoms are temporarily compressed to fit roughly in the space of a single row. The defect is then propagated across the plane of Atoms. When it reaches the opposite boundary of the plane, it relaxes again into several rows. The net effect is a translation of the entire plane. The process can be qualitatively compared to adjusting the position of a large carpet: instead of simply pulling on the carpet in the direction of the desired adjustment, a linear kink is formed at the opposite edge and pushed across the carpet.

1.2.2 Atom Actuation Variants

So far, we have enumerated the basic properties and capabilities of Atoms, but we have not precisely defined all the specifics of Atom actuation. This is because the design of Crystalline Atoms is robust with respect to the specifics of actuation: many different arrangements are possible. As long as all inter-Atomic interfaces contain a connection mechanism and as long as all dimensions of the Atom can be compressed by some constant factor, structures made of Atoms can usually be made to self-reconfigure. The two-dimensional hardware implementation of an Atom, shown in Figure 1.5 and described in Chapter 4, has only one prismatic degree of freedom which contracts both dimensions in

tandem by a factor of two, and only two connection mechanisms (since the connection mechanisms are placed on adjacent faces, and since all Atoms in a Crystal are identically oriented, then all inter-Atomic interfaces will contain one connection mechanism).

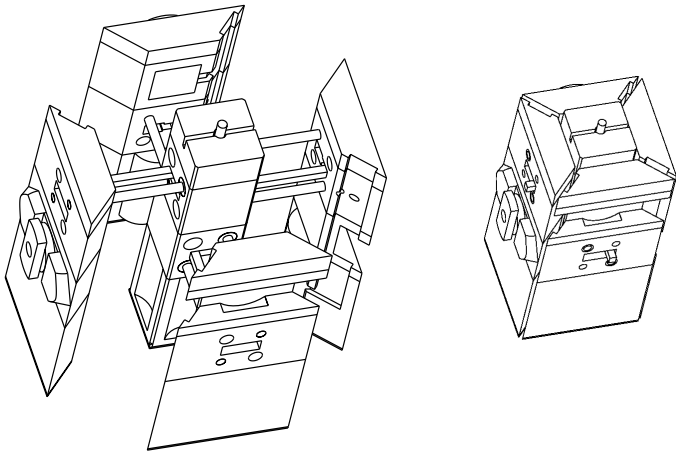


Figure 1.5: A CAD depiction of the mechanics of a two-dimensional Atom. The Atom is shown expanded at left and contracted at right. In this implementation, the Atom contains a single DOF which serves to expand and contract both dimensions in tandem by a factor of two, and two connective DOF on the upper and leftmost faces.

In contrast, the three-dimensional Atom model employed by the software simulator *xtalsim* (described in Chapter 2) contains three prismatic degrees of freedom which serve to contract each dimension independently by a factor of two, and three connection mechanisms (see Figure 1.4).

Now that we have discussed the properties of the Crystalline Atom, we describe how Crystalline Atoms can form structures, and how individual Atoms in such structures can relocate. Hence Atoms are shown to satisfy Theorem 1.1, and unit-modular systems composed of Atoms are self-reconfigurable robots.

1.2.3 Structure Formation

Since Atoms are cubic in three dimensions and square in two, they can be close packed. In such a close packing, we can ***bond*** all inter-Atomic interfaces shared by two Atoms, thus locking the group of Atoms together into a solid structure (a Crystal). We can

use such packings of Atoms to represent any shape to an arbitrary precision by manipulating the size of the Atom (this is analogous to reducing the aliasing error on a raster display by increasing the resolution of the display). Figures 1.4 and 1.6 illustrate three-dimensional Crystals. Figures 1.7 and 1.8 illustrate two-dimensional Crystals.

1.2.4 Module Relocation

An individual Atom cannot relocate without help. However, by contracting and expanding groups of Atoms in a coordinated way, Atoms can be made to relocate within and about Crystals. Importantly, we can successfully achieve such relocations in a *virtual* sense: the Atom which is made to appear in the goal location is not identical to the Atom we originally intended to move. Also, we are not restricted to motion about the surface of the Crystal because we can use the Atoms' compressibility to move Atoms directly through the volume of the Crystal.

To illustrate these ideas, we consider the example of moving an Atom from one spot to another on the surface of a Cubic Crystal, as depicted in Figure 1.6.

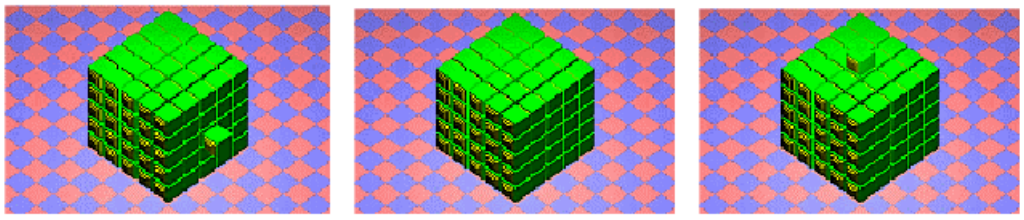


Figure 1.6: An Atom can be pushed into a Crystalline cube and popped out at any location on the surface of the cube in constant time. The three images are snapshots from a simulation. The left image shows the initial configuration (with the extra Atom located on the side face) and the right image shows the final configuration (where the extra Atom is on the top face). The middle image shows the base cube where two internal modules are compressed (not visible in this figure).

Instead of propagating the Atom along the surface of the cube in this example, which would require time linear in the side length of the cube, it is possible to reach the goal using a constant number of Atom motion primitive operations. The number of operations remains

theoretically constant no matter where the start and goal locations are oriented relative to each other and irrespective of how large the cube is. First, the Atom at the start location is pulled inside the cube by ***contract***ing two internal Atoms. 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 ***contract***ed. 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 start 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, those on the supporting line for the start location, are ***expand***ed into the void. Finally, the second pair of contracted Atoms, those on the supporting line for the goal location, are expanded in the direction of the goal location, pushing along the entire column of Atoms on that line so as to pop out an Atom in the goal location.

Collectively, we call such a sequence of Atom motion primitives a *transition*. In Figure 1.7, a transition is depicted as a sequence of two-dimensional Crystal states.

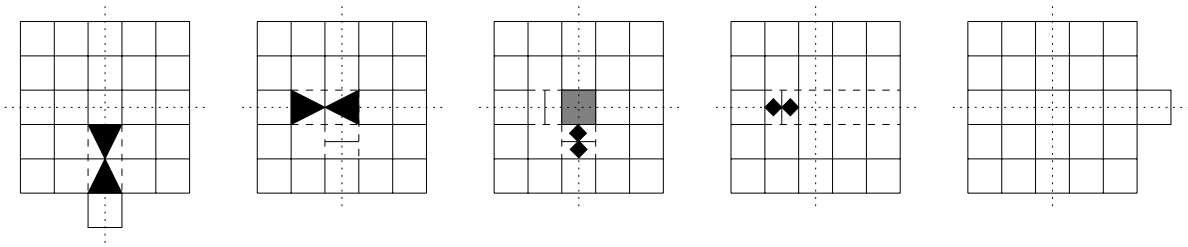


Figure 1.7: This sequence illustrates Atom relocation through the volume of a Crystal involving one transition. The leftmost image shows the initial state, with an extra Atom on the lower surface of a square Crystal. Bowties mark two Atoms about to be ***contract***ed. Dashed lines mark disconnected inter-Atomic interfaces. The supporting lines for the start and goal locations are indicated with dotted lines. The rightmost image shows the final state, where an extra Atom is present on the right side of the square. The intermediate sequence shows the formation and destruction of a void at the intersection of the supporting lines for the start and goal locations. Small dark diamonds mark two compressed Atoms about to be ***expand***ed.

If the supporting lines for the start and goal locations in a three-dimensional Crystal do not intersect, then two transitions will be required instead of one. In the two-dimensional case, the supporting lines will always intersect for convex solid Crystals.

Using this algorithm, an Atom can be relocated in theoretically constant time on any convex Crystalline substrate. We say "theoretically" because this algorithm assumes that the actuators are strong enough to pull or push any number of Atoms during these two operations. If the actuators are not strong enough an inchworm-like propagation can be used instead, but this will no longer be a constant time operation. Alternatively, parallelism can be employed in some cases to retain constant-time performance even with weak actuators.

When a Crystal is non-convex, a similar algorithm effects the Atom relocation operation in O(k)-time, where k is the number of concave angles in the Crystal. The idea is to augment the algorithm for convex substrates with additional transitions at each concave angle between the start location and the goal location.

Figure 1.8 shows the details of an example for relocating an Atom on the surface of a Crystal with two concave corners. The algorithm performs two transitions.

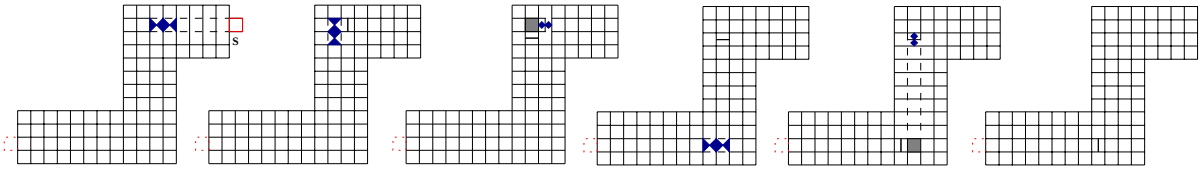


Figure 1.8: This figure illustrates an algorithm for relocating an Atom on a concave substrate, involving two transitions. The left image shows the initial configuration. The relocating Atom is in the upper right corner of the structure. The goal location is in the bottom left corner. Bowties mark two Atoms about to be ***contract***ed. Small dark diamonds mark two compressed Atoms about to be ***expand***ed. Dashed lines mark disconnected inter-Atomic interfaces. The second image shows the structure after the contraction of the first pair Atoms, and two Atoms preparing for the next contraction. The third image shows two pairs of compressed Atoms and a void. The fourth image shows the first compressed pair expanded into the void and a candidate pair of Atoms for the next contraction. The fifth image shows the state of the structure after this contraction, with the resulting void. The right-most image shows the structure after an expansion into the void. At this point, the remaining compressed pair can be expanded into the goal location.

Before defining a general and complete algorithm for Atom relocation, we will first develop a theorem which formalizes the transition operation. We define a *scrunch* to be two adjacent, connected Atoms that are contracted in the dimension normal to their connected faces. We define an *axis* to be a connected string of at least two Atoms along one dimension. Two axes intersect if they have one Atom in common. Finally, we define the *Atom Connectivity Graph* of a Crystal ***C***, ACG(***C***), to be an undirected graph whose vertices represent the Atoms in ***C*** and whose edges represent bonded inter-Atomic interfaces in ***C***. It follows that:

Theorem 1.2 *If one of two intersecting axes in a Crystal **C** contains a scrunch, a transition can be performed that transfers to the scrunch to the other axis, provided there exists sufficient surrounding structure to maintain connectedness throughout the operation.*

Proof: Call the axis initially containing the scrunch ***i*** and the axis to which the scrunch is transferred ***f***. We can assume that the scrunch in ***i*** starts out adjacent to the intersection of the supporting lines for ***i*** and ***f***, as shown in Figure 1.9.

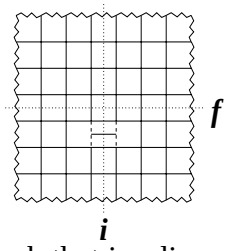


Figure 1.9: Initially, ***i*** contains a scrunch that is adjacent to the intersection of the supporting lines for ***i*** and ***f*** (shown dotted).

We first create a void at the intersection by contracting the Atom on the intersection and an adjacent Atom in ***f***, as shown in Figure 1.10.

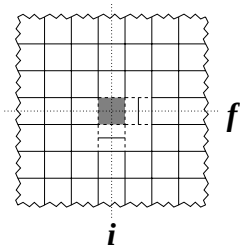


Figure 1.10: A void is created at the intersection of ***i*** and ***f*** by contracting two Atoms in ***f***.

The surrounding structure required for this operation must be such that ACG(***C***) remains a connected graph even when the dashed inter-Atomic interfaces in Figure 1.10 are free and when the Atom at the intersection of the supporting lines is removed.

Next, we expand the scrunch in ***i*** so that the void is filled, as shown in Figure 1.11.

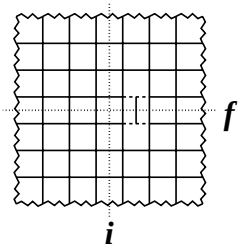


Figure 1.11: The void at the intersection of ***i*** and ***f*** is filled by expanding the scrunch in ***i***.

The surrounding structure required for this operation must be such that ACG(***C***) remains a connected graph even when the dashed inter-Atomic interfaces in Figure 1.11 are free.

The scrunch has now been transferred from ***i*** to ***f***, thus the theorem is proven for situations where the two surrounding structure requirements are met. One general way to guarantee that they are met is to only consider axes ***i*** and ***f*** which each have neighboring axes ***i‘*** and ***f‘*** on the same side (***i‘*** and ***f‘*** need to be on the same side in order to guarantee that ***i‘*** is always connected to ***f‘***):

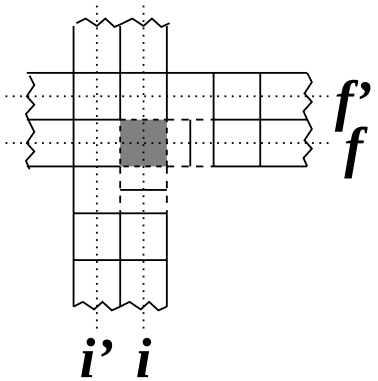


Figure 1.12: Axes ***i*** and f with neighboring axes ***i‘*** and ***f‘*** on the outside.

As an aside, we note that there are some pathological Crystals that contain a concave angle at which no pair of intersecting axes meet the surrounding structure requirements. Atom relocation may still be possible within a pathological Crystal. However, for simplicity, we choose to consider only non-pathological Crystals for this algorithm. We can do this with little loss of generality for large three-dimensional Crystals by specifying that we only consider Crystals with a minimum cross-sectional dimension of two Atom units. This guarantees that every pair of intersecting axes will have a pair of neighboring axes on the same side.

To obtain a general algorithm for relocating an Atom from a start location to a goal location on the surface of a non-pathological Crystal ***C*** the following four steps are sufficient:

Algorithm 1.1: Atom Relocation

1. Find a path with segments oriented along the principal directions from the start location to the goal location.
2. Create a scrunch along the supporting axis for the start location, pulling in the Atom that was in the starting location.
3. Drive the scrunch created in 2 along the path found in 1, performing transitions from each segment of the path to the next.
4. Relax the scrunch along the supporting axis for the goal location, popping an Atom out into the goal location.

Step 1 amounts to finding an optimal path from the start to the goal in the ACG(***C***), so it could execute in $O(n^2)$, where n is the number of Atoms in the structure. The running time of the remaining steps, which is the time required for the physical relocation to occur, is $O(t)$, where t is the number of turns in the path. Note that $t = O(k)$, with k the number of concave angles in the structure.

This Atom relocation algorithm and the structure formation property described in the previous section show that Crystalline Atomic robot systems are self-reconfigurable by Theorem 1.1.

1.2.5 Advantages

In this section, we discuss several reasons why Atoms are a particularly good choice of unit module. We cite four specific factors: non-isometric reconfiguration, volumetric transport, geometric regularity, and mechanical simplicity.

- **Non-isometric Reconfiguration** In a three-dimensional Crystalline Atomic system, the starting and final configurations in a shape metamorphosis may differ in volume by a factor as large as 8. This is true even if each configuration contains the same number of Atoms. This is done simply by manipulating the ratio of contracted to expanded Atoms in the initial and final states. For example, if every Atom in the initial state of some

shape metamorphosis is expanded and every Atom in the final state is contracted, then the shape metamorphosis included a volumetric dilation of 8x.

- **Volumetric Transport** As illustrated above, Atom relocation can take place directly through the volume of a Crystal, and in most cases literal relocations are equivalent to much faster virtual relocations (where the goal state is achieved but the goal Atom is not identical to the start Atom). In most other self-reconfigurable systems that have been proposed, module relocation is restricted to the surface of the structure. This can lead to much faster total reconfiguration times, since individual Atoms can take shorter paths to reach their destination than modules in other systems.
- **Geometric Regularity** Since Atoms are simple cubes or squares, they are easily packed into arbitrary shapes. This is not the case for some other self-reconfigurable systems that have been proposed.
- **Mechanical Simplicity** Atoms can be constructed with as few as three degrees of freedom. This is in contrast to some other self-reconfigurable systems that have been proposed, which include as many as 15 DOF per module.

1.3 Project Outline

The rest of this thesis focuses on three segments of research which attempt to determine whether the Atom is also a good choice from a practical standpoint. These are: simulation, automated planning, and Atom hardware implementation.

- **Simulation (Chapter 2)** A powerful software simulator for Crystalline Atomic robots in two and three dimensions, called *xtalsim*, is presented. Xtalsim includes a high-level language interface for specifying reconfigurations, an engine which expands implicit reconfiguration plans into explicit Crystal state sequences, and an interactive animator which displays the results in a virtual environment.

- **Automated Planning (Chapter 3)** An automated planning algorithm for generating reconfigurations, called the *Melt-Grow* planner, is described. The Melt-Grow planner is fast ($O(n^2)$ for Crystals of n Atoms) and complete for a fully general subset of Crystals. The Melt-Grow planner is implemented and interfaced to xtalsim, and an automatically planned reconfiguration is simulated.
- **Atom Hardware Implementation (Chapter 4)** The mechanics, electronics, and software for an Atom implementation are developed. Two Atoms are constructed and experiments are performed which indicate that, with some hardware improvements, an interesting self-reconfiguration could be demonstrated by a group of Atoms.

1.4 Related Work

There have been a number of projects in the recent past that address aspects of self-reconfigurable robotics. This work includes robots in which modules are reconfigurable using external intervention, e.g. [5]. In [6], Fukuda et. al. propose a cellular robotic system to coordinate a set of specialized modules. Several specialized modules and ways of composing them are proposed.

In [30], Yim studies multiple modes of locomotion that are achieved by manually composing a few basic elements in different ways. This work also presents extensive examples of locomotion and reconfiguration in simulation. Development is currently underway to convert the system in [30] from manual reconfigurability to self-reconfigurability (see Figure 1.13).

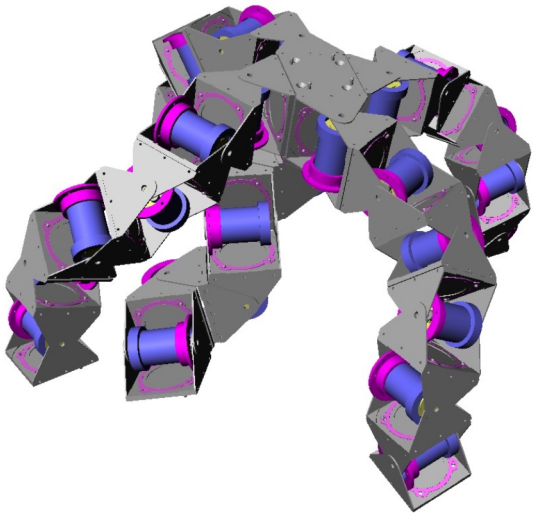


Figure 1.13: An image from a simulation of an extension of the robot presented in [30], currently under development, which will be self-reconfigurable.

In [19, 31, 29, 20], 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 [21] this group developed a twelve DOF module capable of three-dimensional motion (see Figure 1.14).

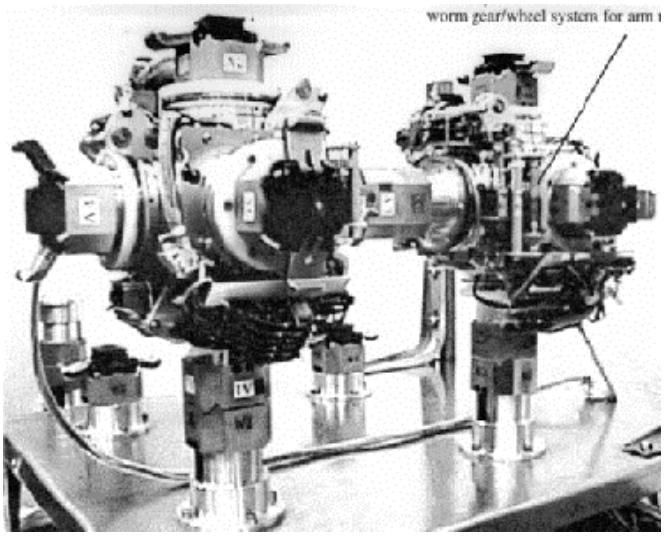


Figure 1.14: Two modules as presented in [21].

In [24], Chirikjian et. al. describe metamorphic robots that can aggregate as two-dimensional structures with varying geometry. The modules are deformable hexagons (see

Figure 1.15). This work also examines theoretical bounds for planning the self-reconfiguring motion of such modules.

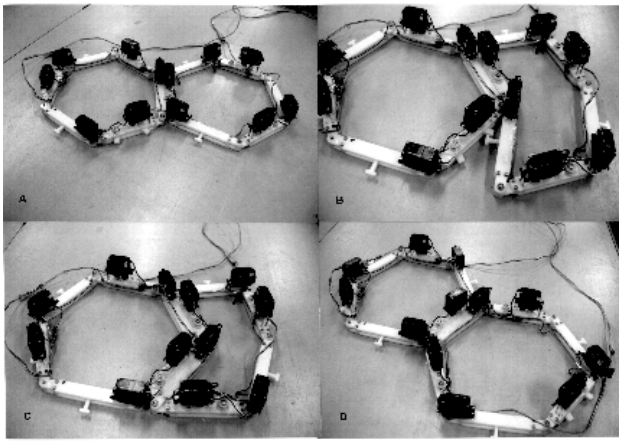


Figure 1.15: A motion sequence showing the operation of two modules as presented in [24].

In [18] we have shown a constant-time reduction between robotic molecule structures our group at DRL has designed to support self-reconfiguration [14, 15, 16] (see Figure 1.16) and metamorphic robots [24].

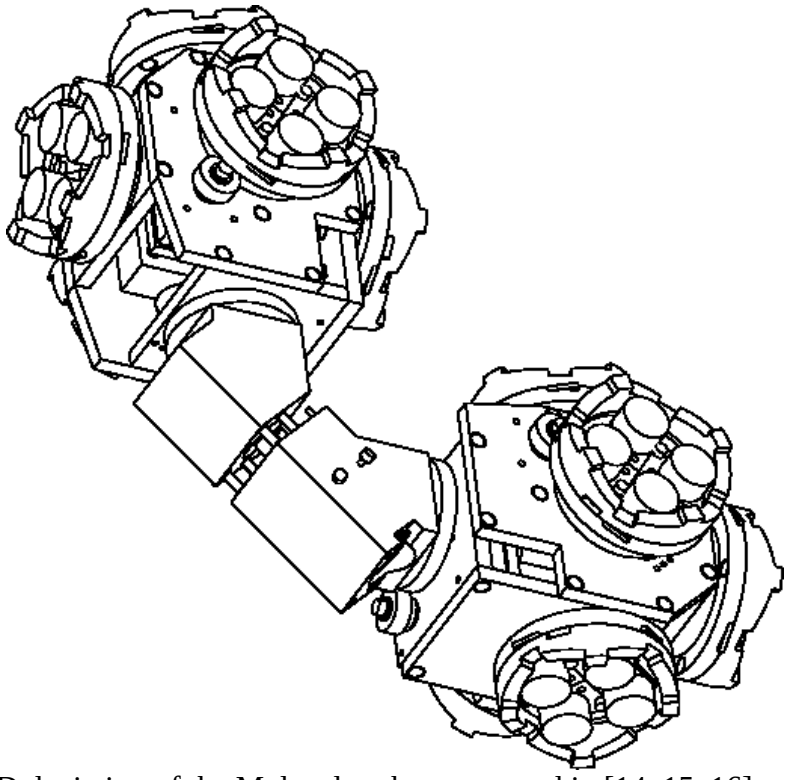


Figure 1.16: A CAD depiction of the Molecule robot presented in [14, 15, 16].

The robot proposed in this thesis 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 [28] (see Figure 1.17).

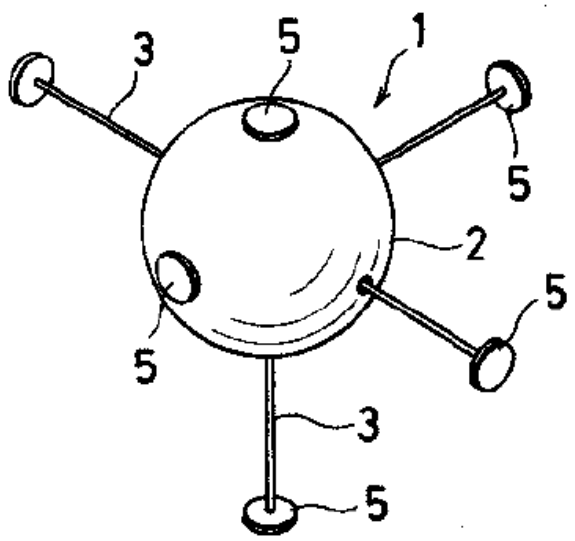


Figure 1.17: A schematic depiction of one module as proposed in [28]. This module functions by selectively retracting the small disks towards the spherical core.

2 Simulation

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 two-dimensional hardware implementation described in Chapter 4, *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 separate components, each of which is implemented as a separate program: the simulation engine *xtalexp*, and the interactive display animator *xtalanim*. *Xtalexp* accepts a simulation script called a *relative deformation*, specified in the language described below, and produces a list of explicit Crystal states called an *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: *xtalexp* is entirely concerned with the logical model of the Crystalline Atomic system, and *xtalanim* is focused on the user-interface and rendering operations.

Figure 2.1 shows several snapshots from a simulation where a three-dimensional "dog" shape is metamorphosed into a "couch" shape.

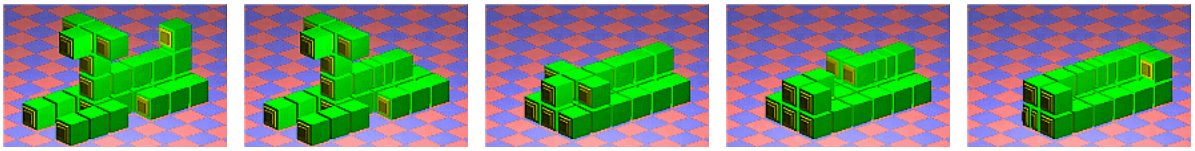


Figure 2.1: Five snapshots from a simulation of a three-dimensional Crystalline Atomic robot. The initial configuration (on the left) is a dog-shaped object. The final configuration (on the right) is a couch-shaped object. The middle images show intermediate steps in the transformation from dog to couch. The planning for this transformation was done manually.

The reconfiguration in Figure 2.1 was manually planned, and the *xtalsim* input code which specifies it is presented in Appendix 4. *Xtalsim* has also been used to simulate several other manually planned reconfigurations, for example the inchworm locomotion algorithm shown in Figure 1.4. Additionally, *xtalsim* been interfaced to our implementation of the Melt-Grow automated planner, as described in Chapter 3.

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 in Chapter 3, which contains 176 Atoms and was planned and simulated in several minutes.

2.1 System Model

As we can see in Figure 2.1, *xtalsim* maintains a fully three-dimensional Atom model. Atoms are idealized as rectangular prisms. Atoms in *xtalsim* can expand and contract independently in each dimension (*x*, *y*, and *z*). The contraction ratio is fixed at two. Atoms are modeled to have passive connection mechanisms on three faces (+*x*, +*y*, and +*z*) and active connection mechanisms on the other three faces (-*x*, -*y*, and -*z*). The active connectors are depicted in the animator (see Figure 2.l) as narrow, light-colored protrusions on three faces of each Atom. Each connector can be independently connected (***bond***ed) and disconnected (***free***d). All degrees of freedom are binary, so any given Atom is always either expanded or contracted in each dimension and either bonded or free at each negative face. All actions are logically modeled as discrete, instantaneous events (i.e. it takes no time to ***expand*** or ***contract***, ***bond*** or ***free***), so only the order of specified actions is significant.

Atoms live in an absolute 3D Cartesian coordinate system with origin (0, 0, 0). The orientation of the coordinate system is such that if *x* is positive right and *y* is positive up, *z* is

positive out of the screen. Atoms are identified by their centroid coordinates, which are always integral. When expanded in a dimension an Atom occupies exactly four units in that dimension; when contracted in a dimension the Atom occupies exactly two units in that dimension. Atoms never change their orientation.

The plane whose normal is the y axis and whose *y* intercept is -2 has special significance—it is called the ground plane. The ground plane is modeled as a fully connected plane of 25 x 25 invisible, expanded Atoms with centroids at $y = -4$. The ground plane establishes a stable reference point during reconfigurations: its absolute position is constant, and all Atom motions are calculated relative to it.

An important aspect of the simulation environment is that only a single Crystal is allowed (i.e. multiple disparate robots are not supported, unless they are all independently connected to the ground). Formally, only a Crystal $\boldsymbol{C}$ where ACG($\boldsymbol{C}$) is a connected graph is allowed (ACG stands for "Atom Connectivity Graph" and is introduced in Chapter 1). The invisible Atoms that make up the ground plane are considered to be part of the Crystal, so as a consequence all other Atoms must be connected through some path in the ACG to the ground plane. If any specified simulation includes actions which cause the ACG to become disconnected, this is flagged as an error and the simulation is aborted. We call such a scenario *fragmentation*.

Another important assumption of the simulation environment is that Atom movements are separated into two types: expansion actions and connection actions. Simulations are divided into sequences of discrete *updates*, and all actions within each update are applied simultaneously. However, only one type of action is allowed per update, so *xtalsim* is not able to simulate one Atom undergoing a ***bond*** action at the same instant that another Atom undergoes an ***expand*** action. This separation is by design: it simplifies the internal algorithms of the simulator and avoids ambiguous simulation specifications without

imposing significant restrictions on the types of simulations that can be run (since all actions are modeled to occur as discrete instantaneous events anyway).

2.2 Input Language

Having described the environment model employed in *xtalsim*, we are ready to examine the language used to specify simulations. An example of a simulation script written in this input language is included in Appendix 4. As mentioned above, the input language specifies a *relative deformation* to *xtalexp*, and *xtalexp* creates a corresponding *absolute deformation* for *xtalanim* (i.e. *xtalexp exp*ands the deformation). These names reflect the idea that, in an input relative deformation, only the specific ***bond***, ***free***, ***expand***, and ***contract*** actions are specified for each update. Even though these actions may have the effect of causing many other Atoms to re-locate, such movements are not explicitly specified in the input. Rather, they are automatically extrapolated by *xtalexp*. The full result of each update is calculated by *xtalexp* as a complete Crystal state which specifies the conditions of every Atom in the Crystal. The ordered sequence of such Crystal states which result from all the updates in a simulation form the absolute deformation that *xtalexp* produces.

2.2.1 Relative Deformation Grammar

Relative deformations are specified as human-readable text files. The C preprocessor is employed, so C-style /* comments */, #directives, and macros can be used. The relative deformation includes two components: the initial Crystal specification, which lays out the position and state of each Atom in the initial configuration, and the update list, which specifies the sequence of updates to be applied which effect the reconfiguration. So the top-level relative deformation grammar production rule is as follows:

relative_deformation := ‘(‘ *initial_xtal* ‘(‘ *update** ’)’ ‘)’

The initial Crystal is a list of Atom specifications which identify the location and state of each Atom in the simulation:

initial_xtal := ‘(‘ *atom** ‘)’
atom := ‘(‘ ‘atom’ ‘(‘ *x y z* ‘)’ *conn_x conn_y conn_z exp_x exp_y exp_z* ‘)’
*conn_*{*xyz*} := ‘b’ | ‘f’
*exp_*{*xyz*} := ‘e’ | ‘c’

In order to avoid fragmentation, the initial Crystal must be fully connected. Since the environment ground Atoms are implied to be part of the Crystal, the initial Crystal specified in the relative deformation must also include a connection to the ground plane.

The second part of the relative deformation, the update list, is an ordered sequence of expansion and connection updates:

update := *exp_update* | *conn_update*
exp_update := ‘(‘ *exp_action** ‘)’
conn_update := ‘(‘ *conn_action** ‘)’

Expansion and connection actions are the four familiar Atom actions that were introduced in Chapter 1:

exp_action := *expand_action* | *contract_action*
expand_action := ‘(‘ ‘expand’ *atom_id dimension* ‘)’
contract_action := ‘(‘ ‘contract’ *atom_id dimension* ‘)’

conn_action := *bond_action* | *free_action*
bond_action := ‘(‘ ‘bond’ *atom_id dimension* ‘)’
free_action := ‘(‘ ‘free’ *atom_id dimension* ‘)’

dimension := ‘x’ | ‘y’ | ‘z’

The grammar rules above are complete except for the definition of one non-terminal symbol: *atom_id*. *atom_id* are used to specify which Atom or group of Atoms to which each action should apply. The top-level grammar for *atom_id* is as follows:

atom_id := *range* | *relative*

An *atom_id* can either refer to an (absolute) *range* of Atoms, or it can refer to a group of Atoms *relative* to some other Atom. First we consider the *range* grammar:

range := ‘<’ *x_dim_range y_dim_range z_dim_range* [*conn_spec*] [*exp_spec*] ‘>’
{*x,y,z*}*_dim_range* := ‘*’ | *constant* | *% min max %* | *% min*[*,incr*] *max %*
constant,incr := **integer**
min, max := **integer** | ‘inf’
% := **parenthesis** | **square bracket**
conn_spec := { ‘b’ | ‘f’ | ‘?’ }^{3}
exp_spec := { ‘e’ | ‘c’ | ‘?’ }^{3}

range is used to specify a group of Atoms whose centroid coordinates fall within some range and whose connection and expansion state match a specified template. Wildcards are provided for all fields, and the semantics of each dimension range can be precisely specified. When evaluating a range, all matching Atoms are selected.

The *relative* specification contains both an *atom_id* and a *range*:

relative := ‘<’ ‘@’ *atom_id range* ‘>’

The *atom_id* in a relative specification is called the *anchor* and the range is called the *follower*. A relative specification differs from the range specification which it includes only (and importantly) in that the coordinates in the follower are relative to the centroid coordinates of the anchor. If the anchor resolves to more than one Atom then the follower is evaluated for each of these.

It is an error to specify an *atom_id* that resolves to the same Atom more than once. Specifying an *atom_id* that resolves to no Atoms generates a warning but is not an error. Anywhere a literal integer is expected a {curly brace} enclosed C-style integer expression involving +-*/(){} is also allowed.

Here is an example of a moderately complex *atom_id*:

<@ <0 {8/2} 0> <(0,4 inf] 0 0 b??>>

This is a relative specification. The anchor is the range <0 {8/2} 0>, which simply selects the Atom at (0, 4, 0) independent of its connection and expansion state (since the connection and expansion specs are omitted). The follower is the range <(0,4 inf] 0 0 b??>, which

selects all Atoms along a line extending in the positive *x* direction from the anchor whose centroid *x* coordinate is a multiple of 4 and whose -*x* active connector is bonded, not including the anchor itself.

2.3 The xtalexp Simulation Engine

The relative deformation input language processor is the front end to the main component of *xtalexp*, which is the simulation engine. The simulation engine maintains a logical model of the simulated Crystal. The logical model is originally set to the initial Crystal specification in the relative deformation. Each update specified in the relative deformation is then applied in sequence to the model. An absolute Crystal is written to the output (and thence to *xtalanim*) before application of each update. We can summarize these steps as the following algorithm:

Algorithm 2.1: *xtalexp* Simulation

1. Setup the Crystal model according to the initial Crystal specification in the relative deformation
2. Process the updates contained in the relative deformation in order

We'll examine the functionality of the simulation engine in three parts: initialization, connection updates, and expansion updates.

2.3.1 Initialization

Setup of the Crystal logical model is a relatively simple operation. Atom objects are created for each Atom specified in the initial Crystal specification part of the relative deformation. These Atom objects, along with hidden Atom objects which represent the ground plane, are then linked together into two data structures: a flat list, and an interlinked graph data structure corresponding to the ACG (we will loosely refer to this latter data structure simply as the ACG). Two consistency checks are performed at this point: a

collision check and a fragmentation check. The collision check verifies that no Atoms are specified to physically intersect with each other. The fragmentation check verifies that the ACG is a connected graph. If either of these checks fail the simulation is aborted.

2.3.2 Connection Updates

Once the Crystal logical model has been set up, *xtalexp* begins to process the updates specified in the latter half of the relative deformation. As described above, these will either contain ***expand***/***contract*** actions exclusively or ***bond***/***free*** actions exclusively. The latter is the simpler case, so we will describe it first.

All ***bond*** and ***free*** actions specified in a connection update are simulated to occur instantaneously and simultaneously. An atom is allowed to ***bond*** one of its active connectors iff both of the following are met:

1. the active connector is currently free (disconnected)
2. there is a neighbor Atom with a passive connector correctly positioned to mate (ground plane Atoms are valid neighbors)

An Atom is allowed to ***free*** one of its active connectors iff both of the following are met:

1. the active connector is currently bonded
2. ***free***ing the active connector will not cause the Crystal to fragment

Violation of any of the above, with one exception, is considered an error and causes the simulation to be aborted. The exception is a ***bond*** request with no neighbor present, which generates a warning and has no effect, but is not considered an error.

A fragmentation consistency check is run after processing each connection update to verify that the Crystal has not become disconnected (if it has, the simulation is aborted).

2.3.3 Expansion Updates

As in connection updates, all **expand**/**contract** actions specified in an expansion update are performed simultaneously and instantaneously. This needs to be done carefully because the actions of individual Atoms may compound each other, as shown in Figure 2.2.

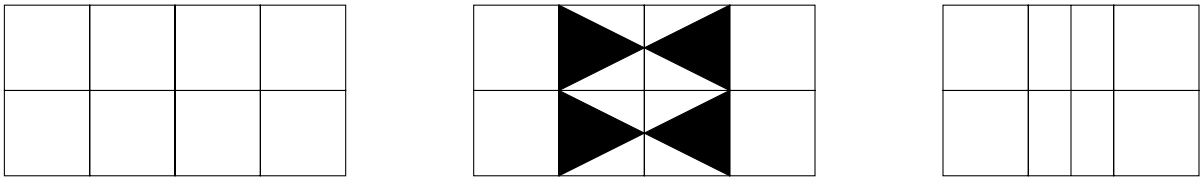


Figure 2.2: An expansion update that includes a compound movement. The four central Atoms are contracted in unison, pulling the two external columns of Atoms together by a total of one Atom unit.

Two erroneous situations also need to be accounted for: movements that cause Atoms to crash into each other, and movements that are not feasible because they specify movement in Atoms which are otherwise fixed. These two situations are illustrated in Figures 2.3 and 2.4, respectively.

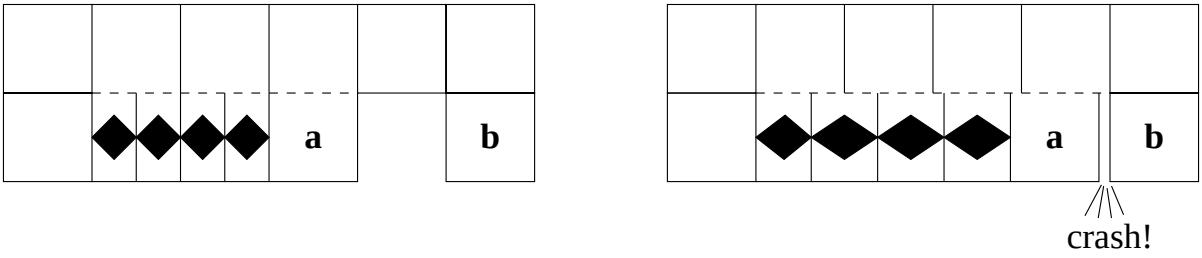


Figure 2.3: An expansion update that causes Atom **a** to crash in to Atom **b**.

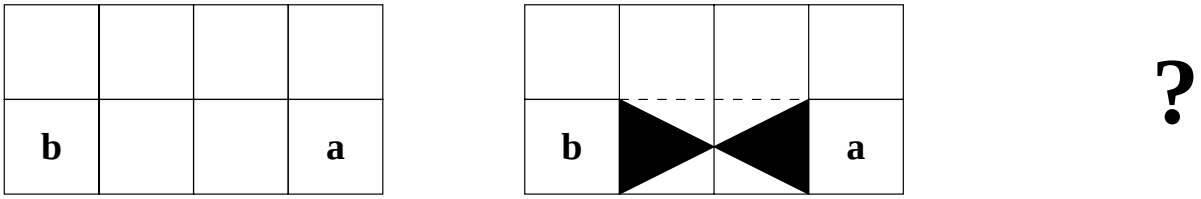


Figure 2.4: An expansion update which specifies Atom **a** to move to the left even though it is otherwise fixed (or is it that Atom **b** should move to the right?).

Xtalexp implements an algorithm to process expansion updates that can handle all of the above cases. Before we present it we need to define a few terms: A *segment* is a maximal connected component of Atoms in a Crystal, all of which will translate by the same distance as a result of the expansion update. The *d axis of a* is the connected column of

Atoms containing Atom *a* and extending in both directions along dimension *d*. An *active face* in a dimension *d* is either half (i.e. active or passive face) of a bonded inter-Atomic interface between an Atom which is specified to ***expand*** or ***contract*** in *d* and a neighboring Atom in the Crystal.

The main idea of the algorithm is to assign all the Atoms in the Crystal to segments, each of which will move as solid blocks. Since the Crystal must be connected to the ground plane, we can assign the segment that contains the ground plane Atoms (the *ground segment*) a net translation of 0 units in all dimensions. We then calculate the displacement of all remaining segments in each dimension relative to the ground segment. These steps can be summarized as the following algorithm:

Algorithm 2.2: *xtalexp* Expansion Update

1. Assign every Atom in the Crystal to a segment so that all Atoms within each segment displace by the same amount
2. Verify that the specified expansion update is feasible
3. Calculate the displacement for each segment based on the specified Atom ***expand*** and ***contract*** actions

The pseudocode for Algorithm 2.2 is as follows:

C is the Crystal in its current state
a.action.*d* is any specified action (***expand***/***contract***) for Atom *a* in dimension *d*
a.segment is the segment assigned to Atom *a*
a.neighbors is the set of Atoms adjacent to *a* in the ACG
displacement[*s*].*d* is the calculated displacement for segment *s* in dimension *d*
g is a ground Atom
Size(*a, d*) is the size of Atom *a* in dimension *d* (4 if *a* is expanded in *d*, 2 if *a* is contracted)

```
DoExpansionUpdate(Crystal C)
    FOR EACH Atom a in C
        a.segment ← unassigned
    AssignSegments(C)
    CheckFeasibility(C)
    FOR EACH Segment s
        FOR EACH Dimension d
            displacement[s].d ← undefined
    FOR EACH Dimension d
        displacement[0].d ← 0
        CalculateDisplacements(C, d)
    RETURN displacement

AssignSegments(Crystal C)
    Integer next ← 0
    SetSegment(C, g, next)
    WHILE there exists an Atom a in C where a.segment = unassigned
        next ← next + 1
        SetSegment(C, a, next)

SetSegment(Crystal C, Atom a, Segment s)
    a.segment ← s
    FOR EACH n in a.neighbors
        IF the interface between a and n does not contain an active face THEN
            IF n.segment = unassigned THEN
                SetSegment(C, n, s)

CheckFeasibility(Crystal C)
    FOR EACH Atom a in C
        FOR EACH Atom n in a.neighbors
            IF the interface between a and n contains an active face THEN
                IF a.segment = n.segment THEN
                    ABORT

CalculateDisplacements(Crystal C, Dimension d)
    WHILE there exists an Atom m in C where m.action.d ≠ null
        Axis x ← the d axis of m
        IF there exists an Atom a in x where a.segment ≠ undefined
            CalculateAxisOffsets(x, a, d)
            m.action.d ← null

CalculateAxisOffsets(Axis x, Atom a, Dimension d)
    FOR EACH Direction i
        Integer shift ← HalfSize(a, d)
        FOR EACH Atom m along the i direction in x, in order and starting from a
            shift ← shift + HalfSize(m, d)
            abs_shift ← displacement[a.segment].d + i*shift
            IF displacement[m.segment].d ≠ undefined THEN
```

```
                IF displacement[m.segment].d ≠ abs_shift THEN
                    ABORT
            ELSE
                displacement[m.segment].d ← abs_shift
            shift ← shift + HalfSize(m, d)
            m.action.d ← null

HalfSize(Atom a, Dimension d)
    IF a.action.d = expand THEN
        RETURN 2
    IF a.action.d = contract THEN
        RETURN 1
    ELSE
        RETURN (1/2)*(Size(a, d))
```

Correctness

The following four items are sufficient to demonstrate the correctness of the **DoExpansionUpdate** algorithm:

1. ***AssignSegments*** *will assign every Atom in* ***C*** *to a segment*: **AssignSegments** runs until all Atoms have been assigned, so as long as **AssignSegments** terminates this condition will be satisfied. **AssignSegments** simply performs a DFS through ACG(***C***) (bounded by active interfaces) for (up to) each Atom in ***C***, so **AssignSegments** will always terminate.
2. *Every Atom within a segment will displace by the same amount*: Each segment is built so that it will not contain two Atoms connected through an active interface (this is guaranteed in **CheckFeasibility**). Thus, there is a path from every Atom in a segment to every other Atom in that segment which consists of edges in ACG(***C***) that correspond only to non-active interfaces. Since the centroid-to-centroid distance of two Atoms connected through a non-active interface is constant, the centroid-to-centroid distances between all pairs of Atoms in a segment will remain constant. So if any Atom within a segment is going to move, all the other Atoms in that segment must move by the same amount.

3. ***CalculateAxisOffsets*** *will shift each segment by the correct distance*: **CalculateAxisOffsets** is given an Atom in the axis whose absolute displacement is already known. The absolute displacements of all the other Atoms in the axis can always be calculated relative to that known displacement because the actions of all these Atoms are defined. A disagreement between the absolute displacement calculated for a segment along one axis and the displacement calculated for that segment along any other axis indicates an infeasible update (i.e. an error in the relative deformation), and such situations are caught by **CalculateAxisOffsets**.

4. *Every Atom that was specified to* ***expand*** *or* ***contract*** *(i.e. every active Atom) will be processed:* **CalculateDisplacements** runs until all active Atoms have been processed, so this will be satisfied as long as **CalculateDisplacements** always terminates. We can prove that **CalculateDisplacements** always terminates by induction: In the base case, only the ground segment displacement has been defined. If there are no other segments, then we are done because this means there are no active Atoms. If there are other segments, then one of them must be connected to the ground segment through some active interface (because the ACG is always connected). So the axis normal to this interface will contain an active Atom that we can process. For the inductive step, we first observe that each time we process an active Atom we define the displacement for at least one more segment (that containing the active Atom). Then we observe that there will always be at least one active Atom that we can process: Since the ACG is always connected, there must always be a path from every Atom in every segment to every other Atom in every other segment. Such paths will always contain edges which correspond to active interfaces at the boundaries between segments. If we pick any segment whose displacement is not yet defined, then we can follow a path from some Atom in this segment to some Atom in some segment whose displacement *has* already been defined.

The last segment we pass through along such a path (before we hit the segment with defined displacement) will be a segment with an undefined displacement that is connected through an active interface to the final segment. The axis normal to this interface will contain an active Atom that we can process.

Running Time

If the size of Crystal ***C*** is n Atoms, then the total running time for **DoExpansionUpdate** is $O(n^3)$. We can demonstrate this most easily from the bottom up, starting with **HalfSize**, which is O(1). Since **CalculateAxisOffsets** processes each Atom in the axis once and since the number of Atoms in any axis must be $O(n)$, **CalculateAxisOffsets** will run in $O(n)$. **CalculateDisplacements** will run in $O(n^3)$ time since it does two nested Atom searches ($O(n)$ each) and at the innermost level it calls **CalculateAxisOffsets**. **CheckFeasibility** clearly runs in $O(n)$ since it processes each Atom once. **SetSegment** amounts to a DFS in ACG(***C***), so its running time is also $O(n)$. Since **AssignSegments** does a linear Atom search and then calls **SetSegment** at the inner level, it will run in $O(n^2)$. Finally, we can consider the top-level **DoEpxansionUpdate** as a sequence of several operations, the most time consuming of which is **CalculateDisplacements** ($O(n^3)$), which it calls for each dimension. Since the number of dimensions is always constant (in the simulator it's 3), **DoExpansionUpdate** will run in $O(n^3)$ time.

2.4 The xtalanim Interactive Display Animator

After all that analysis, it will be a welcome relief to look at a picture again. Figure 2.5 is a screenshot of the GUI that *xtalanim* presents.

Controls are provided in *xtalanim* to

- interactively change the three-dimensional viewpoint of the simulation
- start, stop, and single-step the simulation
- change the speed of a running simulation
- simplify the appearance of the simulated system by hiding the inter-atom connectors or the ground plane or by specifying a wireframe rendering
- save a snapshot of the current view to a file
- automatically save snapshots of every view in an animation to a sequence of files

Xtalanim is a relatively simple program which graphically displays the absolute Crystal sequences provided by *xtalexp*. *Xtalanim* uses Open-GL to handle rendering and TCL/TK for the user interface.

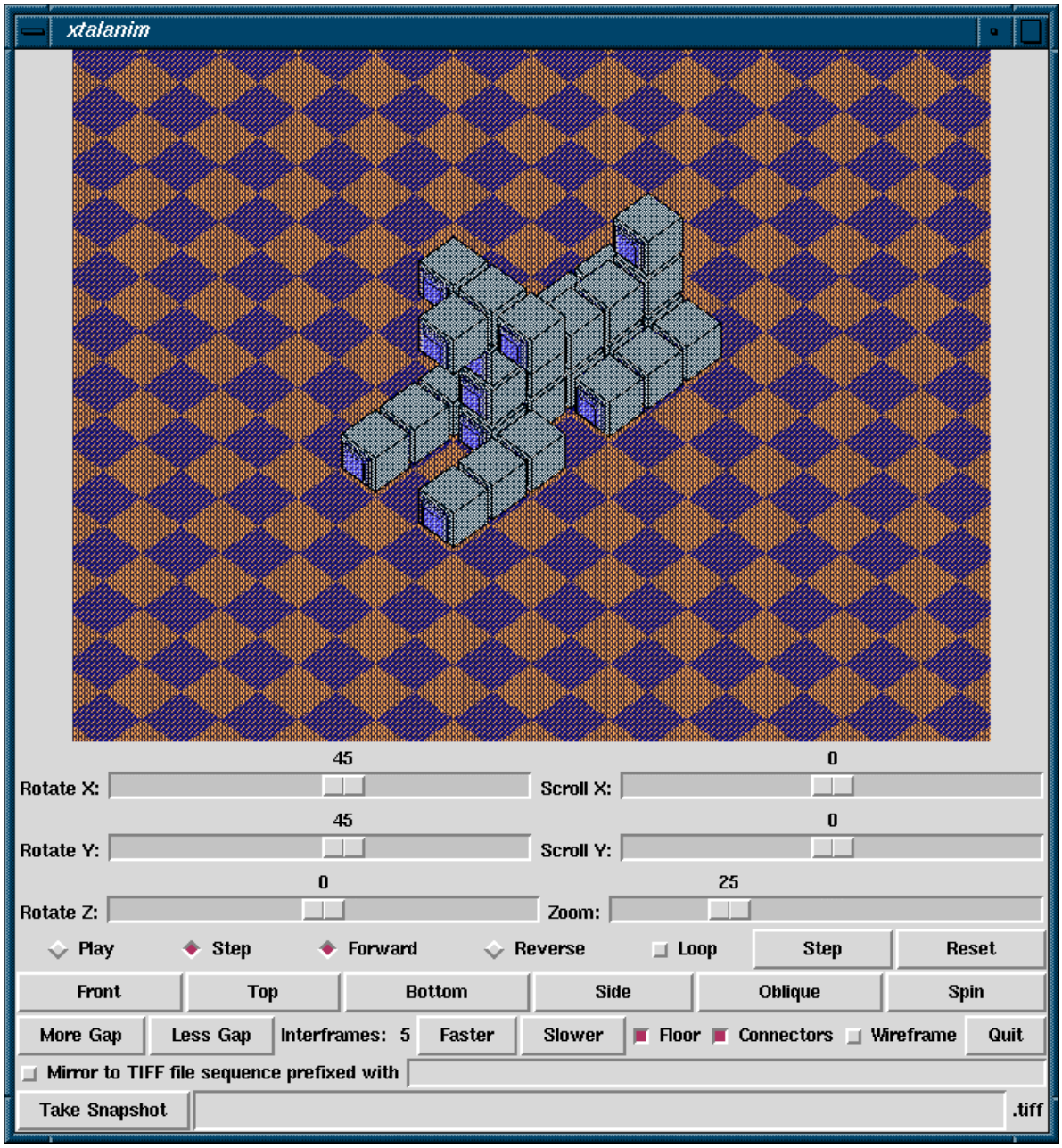


Figure 2.5: A screenshot of the *xtalanim* GUI.

3 Automated Planning

In this Chapter we describe a planner for self-reconfiguration in Crystalline Atomic robot systems. The reconfigurations we have seen so far (e.g. the inchworm in Figure 1.4 and the dogcouch in Figure 2.1) have all been manually planned. That is, a human generated the sequence of Atom motion primitives for each Atom at each stage of the reconfiguration. We would like to avoid this, if possible, and have a computer automatically determine all Atom motion primitives for all stages of any given reconfiguration.

A reconfiguration plan is a partially ordered sequence of Atom motion primitives. A reconfiguration plan is *feasible* iff at no time during the execution of the plan does the Crystal become disconnected or crash into itself (see Figures 2.3 and 2.4). With these definitions, we can define the planning problem for reconfiguration in Crystalline Atomic robots more precisely: given a pair of crystals (***S***, ***G***), each composed of *n* Atoms, find a feasible reconfiguration plan **P** that transforms ***S*** into ***G***.

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.

We have developed a centralized planning algorithm called the *Melt-Grow* planner that is complete over a useful subset *Grain*(4) of Crystals and which can run in $O(n^2)$ time, where *n* is the number of Atoms in the Crystal:

Algorithm 3.1: Melt-Grow

1. **Melt *S*** into an intermediate Crystal ***I***
2. **Grow *G*** out of ***I***

3.1 Grained 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 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 such blocks intersect only at block edges and corners. We call each such block a *Grain*, in keeping with our Materials Science terminology metaphor (see Chapter 1). Figure 3.1 gives an example of a 2D Crystal in *Grain*(4) and a 2D Crystal not in *Grain*(4).

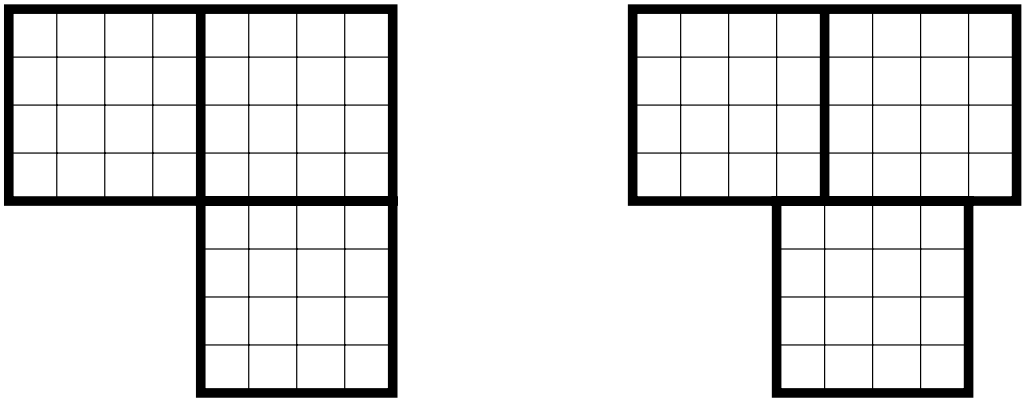


Figure 3.1: Left: a 2D Crystal in *Grain*(4). Right: a 2D Crystal not in *Grain*(4).

We can 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 1.8 and Theorem 1.2). 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>, <$dimension_1$>, <$sense_1$>, <$dimension_2$>, <$sense_2$>) - move 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 3.2.

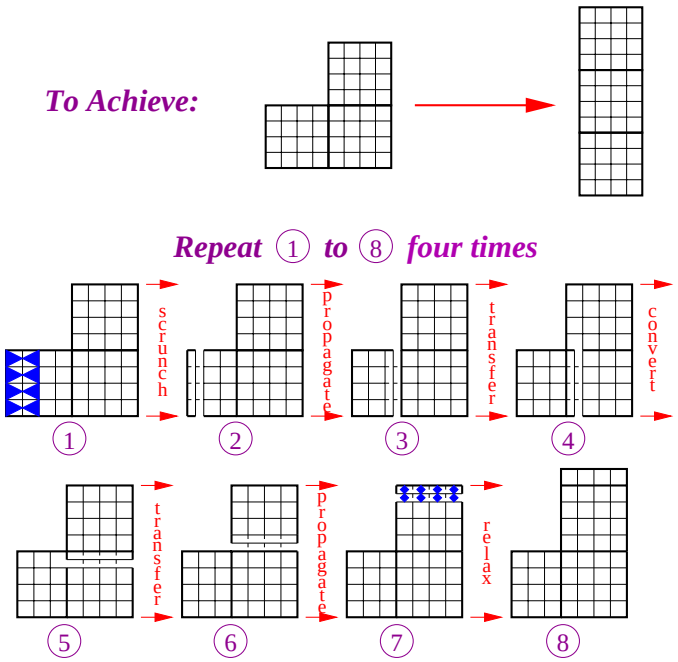


Figure 3.2: 8 steps of a Grain relocation sequence built from the five Grain motion primitives. At the top of the figure the desired Grain relocation is indicated. These 8 steps are repeated four times to complete the relocation.

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 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 3.3.

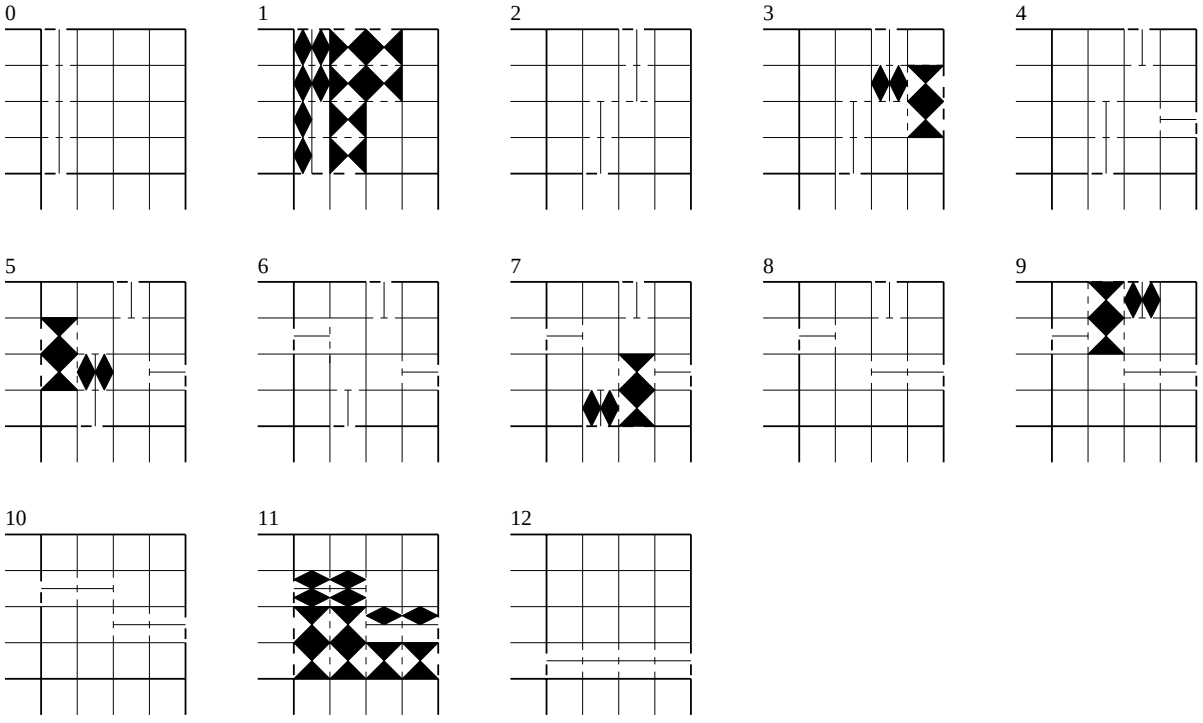


Figure 3.3: The convert operation can be carried out in 12 steps. Note that the algorithm is symmetric; the Grain state after the sixth step is a rotationally symmetric midpoint, and the remaining six steps are simply the reverse of the first six applied in the normal dimension.

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

3.2 Details and Analysis of 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, the Melt algorithm 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, the Grow algorithm 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*** (though in 2D systems stability is often less of a concern).

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. 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***.

Now we present pseudocode for the Melt-Grow algorithm (all Crystals involved are assumed to be in *Grain*(4)):

g_dim is the vertical dimension (i.e. the dimension in which Gravity has an effect)
Volume(***S***) returns the number of Grains in Crystal ***S***
GMPExecute(GMPList **L**, Crystal ***C***) exectutes the Grain Motion Primitive List (GMPList) **L** in Crystal ***C***

Melt-Grow(Crystal ***S***, Crystal ***G***)
    Grain *stem* ← **LocateStem**(***S***)
    Crystal ***I*** ← **Melt**(***S***, *stem*)
    Grow(***I***, *stem*, ***G***)

Melt(Crystal ***S***, Grain *stem*)
    Crystal ***I*** ← {*stem*}
    Crystal ***C*** ← **DesignIntermediate**(**Volume**(***S***), *stem*)
    WHILE S ≠ {*stem*}
        Grain *mover* ← **FindMobile**(***S***)
        Grain *parent* ← **FindParent**(***I***, ***C***)
        Transport(*mover*, *parent*, ***S*** ∪ ***I***)
    RETURN ***I***

Grow(Crystal ***I***, Grain *stem*, Crystal ***G***)
    Crystal ***C*** ← *stem*
    WHILE ***I*** ≠ {*stem*}
        Grain *mover* ← **FindMobile**(***I***)
        Grain *parent* ← **FindParent**(***C***, ***G***)
        Transport(*mover*, *parent*, ***C*** ∪ ***I***)

LocateStem(Crystal ***C***)
    RETURN the Atom in C with the lowest *g_dim* coordinate

DesignIntermediate(Integer *vol*, Grain *stem*)

[RETURN a Crystal containing *stem* with volume *vol* and in which every Grain other than *stem* itself has a *g_dim* coordinate less than that of *stem* (there are several choices for such a Crystal, see text)]

FindMobile(Crystal ***C***)
[RETURN a Grain in ***C*** corresponding to a vertex in GCG(***C***) that is not an articulation point]

FindParent(Crystal ***actual***, Crystal ***skeleton***)
[RETURN a Grain in ***actual*** that is adjacent to a Grain that is in ***skeleton*** but that is not in yet in ***actual***]

Transport(Grain *mover*, Grain *parent*, Crystal ***C***)
Path **P** ← A path from *mover* to *parent* in GCG(***C***)
GMPList **L** ← **GMPDecompose**(**P**, ***C***)
REPEAT 4 times
GMPExecute(**L**, ***C***)

GMPDecompose(Path **P**, Crystal ***C***)
[RETURN a GMPList that implements **P** through ***C*** as a linear sequence of Grain Motion Primitives (as in Figure 3.2)]

Figure 3.4 illustrates the details of the Melt-Grow algorithm with a concrete example.

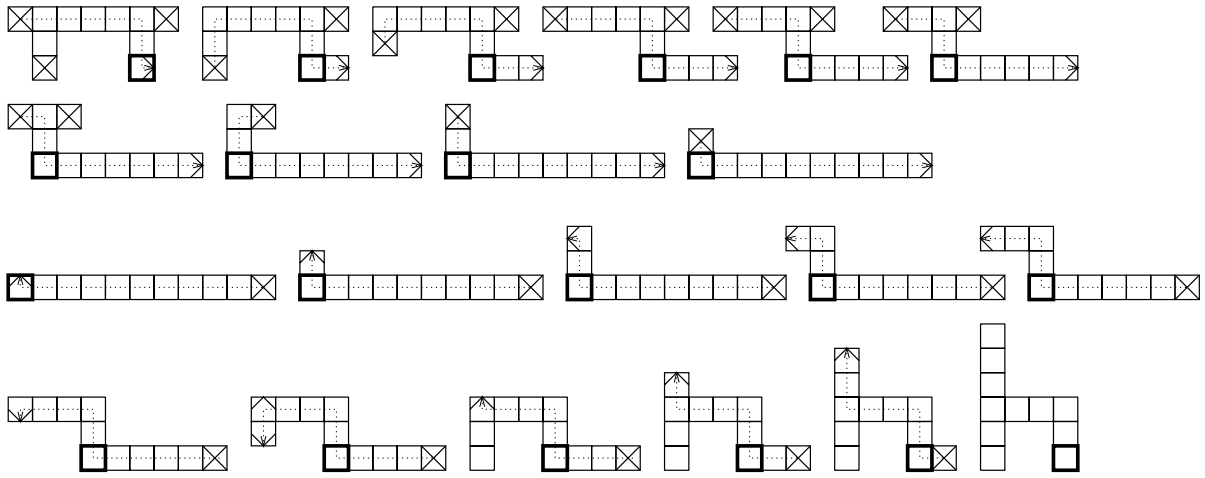


Figure 3.4: Reconfiguring a two-dimensional table shape to a chair shape using the Melt-Grow algorithm. Each square represents one 4x4 Grain (16 Atoms). Grains marked ☒ are mobile, the stem grain is marked ☐, and candidate parent grains are marked◹. This figure represents schematically the output of a simulation where the table and chair are composed of 176 Atoms each.

The first step in the Melt-Grow algorithm 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 **Melt**ing, any mobile Grain still in ***S*** is a suitable mover, and while **Grow**ing, 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 **Melt**ing, any such grain in ***I*** is a suitable parent, and while **Grow**ing, 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 3.2

Completeness

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:

Lemma: 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 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 multiple paths to the new vertex from all other vertices.

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 the prior Lemma, 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, and any remaining Grain in ***S*** will be another), 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 the mobile Grain to the parent Grain can be feasibly executed*: As long as all surrounding Grains are fully expanded (as described above), 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.

Running Time

If the size of Crystals ***S*** and ***G*** is n Grains, then Melt-Grow can be implemented to run as fast as $O(n^2)$. We can demonstrate this 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 running time for **GMPDecompose** is $O(|\mathbf{P}|) = O(n)$, since $|\mathbf{P}| \leq n$. Similarly, the running time for **GMPExecute** is $O(|\mathbf{L}|) = O(n)$.

Transport performs three operations serially. We have just shown that the latter two of these, **GMPDecompose** and **GMPExecute**, are each $O(\mathbf{n})$. The first operation in **Transport** is to find a path from the mobile Grain to the parent Grain. By our observation above, we can find such a path through a DFS in the GCG in $O(n)$ (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's simply a linear search through the Crystal-in-progress. **FindMobile** is also $O(n)$ because the articulation points of any finite

connected graph $\boldsymbol{G}=(\boldsymbol{V}, \boldsymbol{E})$ can be found in $O(\boldsymbol{E})$ time [CLR p.496], and in the GCG $\boldsymbol{E}$ is $O(\boldsymbol{V})$.

DesignIntermediate can construct the skeleton of the intermediate Crystal in $O(n)$. **FindStem** is just a linear search through $\boldsymbol{S}$, 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 $O(n^2)$ because it is a sequence of several operations, the most expensive of which are Melt and Grow at $O(n^2)$ each.

Optimality

An optimal planner would find the shortest reconfiguration plan for any specified $\boldsymbol{S}$ and $\boldsymbol{G}$. This can be difficult, so 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 our 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 $O(n^3)$.

3.3 Implementation

A two-dimensional implementation of the Melt-Grow planner was developed and interfaced to the xtalsim Crystalline Atomic robot simulator described in Chapter 2. The table-to-chair reconfiguration in Figure 3.4 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. The input file for the table-to-chair reconfiguration is included in Appendix 4.

4 Physical Implementation

In this Chapter we consider the design, construction, and evaluation of a two-dimensional Atom. The first several sections document the design process, including specifications, parameters, and implementation alternatives. The next sections describe the details of the implementation of each of the Atom's subsystems. The final section discusses the construction of the Atom and presents measurements and experimental results.

4.1 Design Specifications

Design specifications are guidelines that are fundamental to the design. They formalize the important basic properties of the system. The design specifications were chosen according to the analyses included in this section. For the remainder of the design process they are fixed. We consider the following three major design specifications: number of dimensions, actuator sophistication, and contraction ratio.

4.1.1 Number of Dimensions

Crystalline Atomic robots can be constructed both in two and three dimensions. In two dimensions, Atoms are square; in three, they are cubic. Due to the high degree of regularity in Crystalline Atomic systems, many algorithms developed in two dimensions are readily extendible to three. Similarly, algorithms developed in three dimensions can often be specialized to two.

Development of two-dimensional hardware, however, is more straightforward than development of three-dimensional hardware. One reason is gravity: Atoms in a two-dimensional system can always be supported by a planar environment and never need to exert the strong lifting forces that three-dimensional Atoms must produce. Another reason is

simply the increased size (and possibly complexity) of the expansion and connection mechanisms that a three-dimensional Atom would require.

This project is concerned exclusively with the development of two-dimensional hardware. Where possible, physical designs are used that are readily extendible to three dimensions, but only if such designs are no less optimal than other two-dimensional designs.

Even though the hardware is two-dimensional, some reconfiguration algorithms and the software simulator are developed in three dimensions.

4.1.2 Actuator Sophistication

The actuators that effect Atom expansion and contraction might be designed with fine-grained position and/or force control along the entire range of motion. This could be useful to prevent binding and to maintain efficiency in situations where several connected Atoms need to expand and contract in coordinated unison, for example in a push-pull configuration, as illustrated in Figure 4.1.

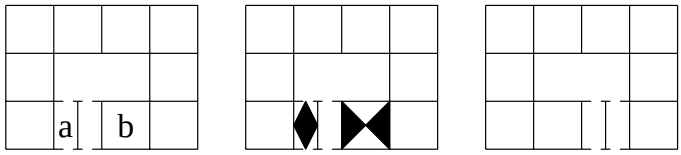


Figure 4.1: A motion sequence which requires coordinated movement. Atom **a** must expand as Atom **b** contracts. Solid lines indicated bonded interfaces, dotted lines indicate free interfaces, diamonds indicate expansion, and bowties indicate contraction.

In the interest of simplicity, fine-grained control is not implemented. Simple binary actuators are used. Situations that suggest coordinated movement are avoided where possible, as depicted in Figure 4.2.

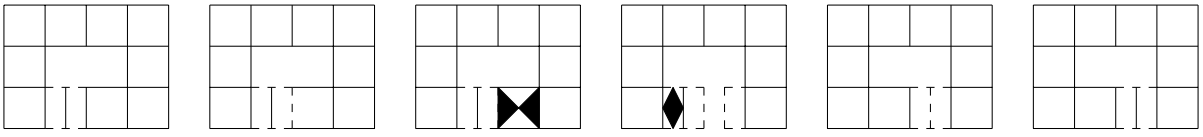


Figure 4.2: A motion sequence which achieves the same reconfiguration as that depicted in Figure 4.1, but without requiring any coordinated movements.

For those cases where coordination is necessary, the actuators are designed to cooperate with minimal binding and acceptable efficiency.

4.1.3 Contraction Ratio

The contraction ratio is the relationship between the side lengths of the fully expanded and the fully contracted Atom. This ratio is important because it determines the number of helper Atoms that must contract in order to achieve the translation of a given Atom. For example, if the ratio is 4:3, four helper Atoms are required, as shown in Figure 4.3.

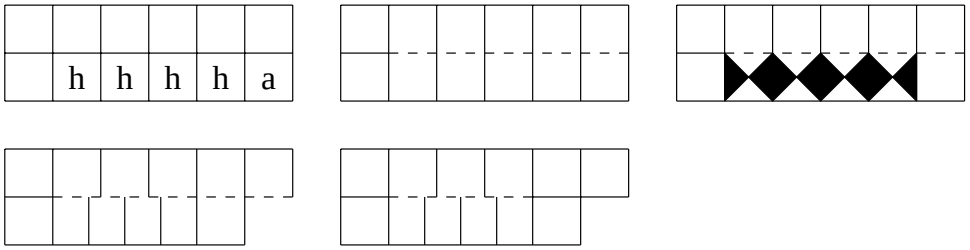


Figure 4.3: A motion sequence which moves Atom **a** one unit to the left, implemented with a contraction ratio of 4:3. The Atoms marked **h** are helper Atoms.

In order to minimize the number of physical Atoms required to effect reconfiguration, the contraction ratio which yields the fewest helper Atoms necessary for translation is most desirable. Obviously, zero helper Atoms is impossible. One helper Atom is also impossible, because it necessitates an infinite (1:0) contraction ratio, as shown in Figure 4.4.

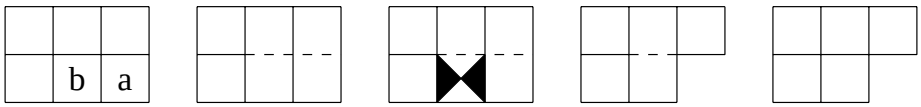


Figure 4.4: A motion sequence which moves Atom **a** one unit to the left, implemented with a contraction ratio of 1:0. Note that Atom **b** disappears.

Two helper Atoms would be required by a 2:1 contraction ratio. This is illustrated in Figure 4.5.

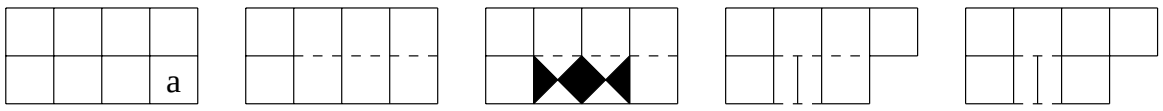


Figure 4.5: A motion sequence which moves Atom **a** one unit to the left, implemented with a contraction ratio of 2:1.

Since two is the fewest number of helper Atoms possible, 2:1 is the most desirable contraction ratio. It is challenging to design a captive linear actuator with such a high contraction ratio, but its benefits are deemed high enough to outweigh this difficulty.

4.2 Design Parameters

Design parameters encapsulate the important details of the design. Unlike the design specifications in the previous section, design parameters are not taken to be immutable. Rather, for each parameter an analysis is undertaken to identify a target value or range. The important relationships between design parameters are represented as equations. This allows the design process to be viewed as an optimization problem: we select the implementation concepts which are predicted to yield design parameters closest to the target values. We consider six categories of design parameters in this section: degrees of freedom, expansion issues, connector issues, rigidity and fault-tolerance, power consumption, and logistics.

4.2.1 Degrees of Freedom

A two-dimensional Atom might be designed with up to eight independent mechanical degrees of freedom. One prismatic DOF per face would effect expansion/contraction, and one connective DOF per face would effect attachment to neighboring Atoms. Such a configuration is shown in Figure 4.6.

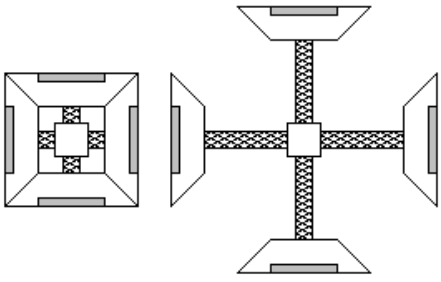


Figure 4.6: An 8 DOF Atom shown fully contracted (on the left) and fully expanded (on the right). Expansion actuators are symbolized by cross-hatching, and connection actuators are drawn as gray rectangles.

Such a design would allow a high degree of controllability. Sixteen individual poses are possible, the two above, and the following four and their isomorphs. This is stated graphically in Figure 4.7.

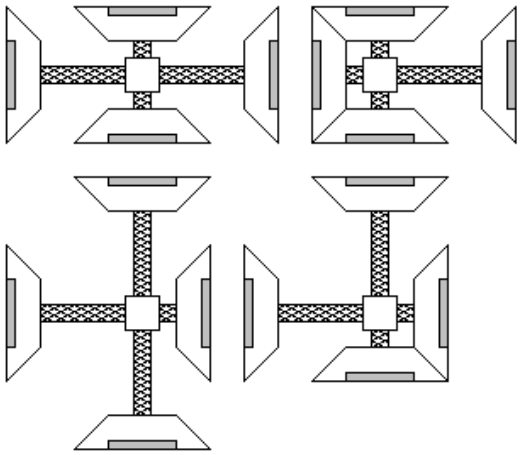


Figure 4.7: In addition to the two poses shown in Figure 4.6, these four and their isomorphs are possible with an 8 DOF Atom.

At the other end of the spectrum of mechanical complexity is a two-dimensional Atom with only three independent mechanical degrees of freedom. One central two-axis prismatic DOF effects expansion/contraction simultaneously on all faces, and two connective DOF on adjacent faces effect attachment, as shown in Figure 4.8.

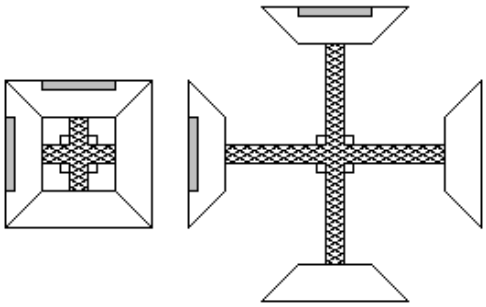


Figure 4.8: A three DOF Atom shown contracted and expanded.

A three DOF Atom is less versatile than the eight DOF version, but two-dimensional systems of such units can still self-reconfigure in a general way. First, since Atoms never rotate relative to each other, the reduction from four connective DOF to two actually imposes

no mechanical restrictions on versatility. Every inter-Atomic interface still necessarily contains one active connection mechanism, so all interfaces are still fully controllable, as we can see in Figure 4.9.

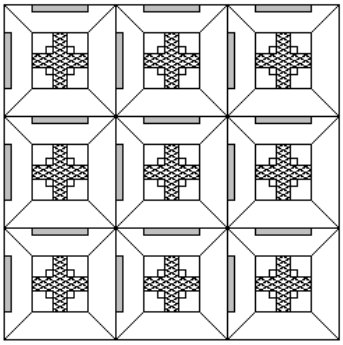


Figure 4.9: A tiling of nine compressed 3 DOF Atoms. Note that every inter-Atomic interface contains exactly one active connection mechanism.

The reduction of prismatic DOF that effect expansion/contraction does impose some restrictions on the types of coordinated movements that are feasible in a neighborhood of Atoms. However, these restrictions are not so severe that general self-reconfiguration in two dimensions is precluded. The proof is by demonstration: most of the two-dimensional reconfiguration algorithms presented herein, including the Melt-Grow planner for shape metamorphosis, are developed so that they can be implemented by three DOF Atoms.

4.2.2 Overall Size/Weight and Speed/Strength of Expansion

The strength required of the actuator which effects Atom expansion and contraction is determined by the internal and external forces it must overcome. There is one major internal force: F_{ways}, the force required to overcome the (static) friction in the mechanical ways which guide the motion of the Atom faces. There are two external forces. One is $F_{inertia}$, the force required to overcome the inertia of the Atoms that move as a result of the contraction or expansion. The other is $F_{friction}$, the force required to overcome the (static) friction between the Atoms to be moved and the planar environment. A safety factor

$$f_{safety} = 2$$

is employed, so the maximum force which the expansion/contraction actuator must produce is

$$F_{expand} = f_{safety}(F_{ways} + F_{inertia} + F_{friction}).$$

The force required to overcome the friction in the sliding ways, F_{ways}, is designed to be negligible.

The force required to overcome inertia, $F_{inertia}$, is calculated from the Atom mass, m_{atom}, the maximum number of Atoms moved by any single expansion or contraction, n_{movers}, and *acceleration*, the necessary acceleration:

$$F_{inertia} = n_{movers} \cdot m_{atom} \cdot acceleration$$

As discussed below, *acceleration* and m_{atom} are kept small to keep $F_{inertia}$ negligible.

By keeping both F_{ways} and $F_{inertia}$ negligible, $F_{actuator}$ is effectively determined by the only remaining parameter, $F_{friction}$. $F_{friction}$ is calculated from m_{atom}, n_{movers}, and u_{static}, the coefficient of static friction between the material which makes up the underside of the Atom and the environment material:

$$F_{friction} = u_{static} \cdot n_{movers} \cdot m_{atom} \cdot g.$$

The coefficient of friction between the Atoms and the environment surface, u_{static}, is kept low (less than about 0.4) by using smooth materials.

The Atom mass, m_{atom}, is made as low as possible (about 10 ounces or less) by keeping the overall size of the Atom small and by using low mass components.

The number of Atoms that will be moved by any one Atom undergoing an expansion or contraction, n_{movers}, is determined by the reconfiguration algorithms that are implemented. The algorithms are designed to require only small n_{movers}. In all algorithms generated by the Melt-Grow planner, n_{movers} is always less than two, demonstrating the feasibility of this goal for general reconfigurations.

In order to keep *acceleration* negligible, the average speed s_{expand} of expansion/contraction must be kept low. If *contracted* is the side length of a contracted Atom, the contraction ratio is 2:1, and the time required for one expansion or contraction is T_{expand}, then

$$s_{expand} = contracted/T_{expand}$$

The expansion speed, s_{expand}, is minimized with a small *contracted* and a large T_{expand}. It is mechanically difficult to make contracted very small (less than about 2 inches). Furthermore, in order to keep the overall running time of hardware experiments low, it is undesirable to make T_{expand} very large (more than about 20 seconds). Even at these bounds, s_{expand} = 0.1 inches/second is not large.

4.2.3 Connector Issues: Size, Strength, Speed, Clearance, and Power Consumption

The holding strength of the inter-Atom connector (measured normal to the inter-Atom interface plane), $F_{connector}$, is determined by the forces it must withstand during expansion and contraction movements:

$$F_{connector} = f_{safety}(F_{inertia} + F_{friction})$$

When disconnected, the connector should impose no holding force.

The connector should be designed so that it can be actively commanded to connect or disconnect. The response time for each of these actions, measured from the time of command issue to the time when completion is guaranteed, should be less than $T_{connect}$ = 20 seconds. The connector should consume power only while responding to commands; it should draw no static power while in the connected or disconnected state.

In order to estimate the volume into which the connection mechanism should fit, some simplifying assumptions are made:

1. The Atom is composed of a central square core of maximum side length *core* and four trapezoidal faces of height *face*.
2. At no time do the core and the faces intrude on each other, nor do the faces of any Atom intrude on the faces of any other Atom. Thus

$$contracted \geq core + 2 \cdot face$$

   or

$$core \leq contracted - 2 \cdot face$$

3. For stability, the maximum allowable height of the Atom above the environment plane height is set at $4 \cdot contracted$.

These are summarized in Figure 4.10.

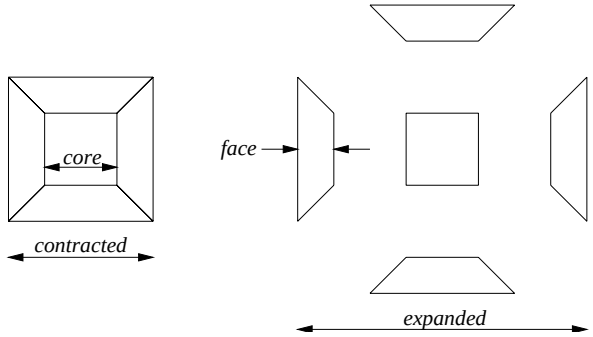


Figure 4.10: A two dimensional Atom composed of a square core and trapezoidal faces.

Thus, the connection mechanism should fit within the trapezoidal face, and it should not be taller than *height*.

Because all inter-Atom interfaces are guaranteed to be composed of one active face and one passive face, protrusions can be allowed on the outer surface of the face trapezoids if suitable intrusions are built into opposing faces. Such protrusions, if present, should be designed so that faces with deactivated active connectors are able to approach faces with passive connectors along any linear trajectory between 0° and 90°, as shown in Figures 4.11 and 4.12.

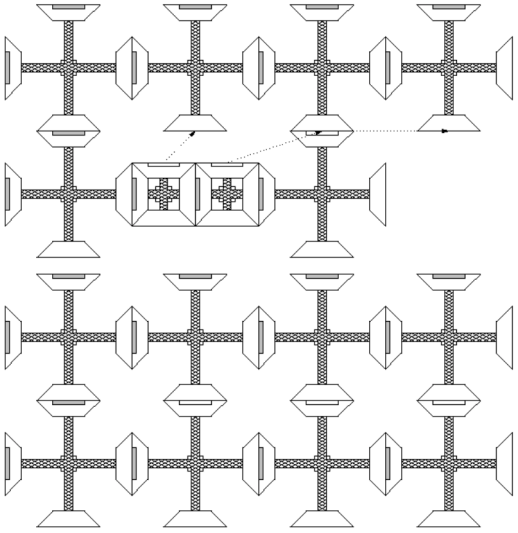


Figure 4.11: A compound movement showing several approach trajectories (dotted lines), including 0°. The upper portion of the figure shows a configuration of 8 Atoms, with two compressed Atoms preparing to expand. The lower part of the figure shows the resulting structure after the expansion of the two Atoms.

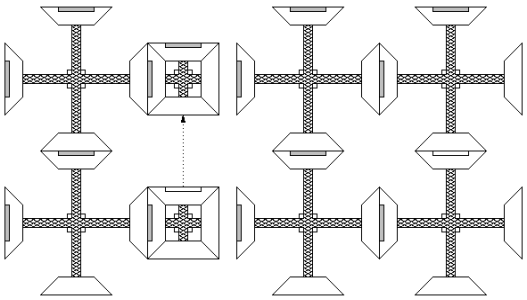


Figure 4.12: A compound movement showing a 90° approach trajectory. The left side of the figure shows a configuration of four Atoms, with two compressed Atoms about to expand. The right side shows the structure that results from the expansion of those Atoms.

4.2.4 Overall Rigidity, Accuracy, Compliance, and Connector Fault-Tolerance

If all Atoms in a Crystalline Atomic system were perfectly rigid (always exactly square) and perfectly accurate (always exactly *contracted* x *contracted* when contracted, *expanded* x *expanded* when expanded), then the connectors at inter-Atomic interfaces would always be positioned at exactly the right locations. Since it is unlikely that either of these suppositions will be true, the connectors are designed to be fault-tolerant to some extent, and the Atoms are designed to be somewhat compliant. When an Atom becomes slightly misshapen (due to a lack of rigidity, accuracy, or both), its connectors should still be able to

activate and connect with neighboring Atoms. By doing so, the built-in compliance should allow the connectors to pull the Atom back into shape.

The rigidity of the Atom is quantified by the maximum angle *rigidity* that a face normal in an expanded Atom can deviate from the core normal, as shown in Figure 4.13.

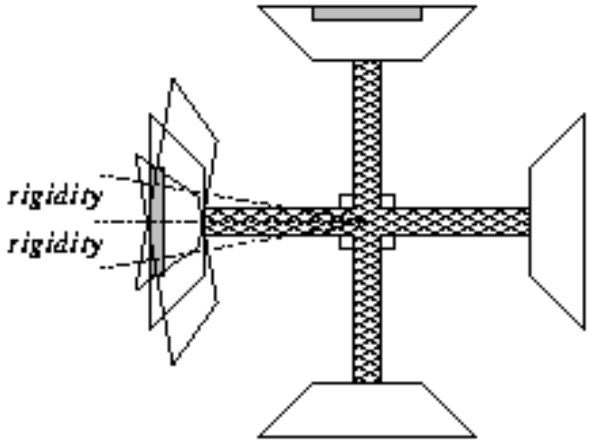


Figure 4.13: Atom rigidity is quantified by the angle *rigidity*.

rigidity should not be more than about 5°.

The accuracy of the Atom is quantified by the maximum distance *accuracy* that a face position (relative to the Atom center) in an expanded Atom can err from *expanded*/2, as depicted in Figure 4.14.

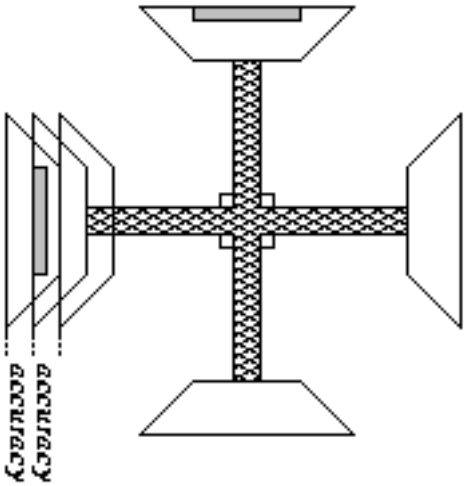


Figure 4.14: Atom accuracy is quantified by the distance *accuracy*.

accuracy should not be more than about 1/32 inch.

The *compliance* of the Atom in each dimension should be at least as great as 2 · *accuracy*, in case the maximum complimentary deviations occur simultaneously on opposing faces:

$$compliance \geq 2 \cdot accuracy$$

The fault-tolerance of the connector should be able to handle all combinations of angular and linear deviation.

4.2.5 Running Time, Power Consumption, and On-board Power Storage

The Atom is powered by on-board batteries. These batteries should have sufficient capacity, $C_{battery}$ (mA · h), to support T_{run} = 4 hours total running time. If the time-averaged current draw of the Atom is I_{atom} = 140 mA, then

$$C_{battery} = 600 \text{ mAh} \geq T_{run} \cdot I_{atom}.$$

4.2.6 Logistical Issues

This project is conducted with limited resources. We would like the Atom to cost no more than *cost* = $300. The average construction time per Atom, after all off the shelf and automatically manufactured parts are in hand, should be no more than T_{build} = 10 hours. Rapid prototyping technology is utilized to speed construction.

A restrictive logistical parameter is T_{design}, the time available to develop a working Atom. Only nine weeks are available.

4.2.7 Summary of Design Parameters

The major design parameters are summarized below in tabular form. They are separated into two categories: primary and derived. Primary parameters are asserted with a

numeric value and are independent. Derived parameters depend on the primary parameters and on each other.

Table 4.1: Primary Parameters

Parameter	Units	Target Value	Description
f_{safety}	(none)	2	factor of safety
F_{ways}	ounces	~0	force required to overcome static friction in face motion ways
m_{atom}	ounces	< 10	Atom mass
n_{movers}	(none)	2	maximum number of Atoms moved by any single Atom expansion or contraction
acceleration	inches/second^2	~0	acceleration of Atoms to be moved by a single Atom expansion or contraction
u_{static}	(none)	< 0.4	coefficient of static friction between the material which makes up the underside of the Atom and the environment material
contracted	inches	2	side length of contracted Atom
T_{expand}	seconds	~20	time required for expansion or contraction
$T_{connect}$	seconds	< 20	time required for connector activation/deactivation
face	inches	0.5	height of Atom face trapezoid
rigidity	degrees	5	Atom rigidity (max angle of face normal deviation)
accuracy	inches	1/32	Atom accuracy (max face position deviation)
cost	dollars	300	maximum construction cost per Atom
T_{build}	hours	10	maximum construction time per Atom
T_{run}	hours	4	minimum total running time
I_{atom}	mA	150	time-averaged Atom current draw
T_{design}	weeks	9	time available for Atom design

Table 4.2: Derived Parameters

Parameter	Units	Target Value	Expression	Description
$F_{inertia}$	ounces	~0	$n_{movers} \cdot m_{atom} \cdot$ *acceleration*	force required to overcome the inertia of the Atoms that move as a result of the contraction or expansion
$F_{friction}$	ounces	< 8	$u_{static} \cdot n_{movers} \cdot m_{atom} \cdot g$	force required to overcome the static friction between the Atoms to be moved and the planar environment
F_{expand}	ounces	16	$f_{safety}(F_{ways} + F_{inertia} + F_{friction})$	maximum force which the expansion/contraction actuator must produce
s_{expand}	inches/ second	~0.1	*contracted*/T_{expand}	average speed of expansion/contraction
$F_{connector}$	ounces	16	$f_{safety}(F_{inertia} + F_{friction})$	minimum connector holding strength
core	inches	1	*contracted* - 2 · *face*	maximum side length of Atom core
expanded	inches	4	2 · *contracted*	side length of expanded Atom
height	inches	8	4 · *contracted*	maximum Atom height
compliance	inches	1/16	2 · *accuracy*	minimum Atom compliance (per dimension)
$C_{battery}$	mA · h	600	$T_{run} \cdot I_{atom}$	minimum battery capacity

4.3 Design Alternatives

In this section we describe the several best results from brainstorming different ways to implement the details of the design. They are each developed far enough to allow a comparison among them to be performed in light of the design specifications and parameters.

4.3.1 Expansion Mechanism

Four alternative concepts for implementing the Atom expansion mechanism are presented. Except for *cost* (construction cost per Atom), T_{design} (design time) and T_{build} (construction time per Atom), it is likely that each of these concepts could be realized so that all design specifications are met and all design parameters are acceptably close to their target values. Thus, *cost*, T_{design} and T_{build} are the critical deciding factors for determining which concept is implemented.

Each concept is developed enough to indicate the major components, allowing *cost* to be compared. Similarly, the number and complexity of components can be predicted, and the number and scope of possible design difficulties can be estimated. These allow T_{design} and T_{build} to be compared.

The four concepts are all based on the simplified Atom geometry described in section 4.2.3. Passive sliding ways are assumed, which allow the trapezoidal faces to move rigidly in and out. The concepts are: "*Springs & Reel*," "*Lead Screws*," "*Linkage*," and "*Rack and Pinion*."

"Springs & Reel"

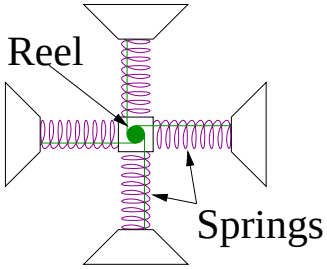


Figure 4.15: The "*Springs & Reel*" expansion mechanism implementation concept.

In this concept, compression springs are mounted between each face and the core so that a continuous outward force is maintained on the faces. In the center of the core a reversible gearmotor is mounted with its drive axis vertical. The motor turns a reel to which four concentric inelastic tapes are attached. The tapes are guided by rollers to exit the core behind the center of each face. The ends of the tapes are fixed to the inner surfaces of the faces. By spinning the reel in one direction, the tapes are let out, and the Atom expands under the force of the springs. By spinning the reel in the other direction, the tapes are pulled in, and the Atom contracts.

"Lead Screws"

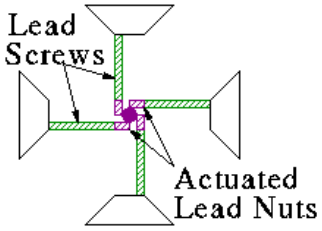


Figure 4.16: The "*Lead Screws*" expansion mechanism implementation concept.

Here, lead screws are rigidly mounted to the rear of each face. The screws are mounted off-center, so that they do not collide with each other in the core. As each screw enters the core it passes through a motorized lead nut. Spinning the nuts in one direction causes the leadscrews to move outward, and the Atom expands. Spinning the nuts in the other direction causes the Atom to contract.

"Linkage"

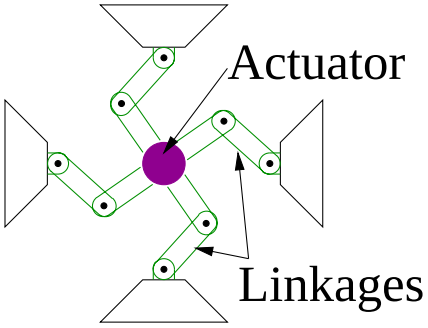


Figure 4.17: The "*Linkage*" expansion mechanism implementation concept.

In this design, a two-bar linkage in the plane of the Atom is attached to the rear of each face. The linkages behind adjacent faces are staggered vertically so that they do not collide with each other. The bars in each linkage are pivoted freely against each other and against the rear of the face, but are rigidly attached to the shaft of a vertically mounted reversible gearmotor at the center of the core. By spinning the motor in one direction, the linkages are forced to collapse, and the Atom is contracted. By spinning the motor in the other direction, the linkages are forced to extend, and the Atom is expanded.

"Rack and Pinion"

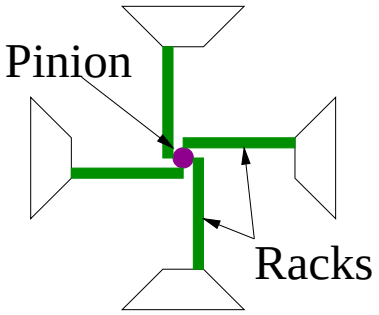


Figure 4.18: The "*Rack and Pinion*" expansion mechanism concept.

Gear racks are rigidly mounted to the rear of each face. So that they miss each other at the center of the core, racks from opposing faces are mounted off-center, and racks from adjacent faces are staggered vertically. The racks are spaced so that a vertically mounted pinion at the center of the core mates simultaneously with all racks. The pinion is driven by a

gearmotor, which spins in one direction to extend the racks, expanding the Atom, and in the other direction to retract the racks, contracting the Atom.

Comparison

Except for "*Lead Screws*", each concept seems to require a at least one gearmotor, which would probably be the most expensive component. "*Lead Screws*" seems to require either multiple gearmotors or some type of gear train, either of which elevate its cost relative to the other concepts.

T_{design} would probably be high for "*Springs & Reel*" because the relative strengths of the springs and the gearmotor need to be carefully matched in order to achieve the required values of T_{expand} and F_{expand}. This could involve custom design or very careful selection of the springs and motor. T_{design} would also be relatively high for "*Linkage*" because the geometry of the bars and pivots of the linkage would need to be carefully calculated and constructed in order to ensure T_{expand}, F_{expand}, *accuracy*, and *compliance* are all met.

T_{build} would probably be relatively high for "*Lead Screws*" and "*Linkage*" due to their relatively higher moving parts count.

This leaves "*Rack and Pinion*" as the concept with the fewest apparent complications. It requires only one moving part per face (the rack) plus one moving part for the core (the pinion). The required motor strength for a given F_{expand} and speed for a given T_{expand} are related simply to the diameter of the pinion, and there are no antagonistic forces to consider as in "*Springs & Reel*". *accuracy* can be controlled by carefully positioning limit sensors.

4.3.2 Connection Mechanism

We considered three alternative concepts for implementing the inter-Atom connection mechanism. As with the expansion mechanism, all three concepts could potentially be realized, so the deciding factors are cost, T_{design}, and T_{build}.

All three concepts assume the asymmetric active/passive face scheme described in section 4.2.1. The concepts are: "*Breakable Permanent Magnet*," "*Clasp*," and "*Channel and Key*."

"Breakable Permanent Magnet"

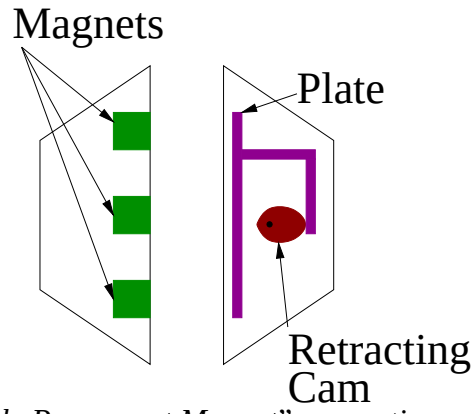


Figure 4.19: The "*Breakable Permanent Magnet*" connection mechanism concept.

In this design, strong permanent magnets are mounted flush with the outer surface of the passive face. The active face contains a ferrous plate which can either be positioned at the outer face surface or slightly retracted. The plate position is controlled by a cam attached to a gearmotor. When the plate is in the outer position, the connector is bonded due to the force of attraction between the magnets in the passive face and the closely positioned ferrous plate in the active face. When the plate is in the retracted position, the force of magnetic attraction is greatly decreased due to the increased distance between the plate and the magnets, and the connector is freed.

"Clasp"

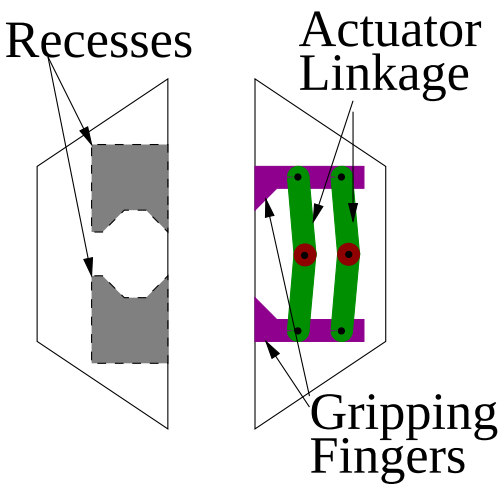


Figure 4.20: The "*Clasp*" connection mechanism concept.

Here the passive face contains two deep recesses. Between the recesses a rib is formed. The active face contains a two-finger gripper actuated by a motor and linkage or gear train. The connector is bonded by extending the gripper so that it clasps the rib in the passive face. The connector is freed by retracting the gripper.

"Channel and Key"

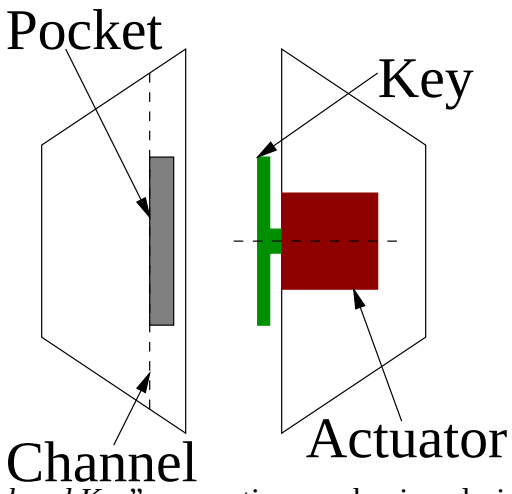


Figure 4.21: The "*Channel and Key*" connection mechanism design concept.

In this concept, the passive face contains a deep horizontal channel 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 gearmotor mounted with its drive axis horizontal and normal to the outer face surface. A bar is attached transversely to the output shaft of the gearmotor.

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 in the passive face, and the connector is bonded.

Comparison

The parts cost would probably be relatively equal for each of these concepts, as they all seem to require a single gearmotor and few other components. One exception could be "*clasp*", which might contain a relatively expensive gear train or linkage.

T_{design} would probably be relatively high for "*Breakable Permanent Magnet*," as the cam mechanism would need to be carefully specified and constructed to give consistent and reliable movement of the ferrous plate. T_{design} for "*clasp*" could also be relatively high due to the linkage or gear train that may be necessary.

T_{build} would likely be high for "*clasp*" due to the larger number of moving parts relative to the other concepts.

"*Channel and Key*" has the fewest apparent complications. It contains only one moving part (the bar). In fact, there are only four components in all: the motor, the bar, the channel, and the motor surround.

4.4 Expansion Mechanism

In this section, we consider the details of the expansion mechanism design. Given the above comparison, this design implements the "*Rack and Pinion*" concept.

All structural blocks in the expansion mechanism are constructed out of ABS plastic by a Fused-Deposition Modeling rapid-prototype system directly from the CAD model. All other parts are stock or modified from stock.

4.4.1 Basic Design

The basic design concept, "*Rack and Pinion*", and dimensions, *contracted* = 2 inches and *expanded* = 4 inches, of the expansion mechanism have already been determined. They are summarized in Figure 4.22.

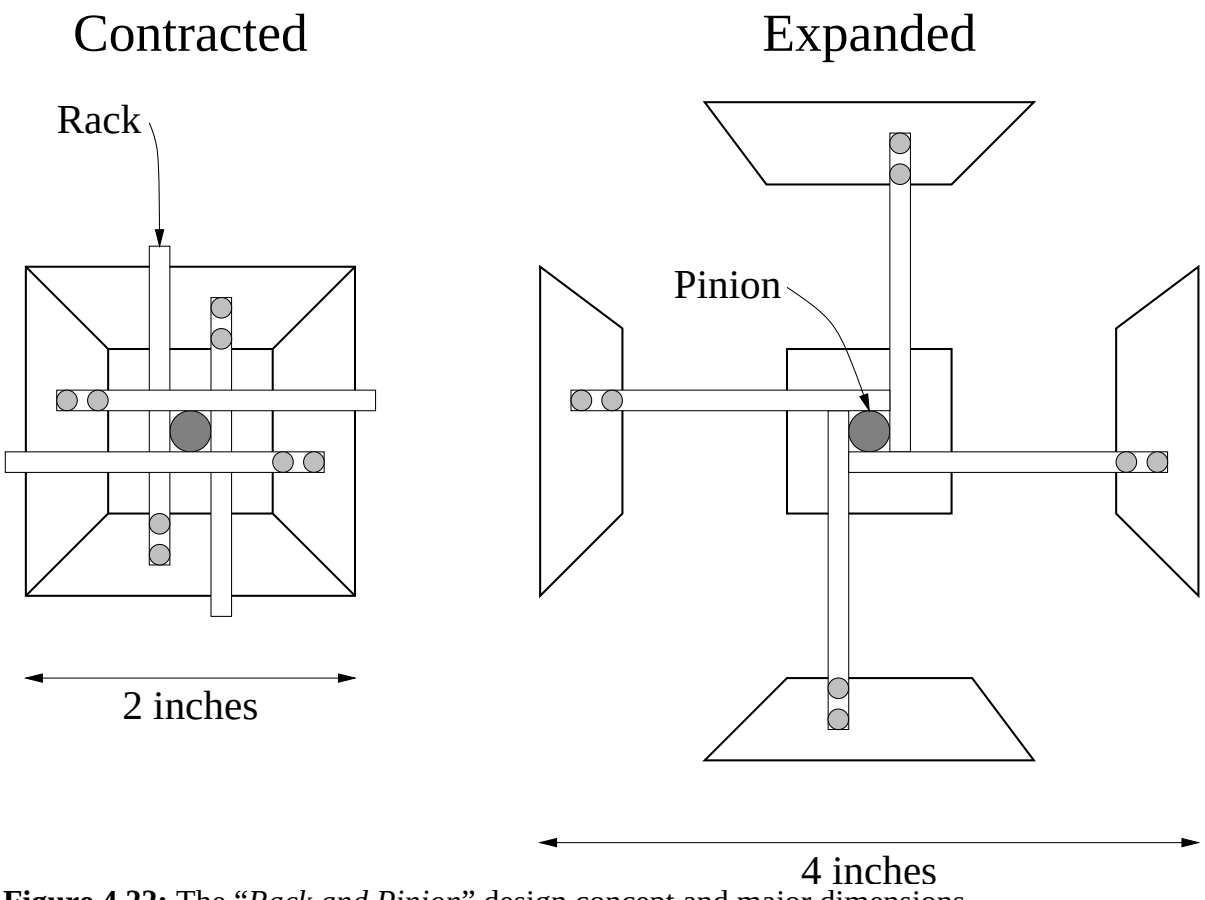


Figure 4.22: The "*Rack and Pinion*" design concept and major dimensions.

The expansion mechanism is constructed as a stack of four vertical stages. The lowest stage contains the actuator which spins the pinions. The second stage contains the racks, pinion, and guide ways for the North-South (NS) faces. The third stage is the same as the second, but it is rotated 90° in order to support the West-East (WE) faces. The fourth stage contains sensors which allow the expansion and contraction motion to be controlled. The height of each stage is kept minimal, but is limited by the packing of the components which must be enclosed. Figure 4.23 gives a schematic representation of the four stages.

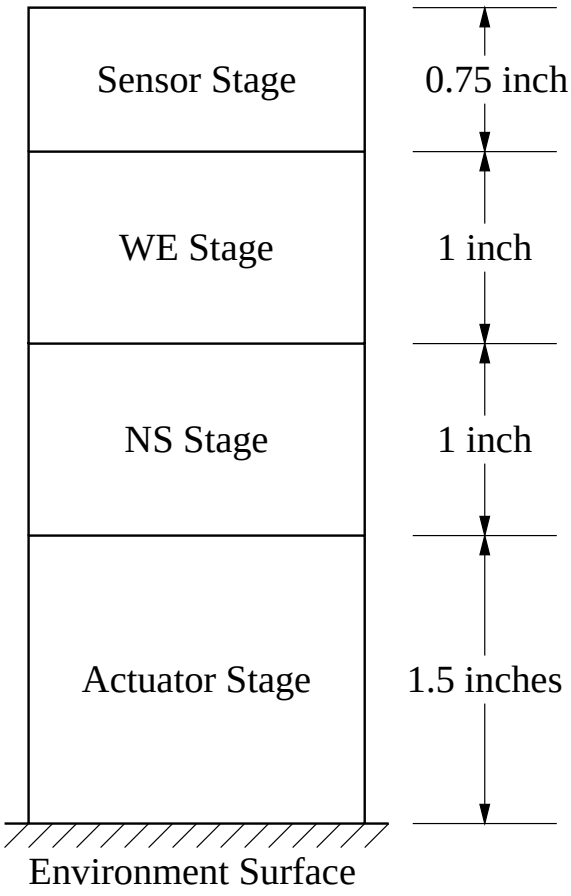


Figure 4.23: The four vertical stages of the expansion mechanism.

Composing the Atom from vertical stages is beneficial in several ways. First, the stages help impose a good scheme for packing the various components in a regular and symmetric fashion. The inter-Atom connection mechanisms for the West-East dimension fit nicely in the West and East faces at the level of the NS actuator stage, and conversely the inter-Atom connection mechanisms for the North-South dimension fit in the North and South faces at the level of the WE actuator stage. Second, the stages facilitate a design which reuses a relatively small number of geometric components in several places, thereby simplifying construction. For example, the face geometry at the actuator stage is identical at all four faces.

A fully detailed CAD model of the mechanics was developed. In the sections that follow, detailed views are presented of each major component. Figures 4.24 and 4.25 present

two views which show the entire mechanism in the fully contracted and fully expanded states.

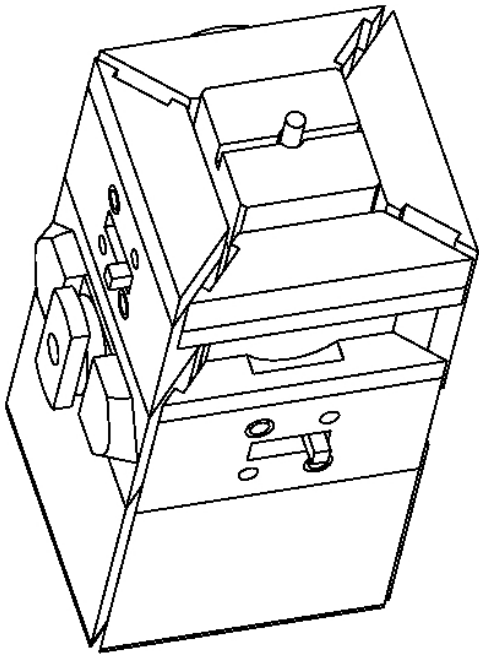


Figure 4.24: The fully contracted Atom. The viewer is looking down at the Atom from an angle. The South face is nearest the viewer.

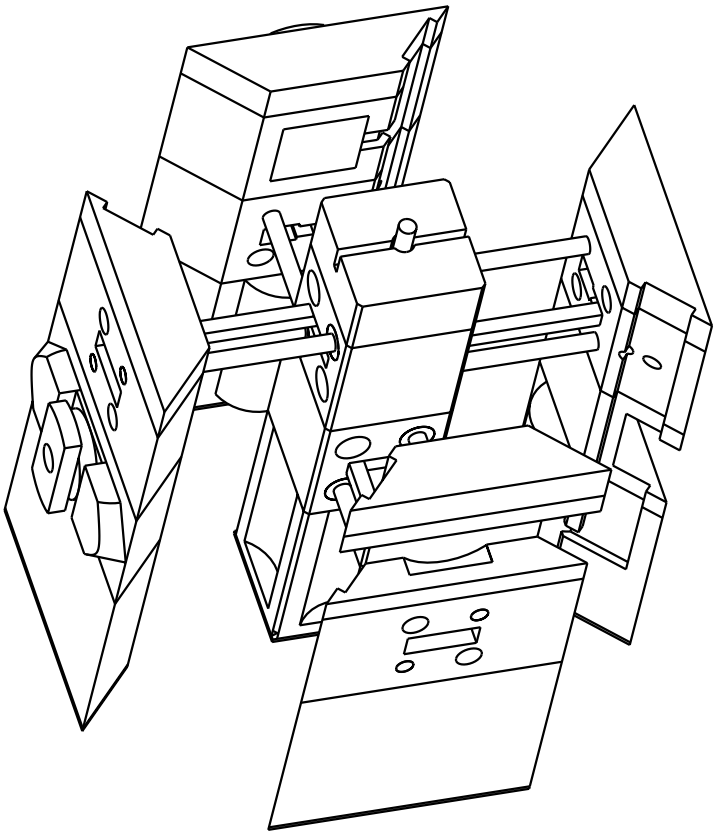


Figure 4.25: The fully expanded Atom.

In these views of the CAD model, separations between the four vertical stages are visible as solid lines. The trapezoidal cross-section of the faces is evident, as is the square cross section of the central core.

The faces and core are designed so that their internal surfaces are all recessed 1/32 inch. This leaves 1/16 inch clearance between the faces and the core, even when the Atom is fully contracted. This clearance allows for adjustments to be made in order to fine-tune the contracted size of the Atom.

Finding a clean way to arrange the various electrical connections that are required to the faces and the core was a challenge. The electronics for the Atom are mostly contained in a circuit board that is mounted vertically on the top of the mechanics, so cabling is required to connect all sensors and actuators in the lower stages. In order to accommodate these cables, a wiring channel is designed into the clockwise-most vertical surface (looking down on the Atom) of each face trapezoid. The two wires which supply power to the expansion motor, which is located in the core at the actuator level, are affixed to two chamfered vertical corners on the core block.

A veneer of Teflon sheet is affixed to the bottom of each face and to the bottom of the core in order to minimize friction with the environment.

4.4.2 Actuator Stage

Because the actuator stage contains both the expansion drive motor and the batteries, it was placed lowest in order to keep the center of mass of the Atom close to the environment surface. Figure 4.26 presents a detailed, exploded view of the actuator stage.

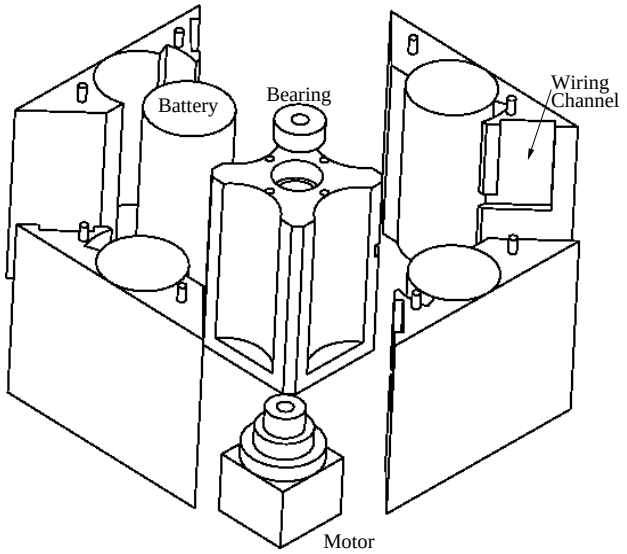


Figure 4.26: The actuator stage, exploded view.

The batteries are 3V 2/3A size Lithium cells. Their electrical selection is discussed below in section 4.6.1. They are placed in semi-cylindrical pockets in the actuator stage of each face.

The motor is a Lego toy "Mini-Motor". This motor was selected for its small size (~5/8 inch cube), relatively high torque (about 2 oz-in peak at 10 rpm, 12 V, 80 mA), and low cost ($11). The motor's torque and current draw were measured experimentally as a function of speed (see Appendix 2). It was critical to ensure that the motor would supply enough power to satisfy the expansion force (16 ounces) and expansion speed (0.1 inch/second) design parameters. Since the pinions that the motor drives have 1/4 inch pitch diameters, the motor could theoretically supply (2 oz-in) / (1/8 in) = 16 ounces of force to the four racks at a speed of (10 rev/min) · (3.14 · 1/4 in/rev) = 7.85 in/min = 0.13 in/sec. Thus, both design parameters are met.

The motor transmits rotary power to the pinions in the NS and WE expansion stages through a single 1/8 inch shaft. The shaft is supported at its extremities by shielded ball bearings. While the shaft itself is not illustrated, the bearing which supports its lower end is visible in Figure 4.26.

4.4.3 Rack and Pinion Stages

The rack and pinion stage contains the pinion gear, racks, and sliding support ways for two opposing faces of the Atom. There are two rack and pinion stages, one for the North/South pair of faces, the other for the West/East pair of faces. The components of the two rack and pinion stages are exactly identical; the NS rack and pinion stage is positioned above the actuator stage, and the WE rack and pinion stage is positioned above and arranged normal to the NS stage. A detailed, exploded view of one rack and pinion stage is presented in Figure 4.27.

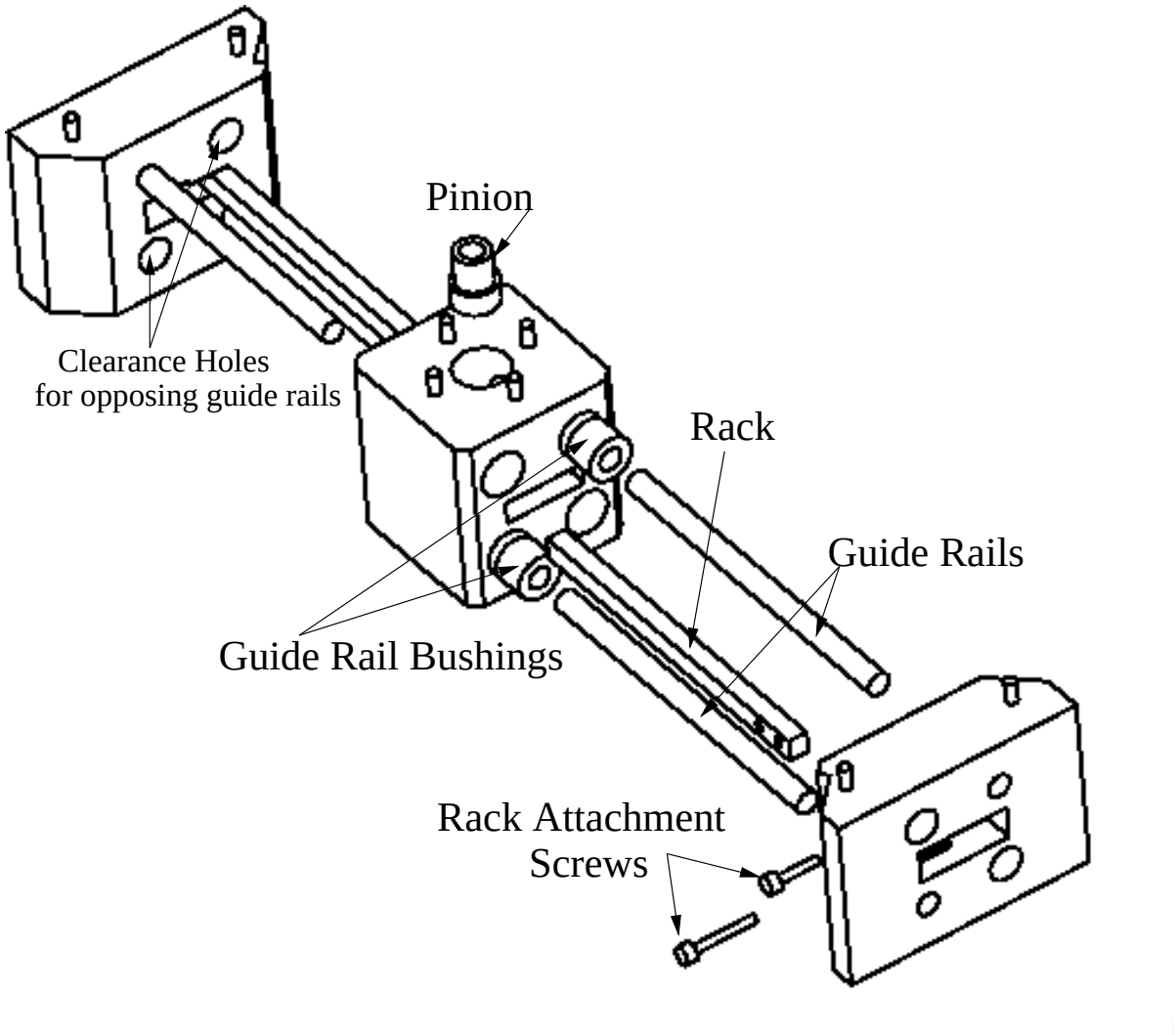


Figure 4.27: The rack and pinion stage, exploded view.

The sliding ways which support each face are composed of two 1/8 inch diameter steel guide rails affixed to the rear of the face. The guide rails extend into Rulon bushings mounted in the core, which allow the rails to slide with minimal friction. Two guide rails are used per face for increased rigidity. In order to support a 2:1 contraction ratio, the faces are designed to include clearance holes which allow the ends of the guide rails from the opposing face to pass through when the Atom is contracted. At full contraction, the end surfaces of the guide rails for each face are coplanar with the outer surface of the opposing face. At full expansion, about 1/2 inch of the guide rails remain embedded in the Rulon bushings.

The 64 diametral pitch racks, which are 1/8" wide at the face and 1/8" deep, were chosen as the smallest commonly available size. They are mated with 16 tooth 1/4 inch pitch diameter pinions, again chosen for their small size. The racks are rigidly mounted to the rear of each face with transverse screws.

The boundary state of interaction between the racks and the pinion that occurs at full expansion was carefully considered. Since the contraction ratio is 2:1, it would seem that at full expansion the end surfaces of the racks would need to coincide exactly with the center plane of the pinion. This would not be a good situation, because it is likely that there would not be enough meshing between the pinion and rack teeth to ensure that the racks are held captive, as shown in Figure 4.28.

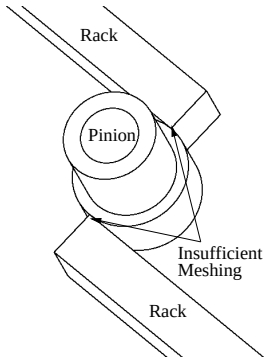


Figure 4.28: Detail of interaction between racks and pinion at full expansion, showing lack of sufficient tooth meshing. Teeth are not modeled directly; only pitch surfaces are shown. The racks are in danger of becoming disengaged from the pinion.

If the racks are made longer, this situation can be avoided. This is the solution employed in the design. The racks are made 1/8 inch longer (about 2.5 teeth) than they would be in the flush situation, as illustrated in Figure 4.29.

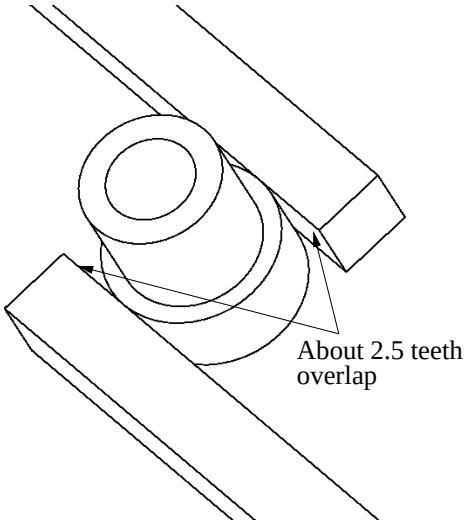


Figure 4.29: Detail of interaction between elongated racks and pinion at full expansion, illustrating sufficient tooth meshing.

But if the racks are made longer, then they will protrude out of the outer face surfaces when the Atom is fully compressed, as shown in Figure 4.30.

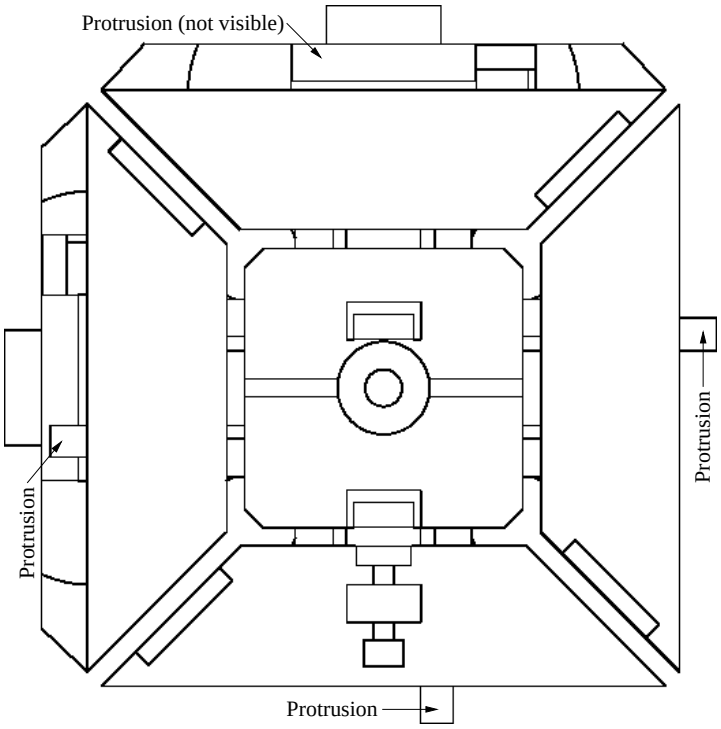


Figure 4.30: Top view of compressed Atom, showing protrusion of racks.

This is a serious consequence, because it implies that adjacent compressed units will not be able to slide relative to each other. Since the racks in both units are at the same height, they will collide during some sliding motions between compressed units. This imposes a limitation on the type of movements possible during reconfiguration. However, this particular imposition is minor. No reconfiguration plans generated by the Melt-Grow planner require adjacent compressed units to slide relative to each other, nor do any of the hand-coded reconfigurations.

4.4.4 Sensor Stage

Two bi-level position sensors are used to control the expansion and contraction movements of the Atom. Hall-effect sensors are used because of their small size, ease of adjustment, and low cost.

One sensor is used to determine when the faces have moved close enough to the core during a contraction movement. The Hall element for this sensor is rigidly affixed to the core block. A permanent magnet that trips the Hall element is mounted to the face via a linear adjustment screw mechanism. This allows the contracted face position to be fine-tuned.

The other sensor indicates when the faces have moved far enough from the core during an expansion movement. A permanent magnet is affixed to the top of the pinion drive shaft so that it spins with the pinions. The Hall element for this sensor is mounted to the core block so that it is triggered by the rotating magnet once per revolution. Since about 2.5 revolutions are required to fully expand the Atom, the third triggering of the Hall element during an expansion is interpreted as the limit indication. Fine-tuning of the extended face position is achieved by adjusting the rotational phase of the spinning magnet. Figure 4.31 illustrates this arrangement.

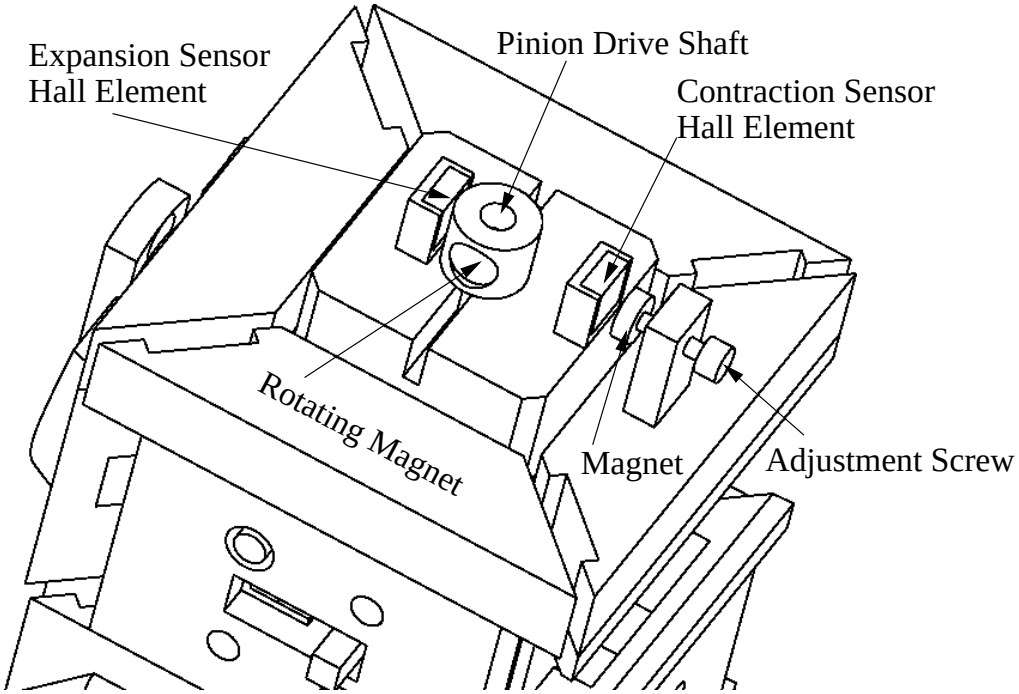


Figure 4.31: Detail of the two expansion/contraction sensor assemblies.

4.5 Connection Mechanism

Now we consider the details of the connection mechanism design, which is based on the "*Channel and Key*" design concept. As with the expansion mechanism, structural blocks are constructed out of ABS plastic by a Fused-Deposition Modeling rapid-prototype system, and all other parts are stock or modified from stock.

4.5.1 Basic Design

The basic parts and operation of the "*Channel and Key*" connection mechanism have already been described. The half of the connector that contains the key is called the active face, and the half containing the channel is called the passive face. Figure 4.32 presents a high-level representation of the components.

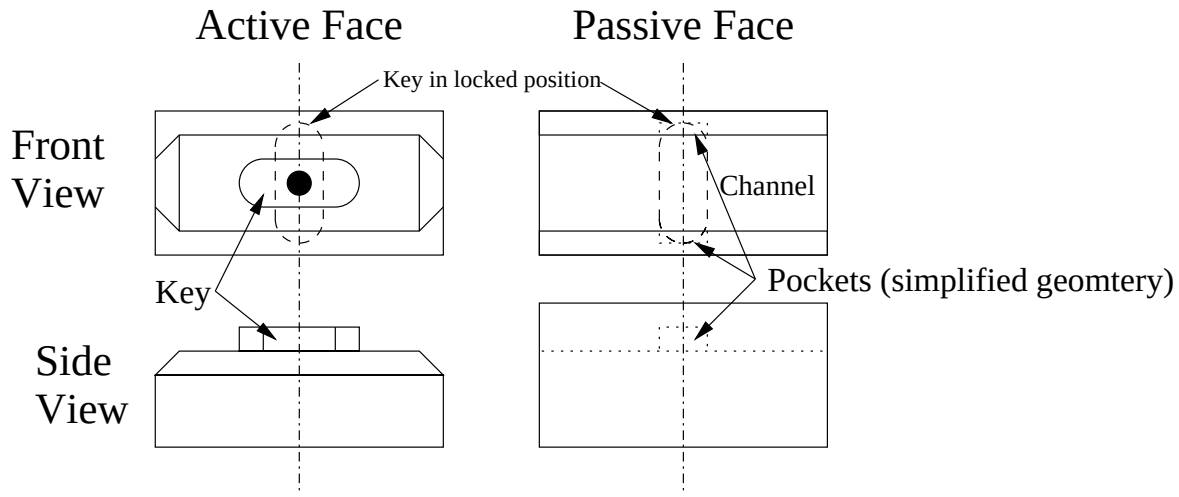


Figure 4.32: The "*Channel and Key*" design concept. When in the locked position, the key on the active face fits into the pockets in the passive face, locking the two faces together. This diagram presents a simplified pocket geometry; in the actual design, the pockets are profiled and the key is in a locked position when it is diagonal rather than vertical.

The channel on the passive face is designed to accommodate a rectangular protrusion on the active face. This way, horizontal sliding is permitted between the two mated faces as long as the key is not in the locked position, but vertical sliding and rotation are prevented. When the two faces are mated and the key is rotated into the locked position, all motion between the faces is prevented. The above diagram shows a simplified pocket geometry; in the actual design, the pockets are profiled and the key is in a locked position when it is diagonal rather than vertical.

4.5.2 Active Face

The active face contains a gearmotor for rotating the key as well as sensors for detecting the position of the key. The same motor is used as that in the expansion mechanism, the Lego Mini-Motor. The Mini-Motor was chosen for its compactness, appropriate mechanical configuration, and relatively slow speed. Hall effect sensors are used to detect the position of the key. Two sensors are employed: one to detect when the key is in the unlocked position, and one to detect when the key is in the locked position. The Hall elements for the sensors are mounted in the body of the active face below the key. Permanent

magnets, which trip the sensors, are mounted in the extremities of the key. When the key is in a horizontal position, one of the magnets is located above the unlocked Hall element. When the key is in a diagonal position, one of the magnets is located above the locked Hall element. Figure 4.33 shows the details of the active face.

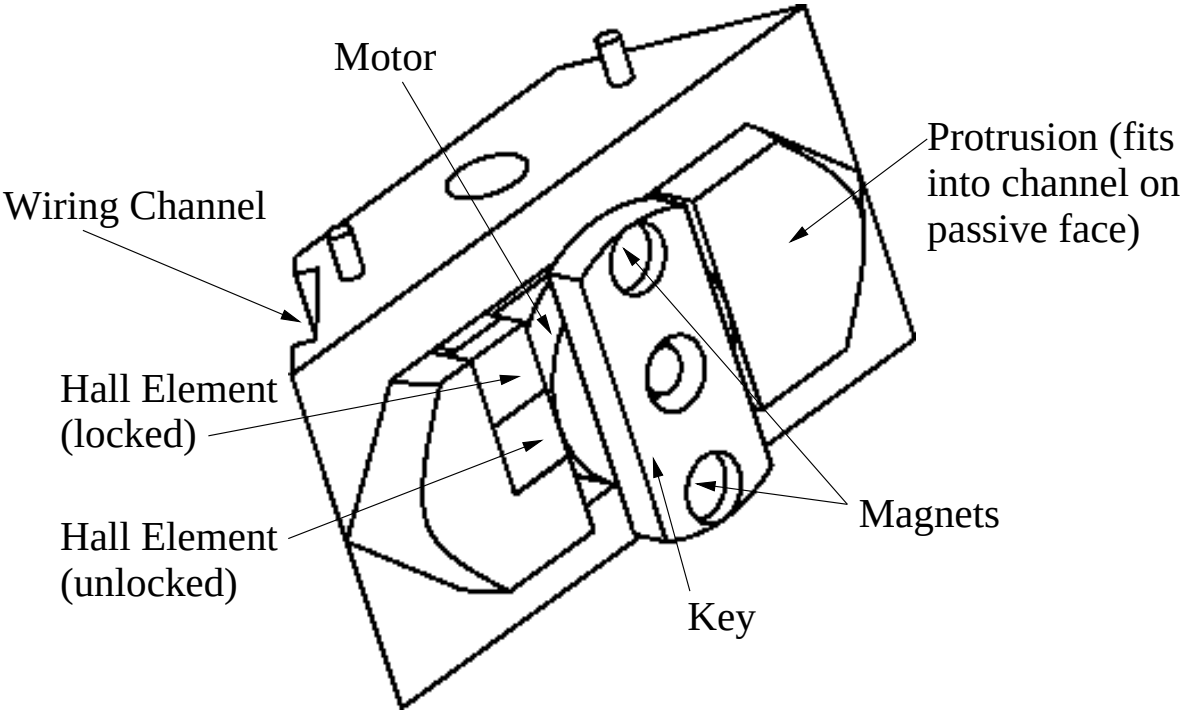


Figure 4.33: The connector active face, containing the key.

The end surfaces of the key are rounded to a radius that is slightly shorter than the circular path through which the key travels. This eases the mechanical interference which occurs when the key is rotated into the pockets in the passive face. Also, the ends of the protrusion are formed into conical chamfers to aid self-alignment when the active and passive faces approach each other.

4.5.3 Passive Face

The passive face contains a channel into which the protrusion on the active face fits as well as the pockets which receive the key, as shown in Figure 4.34.

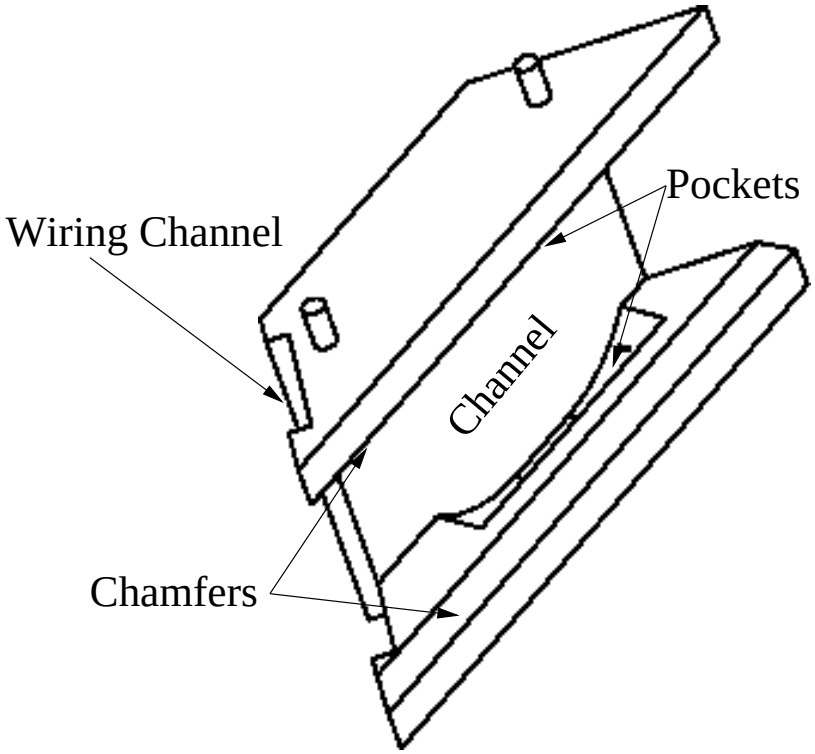


Figure 4.34: The connector passive face, containing the channel and pockets.

Chamfers are included to help guide the protrusion of the active face into the channel.

As discussed in Appendix 2, the Lego Mini-Motor must not be driven to a full mechanical stall. If it is, it becomes impossible to operate the motor at all, even to back it out. This is one factor that complicates the design of the key and pocket geometry. Another factor is that it is desirable for the key and pocket to interact so that the rotary action of the key during locking serves to actively correct the alignment of the two faces. As discussed above, the faces may not start out perfectly aligned due to play in the Atom rigidity and compliance.

To account for these factors, a “scooping” action was designed for the key and pockets. The pockets are elongated, have rounded profiles, and are positioned with complementary offsets from the center of the connector. The key is always actuated in the same direction, clockwise in Figures 4.35 and 4.36.

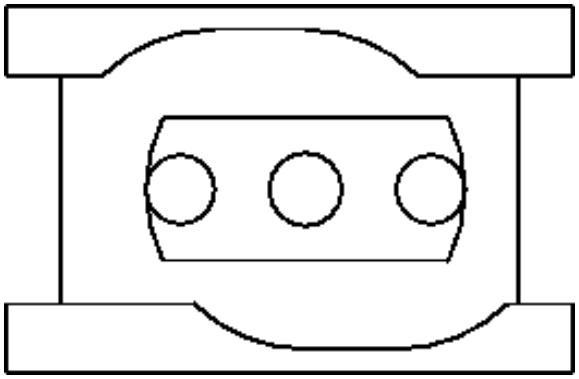


Figure 4.35: Cut-away view of the connector showing the key in the unlocked position.

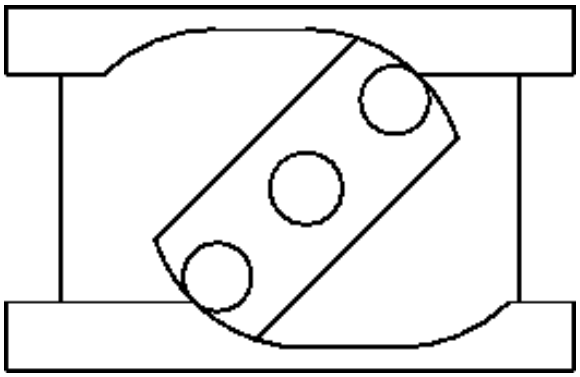


Figure 4.36: Cut-away view of the connector showing the key in the locked position. The connector is always rotated clockwise.

If the key is not perfectly centered horizontally in the channel, then the clockwise rotary scooping action will draw it into alignment during the locking movement. Even though horizontal movement is prevented when the key is in the locked position, the only restriction ever imposed on the rotary movement of the key is friction. Thus, the motor should never be presented with a hard mechanical stall.

To effect locking, the key is assumed to start out in the unlocked position. The motor is activated in the clockwise direction until the locked position sensor is tripped. To effect unlocking, the motor is activated in the clockwise direction until the unlocked position sensor is tripped.

4.6 On-board Electronics and Software

In the past several sections, we have examined the design of the major mechanical subsystems of the Atom. The mechanics are controlled by another major subsystem: the on-board electronics and software. The Atom contains an on-board processor, power supply, and support circuitry which allows both fully untethered and tethered operation. Atoms are connected by a wired serial link to a host computer for purposes of hardware debugging and low-level control. For untethered operation, an experiment-specific operating program (an Atom state sequence) is first downloaded to each Atom over its tether. When the tether is removed, the Atoms rely on their on-board infra-red receivers to detect synchronization beacons from the host. As described below, each Atom transitions to its next stored state upon receipt of a synchronization beacon.

In this section we will discuss the on-board electronics and control software. In the subsequent section, we will present the host electronics and software that is used to communicate with the Atom. Figure 4.37 presents a high-level block diagram of the on-board electronics, and Appendix 2 contains a full schematic diagram.

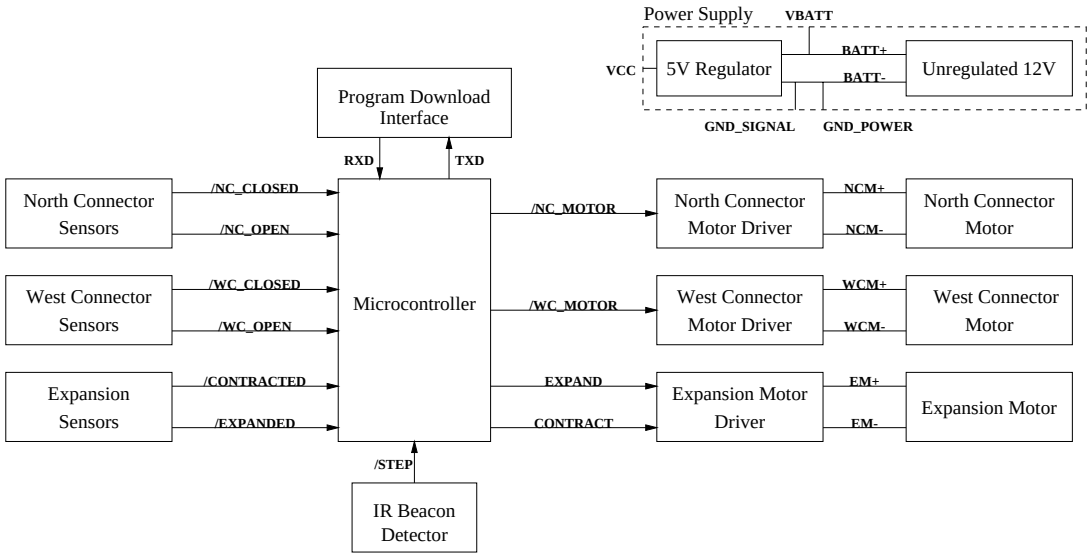


Figure 4.37: Block diagram of the on-board electronics.

The on-board electronics are contained entirely on one 2 by 3 inch printed circuit board, which is mounted vertically to the top of the Atom. Cables embedded in the wiring channels built-in to the mechanics connect the electronics to the Atom's sensors and actuators (see Figures 4.26, 4.33, and 4.34).

At the heart of the on-board circuitry is an Atmel AT89C2051 microcontroller (U5). This chip runs the on-board software which communicates with the host over the program download interface (i.e. the tether) and which monitors and controls the Atom's sensors and actuators. The AT89C2051 (or just the '2051) is an 8-bit microcontroller with an MCS-51 (8051) compatible core. The '2051 was chosen for the following reasons:

- **Ease of Development** - assemblers, monitors, development boards, technical data, and programmers are all readily available for the '2051.
- **On-chip Flash** - the '2051 has 2k of on-chip flash memory for storing programs, in addition to the 128 bytes of RAM which are part of the MCS-51 core. This means that chip count can be reduced, since neither separate program storage ROM nor data storage RAM devices are necessary.
- **I/O Configuration** - the '2051 is a tight fit to the I/O requirements of this project. Only two I/O pins are left unused.
- **Low Cost** - the '2051 is available from a friendly supplier for only $3.49 each in single quantities.

The '2051 is run at 11.059MHz by a crystal oscillator. This frequency is chosen to match the baud rate requirements for the program download interface. A power-on reset circuit is included, as well as a terminal for manual reset.

We will consider the '2051 more when we examine the on-board control software. But first we will look at the three major types of circuit which interface to and support the '2051: the power supply, actuator control, and sensor interface circuits.

4.6.1 Power Storage and Supply

The Atom contains an on-board power supply designed to provide roughly 100mA for 3-4 hours. Since the Atom is expected to require about 50-100mA of current on average (depending on how often the motors are being used), this should yield a reasonable running time before the batteries need to be replaced, thus satisfying the running time design parameter. As described above, four 2/3A size Lithium primary cells provide about 3 volts each at 600mAh. These cells are connected in series to provide about 12 volts, which is used as the motor power supply. The rest of the Atom's circuitry is run at 5 volts, which is provided by a 7805 regulator (U1) in a TO-220 case.

Test points are provided to allow the power supply voltage levels to be monitored, and a quick-release power switch is included to allow the Atom to be switched off in case of disaster.

4.6.2 Actuator Interface Circuits

The Atom contains three actuators: the expansion motor and two connector motors. As discussed above, all three of these are Lego Mini-Motors. An experimental analysis of these motors was performed, the full results of which are presented in Appendix 2. Importantly, this analysis showed that the Mini-Motors would be suitable if they were run at about 12 volts, and that they would draw about 70-100mA while operating. MOSFET-based power driver circuits were designed to meet these requirements. A two-stage approach is employed for each driver. The initial stage for each driver is formed by a 2N2222 NPN common-emitter amplifier, which translates the TTL-level signals from the microcontroller to 0-12V levels suitable for the MOSFETs. The output stages are built from IRF7105 complementary power MOSFET chips, which each contain one P-type and one N-type device.

Two ‘7105s (U6 and U7) are used in an H-bridge configuration to allow bi-directional control of the expansion motor. Importantly, this configuration also serves to electrically brake the motor when it is not energized.

One ‘7105 is used for each connector motor (U2 and U3). Even though the connector motors are only driven in one direction, a design was chosen that uses two power devices per motor to support electrical braking. In the off state, a 100 ohm braking resistor is connected across the motor. This simplifies control, since the motors stop moving very soon after power is removed.

1uF bypass capacitors (not shown in the schematic diagram) are connected across the leads to each motor to reduce noise in the rest of the circuitry. These capacitors are embedded in the Atom mechanics as close as possible to each motor to minimize the effect of the inductance of the motor supply wires.

4.6.3 Sensor Interface Circuits

The Atom contains seven sensors: two for the expansion mechanism, two for each of the connection mechanisms, and one infra-red detector. The six mechanism sensors are bi-level DN6852 Hall-effect devices from Panasonic. These provide active-low logic level outputs which, after 10k pullup resistors, are connected directly to the microcontroller.

The infra-red detector employed is a Sharp GP1U581Y module. This module includes an IR phototransistor as well as signal amplification and conditioning circuitry. The GP1U581Y provides a logic-level signal output that represents any digital control signal that has been modulated onto the 38kHz IR carrier the unit is designed to receive.

4.6.4 Control Software

The on-board control software for the Atom is written entirely in MCS-51 assembly language. The total code size is about 1.5k. There are two major modes of operation: command mode and sequence mode.

Command Mode

The Atom enters command mode upon power-up, after all initialization routines have been completed (these simply set up the microcontroller's on-board peripherals and make sure that the Atom's motors are not active). In command mode, the Atom presents a full user interface to a human operator (who uses a 9600 baud serial terminal program) via the host program download (tether) interface. An interactive prompt is presented to the operator which allows all Atom functionality to be explored. Commands are provided to allow

- interrogation of the Atom's sensors
- activation and deactivation of the Atom's motors
- entering a state sequence to be performed subsequently in untethered mode
- switching to untethered mode

Command mode is used for development, debugging, and alignment of the Atom. Command mode is also used to set up each experiment that is performed in unthethered mode.

Sequence Mode

The Atom enters sequence mode upon receipt of the corresponding command from the operator. The Atom remains in sequence mode until it is powered off or until it receives any data on the program download interface (i.e. until the operator hits a key on the host while the tether is attached to the Atom). The Atom will only allow itself to enter sequence mode when the on-board state sequence buffer is not empty.

The *state sequence buffer* is a region of RAM on the microcontroller which stores a list of Atom states that define each configuration of the Atom during an experiment. Two bits are

used to store each state, and up to 256 states may be stored. Even numbered states define the connection status of the Atom, and odd numbered states define the expansion status. Since the Atom has two binary connectors, four connection states are possible:

0. Disconnected North, Disconnected West
1. Connected North, Disconnected West
2. Connected West, Disconnected North
3. Connected North, Connected West

Since the Atom has only one binary expansion actuator, two expansion states are possible:

0. Expanded
1. Contracted

As described above, the operator enters a state sequence while in command mode. Upon switching to sequence mode, the Atom waits to receive signals from its infra-red detector. Such signals will be sent by the host to indicate when to transition from state to state. This allows all Atoms involved in an experiment to stay temporally synchronized. When an Atom reaches the end of its state sequence during an experiment, the sequence is recycled so that the next state is the first state in the sequence.

4.7 Host Electronics and Software

A small interface board was constructed which allows the Atom to be connected to a serial port on a host computer. The major components of this interface board are depicted in Figure 4.38, and a full schematic diagram is included in Appendix 3.

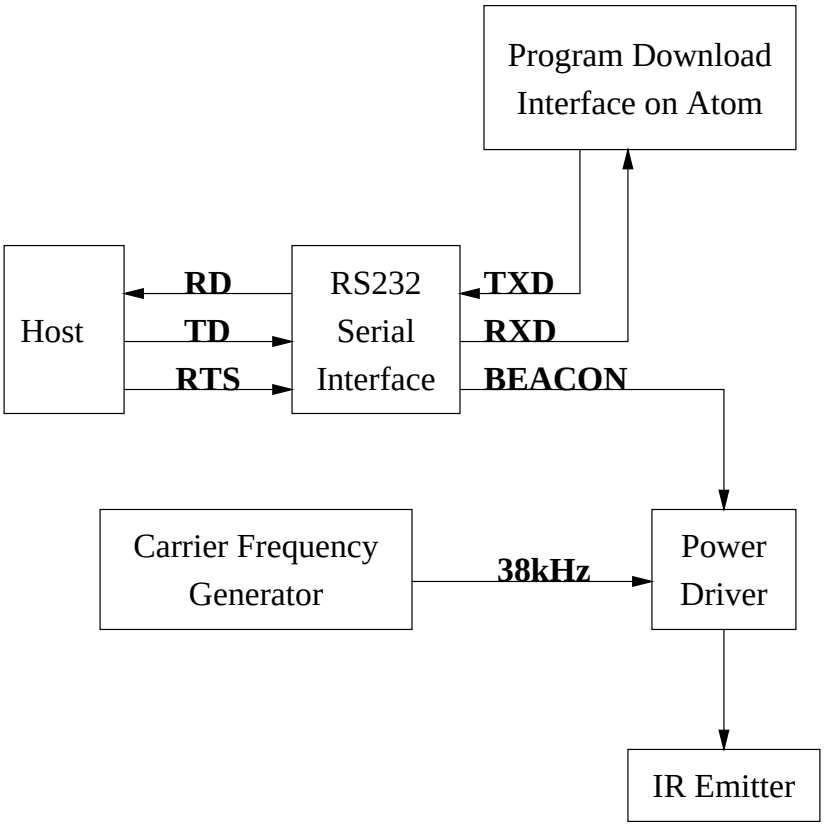


Figure 4.38: Block diagram of the host electronics.

The host machine is an IBM-style PC. The host interface electronics connect to one of the machine's RS-232 serial ports through a 9-pin connector. We will consider the design and function of the interface electronics and the software used on the host in the next two sections.

4.7.1 Level Translation and Synchronization Beacon Electronics

One major function of the host interface electronics is to perform signal level translation. A Maxim MAX232CPE chip (U8) is employed for this purpose. The '232 translates the RS-232 12 volt signals to and from TTL level for communication with the Atom via the program download interface. The '232 also does signal level translation for the RTS signal, which is an RS232 output that would be otherwise unused in this project. We use RTS to control the infra-red synchronization beacon.

Control of the synchronization beacon is the other main function of the host interface electronics. As mentioned above, this beacon must be supplied with a modulated 38kHz squarewave carrier. In our implementation, the carrier waveform is generated by a 555 timer circuit (U9) running in astable mode. We modulate the carrier with the signal from RTS using a saturated 2N2222 switch. Finally, we amplify the modulated signal with another 2N2222 stage before it is sent to an infra-red LED.

4.7.2 Software

Two programs are run on the host to operate the Atom. One is just a serial terminal program. We use the DOS program Kermit, but any such program should work. The other program is a small utility used to control the synchronization beacon through the RS-232 RTS signal. We developed this utility, called simply *atom*, in C++ to run under DOS. *atom* handles the low-level timing associated with generating the appropriate beacon waveform (the beacon uses a pulse code to gain some noise rejection over a simpler level-activated approach). *atom* can be run in automatic mode, where it supplies a beacon at regular intervals (typically 10-20 seconds), or it can be run in manual mode, where the operator specifies when to send each beacon.

4.8 Experiments and Evaluation

We have now completed our narrative of the design of the Atom. We turn to discussing the construction of the Atom and the measurements and experiments that were performed to evaluate it. Two Atoms were constructed. Figure 4.39 presents an image of the completed Atom.

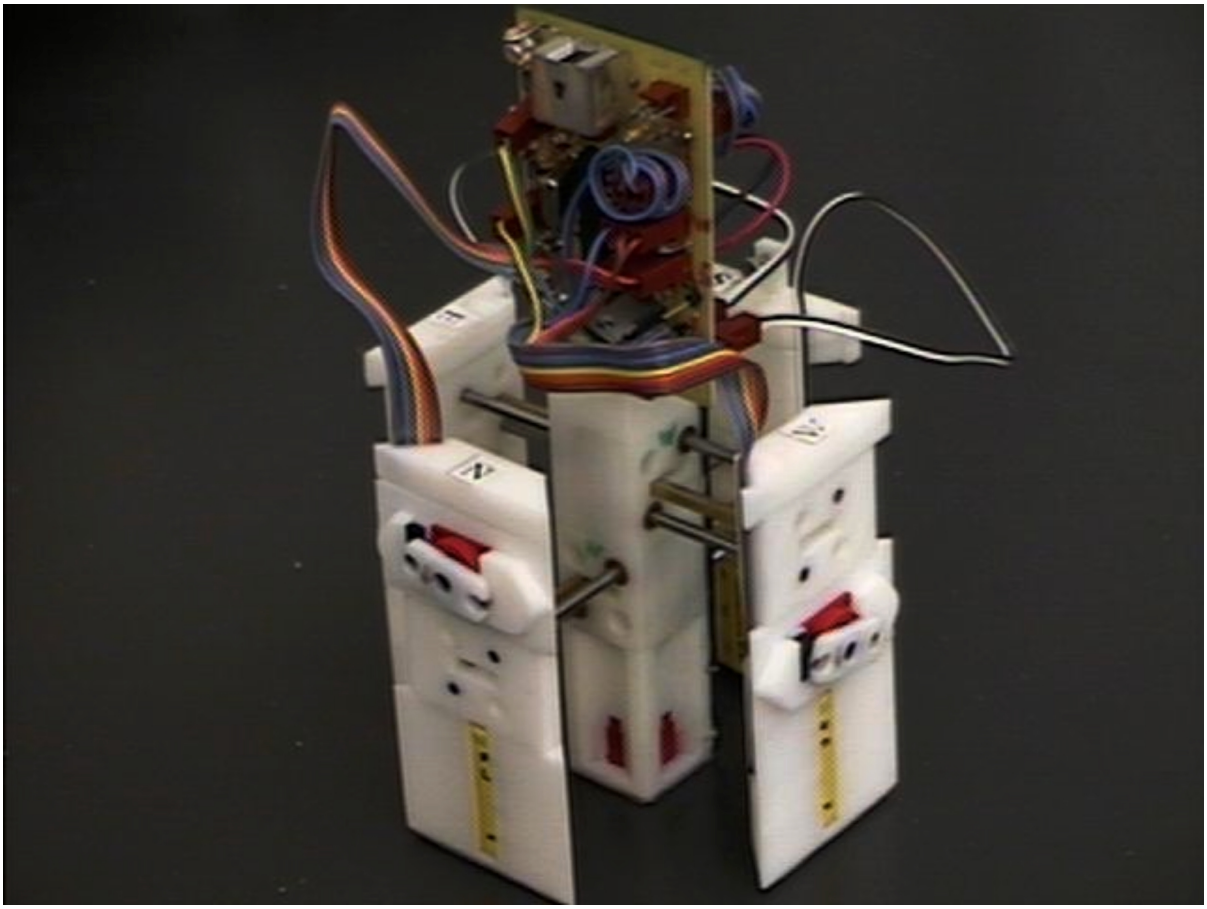


Figure 4.39: The completed Atom, shown expanded.

The Atom is 4 inches square when expanded and 2 inches square when contracted, and it stands 7 inches tall above the operating surface (which is just the Formica veneer of our laboratory workbenches). Thus the Atom meets the *height* and *expanded* design parameters. The Atom weighs about 12 ounces, which is only slightly heavier than what the m_{atom} design parameter calls for (10 ounces). The parts cost per Atom is roughly $250, which is better than our *cost* design parameter of $300.

The next section contains a brief description of the assembly process for the Atom. The measurements that were taken and the Experiments that were performed are described in the following two sections.

4.8.1 Construction

Most of the structural components of the Atom were fabricated out of ABS plastic on DRL's Stratasys FDM1600 Rapid Prototyper (these are the white components in Figure 4.39). The FDM1600 requires about 48 hours to build all components necessary for one Atom. Once all the components have been fabricated they require some post-processing to remove the support material that remains from the build process. This takes about 3 hours.

The Atom does contain a few components which need to be machined: the racks need to be cut to length and drilled, the pinions need to be fitted with setscrews, and the guide rails need to be cut. These operations are easily accomplished in 3 or 4 hours at a machine shop.

Once all the components for the Atom have been gathered, assembly begins. Cyanoacrylate glue is used to bond most of the structural components together. Each vertical stage of the faces and the core is fabricated as a single block on the FDM1600. Guide pins are used to maintain alignment from stage to stage. The racks are bolted to the faces and the guide rails and Rulon bushings are pressed into their respective locations (see Figure 4.27). Mechanical assembly takes about 3 hours

After all four faces and the core have been mechanically assembled, the wiring that connects the sensors and motors on the faces and in the core is applied. Standard ribbon cable is used, secured in place with Cyanoacrylate glue. This is another 3 to 4 hour job.

As mentioned above, the electronics are fabricated on a printed circuit board. We have these boards printed off-site. Assembly and testing of the electronics takes about 2 hours.

Final assembly takes place after the faces and the core are each constructed and wired and the electronics is assembled. Each face is carefully inserted into the core and the expansion motor is activated to draw the faces in. Usually, several tries are necessary before all four faces start themselves with the correct alignment.

In total, about 17 hours of labor are required to construct an Atom. This does not include the 48 hours of FDM1600 build time, which can be largely unsupervised. We were targeting a T_{build} of 10 hours, so we did not meet that goal. But 17 hours did turn out to be acceptable.

4.8.2 Measurements

Four types of measurements were taken: current draw, speed, rigidity, and accuracy:

- **Current Draw** – The Atom was measured to draw about 35mA while powered on but not moving. Activating any of the motors raises the current draw to about 60mA, and if any of the motors becomes stalled current draw jumps to 120mA. These numbers indicate that the time-averaged current draw is clearly better than the I_{atom} design parameter of 150mA.
- **Expansion and Contraction Time** – The Atom takes about 3 seconds to contract and expand under no-load conditions. When pulling another Atom, this time increases somewhat, however it's clear that the expansion speed design parameter was met.
- **Rigidity** – Since the Atom faces are rigid to within about 1° of the face normal when fully extended, we have met the *rigidity* design parameter of 5°.
- **Accuracy** – The extension mechanism of the Atom is accurate to within about 1/32 inch, which meets the *accuracy* design parameter
- **Connector Fault-Tolerance** – The connection mechanism can handle misalignments up to about 1/8 inch in the lateral direction (Atoms sliding relative to the inter-Atomic plane). However, very little (about 1/32 inch) misalignment is tolerable in the normal direction.

4.8.3 Experiments

Two experiments were performed to evaluate the feasibility of using multiple Atoms to demonstrate reconfiguration (the Atom state sequences for these experiments are included in Appendix 4). To facilitate experimentation, a row of 8 static passive connectors was constructed that simulates the surface of a Crystal. The static connectors are placed as they would be for a Crystal composed of 8 Atoms, all in the contracted state. In the descriptions that follow, we will refer to the two real Atoms as **a** and **b**, and we will number the static 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 static connectors.

The first experiment was designed to determine if an Atom could reliably expand and then connect with a neighbor. The procedure was as follows: **a** was expanded and affixed to **0** (at **a.n**). **b** was contracted and affixed to **2** at (**b.n**). **b** was loaded with a state sequence that caused it to expand and then connect with **a** at the **b.w**/**a.e** inter-Atomic interface, as shown in Figure 4.40.

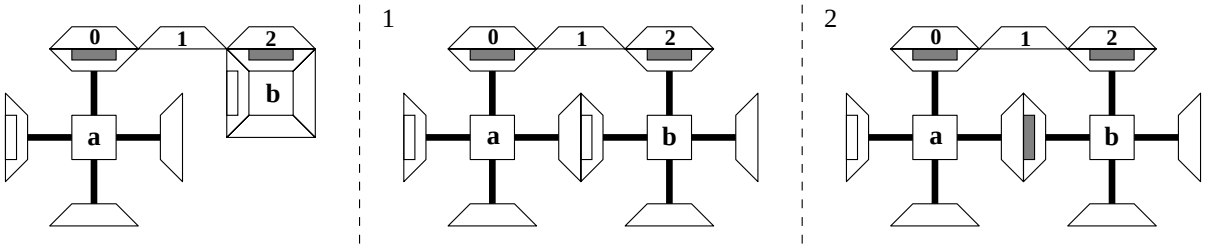


Figure 4.40: In the first experiment, **a** was affixed to **0**, **b** to **2**. **b** was programmed to expand and connect with **a**.

This experiment was successful: it was 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 Atom **a** is 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 an inchworm translation along the static 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 4.41.

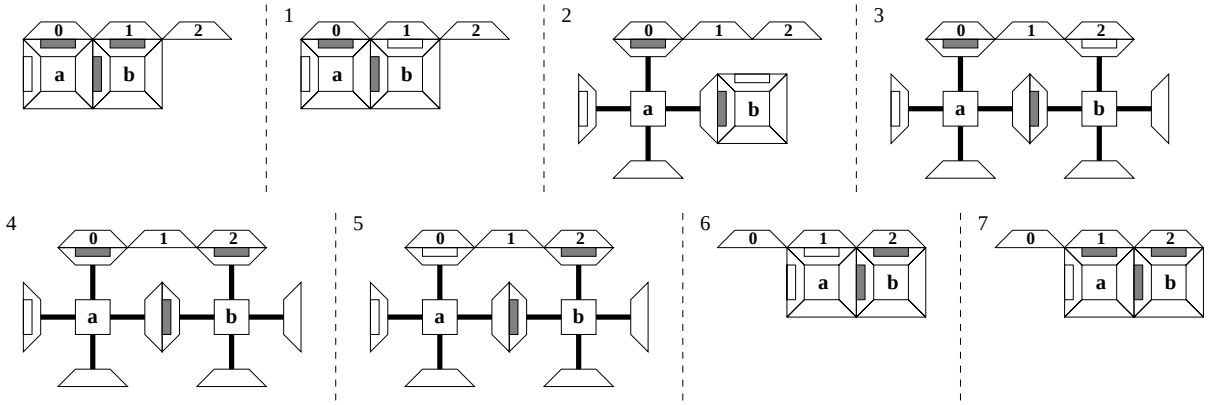


Figure 4.41: The second experiment tests an inchworm propagation algorithm.

This second experiment was partly successful and pointed out the need for some design improvements. In all trials, failure occurred either at step 4 or step 7. These are the two steps where an Atom re-connects with the static connector substrate after being moved. It was clear that these connection operations failed because the active and passive faces involved were too far misaligned in the normal direction (i.e. the direction normal to the inter-Atomic interface plane). As mentioned above, the connection mechanism, as implemented, can only handle about 1/32 inch misalignment in this dimension. During the experimental trials, misalignments of 1/8 to 3/16 inch were commonly observed.

Why was the observed normal misalignment so much higher than the normal direction fault-tolerance of the connection mechanism? The answer to this question is the sum of the following two contributing factors:

- **not enough normal fault-tolerance:** The connection mechanism was designed with good lateral fault tolerance, but the design failed to also provide similarly good normal fault-tolerance.
- **connection mechanism not rigid enough:** As constructed, connected interfaces are free to flex up to about 3°, as shown in Figure 4.42.

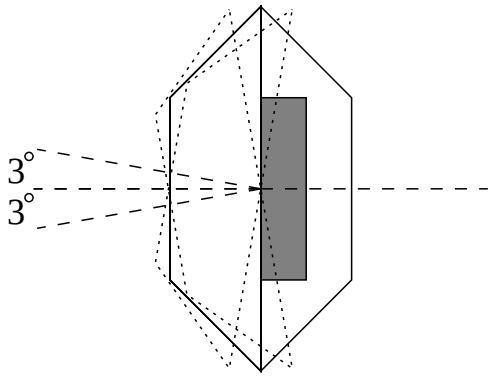


Figure 4.42: The connection mechanism, as designed and constructed, is not rigid enough.

These two factors, combined with the measured Atom face rigidity of 1°, lead to a worst-case scenario in step 4 (and similarly in step 7) as shown in Figure 4.43.

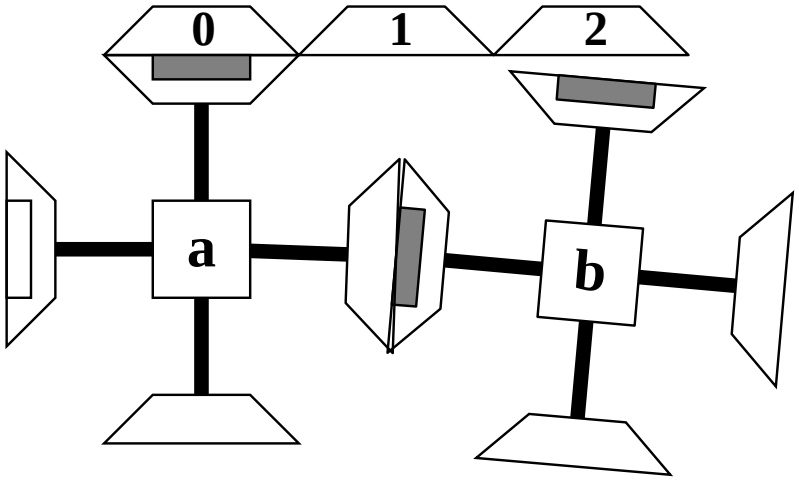


Figure 4.43: A worst-case combination of the Atom face rigidity and connection mechanism rigidity, which causes **b.n** to be too far misaligned from **2**.

A1 References

[1] Y. Cao, A. Fukunaga, A. Kahng, and F. Meng. Cooperative mobile robots: Antecedents and directions. Technical report, UCLA Department of Computer Science, 1995.

[2] Chen and J. Burdick. Enumerating the Non-Isomorphic Assembly Configurations of a Modular Robotic System. To appear in the *International Journal of Robotics Research*.

[3] P. Chew and K. Kedem. Getting around a lower bound for the minimum Hausdorff distance. In *Third Scandinavian Workshop on Algorithm Theory*, eds. O Nurmi and E. Ukkonen, Lecture Notes in Computer Science 621, pp 318--325, Springer Verlag 1992.

[4] G. Chirikjian and J. Burdick. Kinematics of a hyper-redundant robot locomotion with applications to grasping. In *Proceedings of the IEEE International Conference on Robotics and Automat ion*, 1991.

[5] R. Cohen, M. Lipton, M. Dai, and B. Benhabib. Conceptual design of a modular robot. In *Journal of Mechanical Design*, pp. 117-125, March 1992.

[6] T. Fukuda and Y. Kawauchi. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. In *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 662-667, 1990.

[7] G. Hamlin and A. Sanderson. Tetrabot modular robotics: prototype and experiments. In *Proceedings of the IEEE/RSJ International Symposium of Robotics Research*, pp 390-395, Osaka, Japan, 1996.

[8] Kazuo Hosokawa, Isao Shimoyama, and Hirofumi Miura. Dynamics of self-assembling systems --- analogy with chemical kinetics. *Artificial Life*, 1(4), 1995.

[9] D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Matching and Machine Intelligence*, 1993.

[10] S. Kelly and R. Murray, Geometric phases and robotic locomotion. CDS Technical Report 94-014, California Institute of Technology, 1994.

[11] K. Kotay and D. Rus. Navigating 3d steel web structures with an inchworm robot. *Proceedings of the 1996 International Conference on Intelligent Robots and Systems*, Osaka, 1996.

[12] K. Kotay and D. Rus. Task-reconfigurable robots: navigators and manipulators. In *The 1997 International Conference on Intelligent Robots and Systems*, 1997.

[13] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfigurable robotic molecule. In *Proceedings of the 1998 International Conference on Robotics and Automation*, 1998.

[14] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfiguring robotic molecule: design and control algorithms. In *The 1998 Workshop on Algorithmic Foundations of Robotics*, 1998.

[15] K. Kotay and D. Rus. Motion Synthesis for the Self-reconfiguring Robotic Molecule. In *Proceedings of the 1998 International Conference on Intelligent Robots and Systems*, 1998.

[16] K. Kotay and D. Rus. Locomotion Versatility through Self-reconfiguration. *In Robotics and Autonomous Systems*, 1998 (to appear).

[17] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers 1991.

[18] C. McGray and D. Rus. Motion Self-reconfiguring Molecules as 3D Metamorphic Squares. In *Proceedings of the 1998 International Conference on Intelligent Robots and Systems*, 1998.

[19] S. Murata, H. Kurokawa, and Shigeru Kokaji. Self-assembling machine. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, San Diego, 1994.

[20] S. Murata, H. Kurokawa, K. Tomita, and Shigeru Kokaji. Self-assembling method for mechanical structure. In *Artif. Life Robotics*, 1:111--115, 1997.

[21] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D Self-Reconfigurable Structure. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, Leuven, 1998.

[22] R. Murray. Trajectory generation for underactuated systems with applications to robotic locomotion. In *Workshop on Algorithmic Foundations of Robotics*, eds. P. Agrawal, L. Kavraki, and M. Mason, A. K. Peters, 1998.

[23] B. Neville and A. Sanderson. Tetrabot family tree: modular synthesis of kinematic structures for parallel robotics. In *Proceedings of the IEEE/RSJ International Symposium of Robotics Research*, pp 382-390, Osaka, Japan, 1996.

[24] Pamecha, C-J. Chiang, D. Stein, and G. Chirikjian. Design and implementation of metamorphic robots. In *Proceedings of the 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*, Irvine, CA 1996.

[25] Paredis and P. Khosla. Kinematic Design of Serial Link Manipulators from Task Specifications. In *International Journal of Robotic Research*, Vol. 12, No. 3, pp 274--287, 1993.

[26] Paredis and P. Khosla. Design of Modular Fault Tolerant Manipulators. In *The First Workshop on the Algorithmic Foundations of Robotics*, eds. K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, pp 371-383, 1995.

[27] Rus. Self-Reconfiguring Robots. *IEEE Intelligent Systems*, 13(4), 2-5, July/August 1998

[28] K. Tanie and H. Maekawa. Self-reconfigurable cellular robotic system. US Patent 5361186, 1993.

[29] K. Tomita, S. Murata, E. Yoshida, H. Kurokawa, and S. Kokaji. Reconfiguration method for a distributed mechanical system. In *Distributed Autonomous Robotic Systems 2*, pp 17--25, Springer Verlag 1996.

[30] M. Yim. A reconfigurable modular robot with multiple modes of locomotion. In *Proceedings of the 1993 JSME Conference on Advanced Mechatronics*, Tokyo, Japan 1993.

[31] Yoshida, S. Murata, K. Tomita, H. Kurokawa, and S. Kokaji. Distributed Formation Control of a Modular Mechanical System. In *Proceedings of the 1997 International Conference on Intelligent Robots and Systems*, 1997.

[32] D. Rus, M. Vona. Self-reconfiguration Planning with Compressible Unit Modules. In *Proceedings of the 1999 International Conference on Robotics and Automation*, Detroit, MI 1999.

[CLR] Cormen, Leiserson, Rivest. *Introduction to Algorithms*. McGraw-Hill 1990.

A2 Lego Mini-Motor Measurements

In this Appendix we describe the Lego toy Mini-Motor which is used as the actuator for both the expansion and the connection mechanism in the Atom. We then present the results of an experiment to measure the performance of the motor.

A2.1 The Lego Mini-Motor

The Mini-Motor is a component of the Lego "Technics" system. The Mini-Motor is available from Lego's shop at home service as part number 5119 for $11. The Mini Motor, depicted in Figure A2.1, is roughly a 5/8 inch cube and weighs about 0.3 ounces.

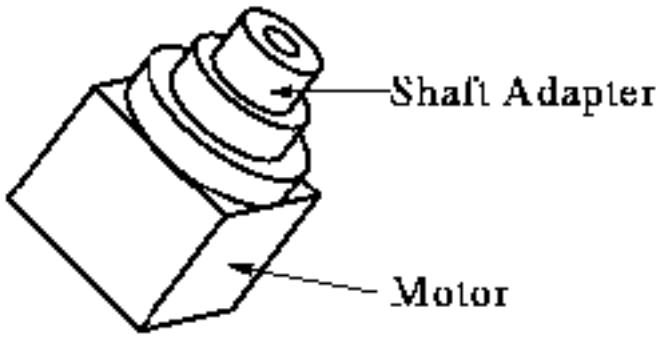


Figure A2.1: The Lego Mini-Motor and attached output shaft adapter.

Electrical connections are made to the bottom surface of the motor. The motor rotary output is through a pancake-style disk with a central nipple on the top surface. The motor is supplied with an adapter that friction mounts onto the output disk and accepts a standard Lego shaft on the other side. A 1/8 inch diameter steel shaft will fit into the orifice meant for the Lego shaft if it is glued in place. The friction fit of the adapter onto the motor is not very tight and can be glued as well.

Disassembly reveals that the motor consists of a low mass rotor, a strong field magnet (likely rare-earth), and two stages of gear reduction. The reducers, which are similar to

harmonic drives, consist of eccentrically driven circular gears inside slightly larger internal gears.

While the motor is surprisingly strong for its size, it does have one major drawback. If the friction fit of the output adapter is bypassed, as described above, mechanical binding becomes a problem. If the motor is driven to a hard mechanical stall it becomes very difficult to re-activate it, even in reverse, until the mechanical stop is physically removed. This was an important consideration in the design of the Atom.

A2.2 Experimental Procedure

No detailed performance data is available for the Lego Mini-Motor. Thus, in order to determine its torque and current characteristics, an experiment was performed. The motor was coupled to a spindle to which various weights were attached. A tachometer was also connected to the spindle to monitor the motor's speed.

The stall torque was determined by gradually reducing the applied weight until the motor began to spin. After that, the weight was reduced in constant decrements. For each weight, the motor speed and current draw were recorded.

The motor was operated at 12V DC, regulated. When used in the Lego toy, the motor is normally operated at 9V DC, unregulated.

A2.3 Experimental Results

The motor stall torque was measured at about 2.0 oz-in, at which it drew about 110 mA. Below that, the torque speed curve was measured as shown in figure A2.2.

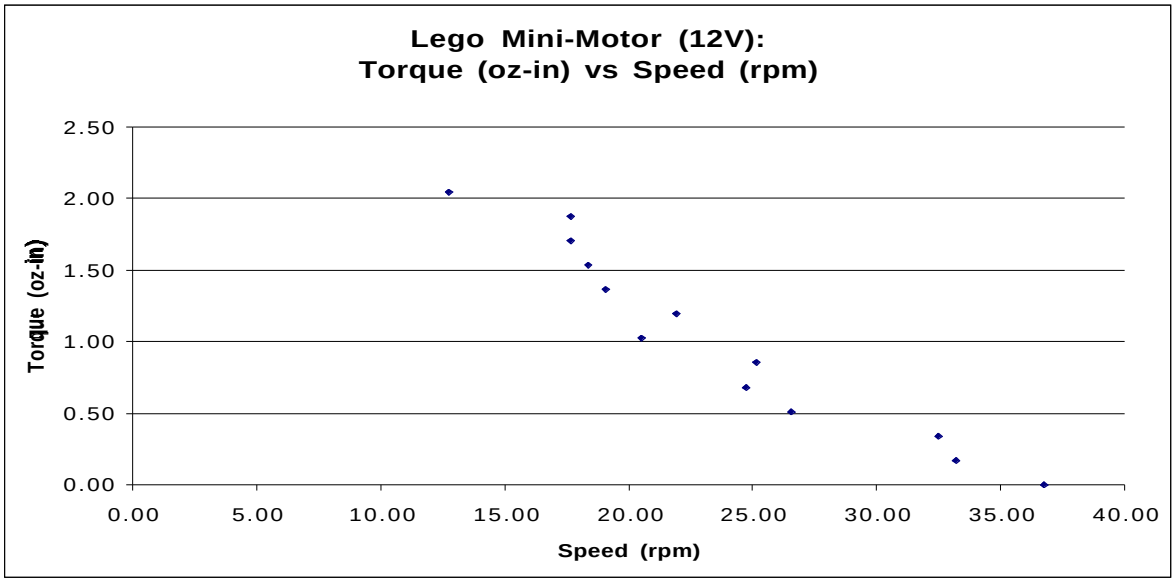


Figure A2.2: Lego Mini-Motor torque-speed performance data.

For each tested weight, the current draw was also recorded, as shown in figure A2.3.

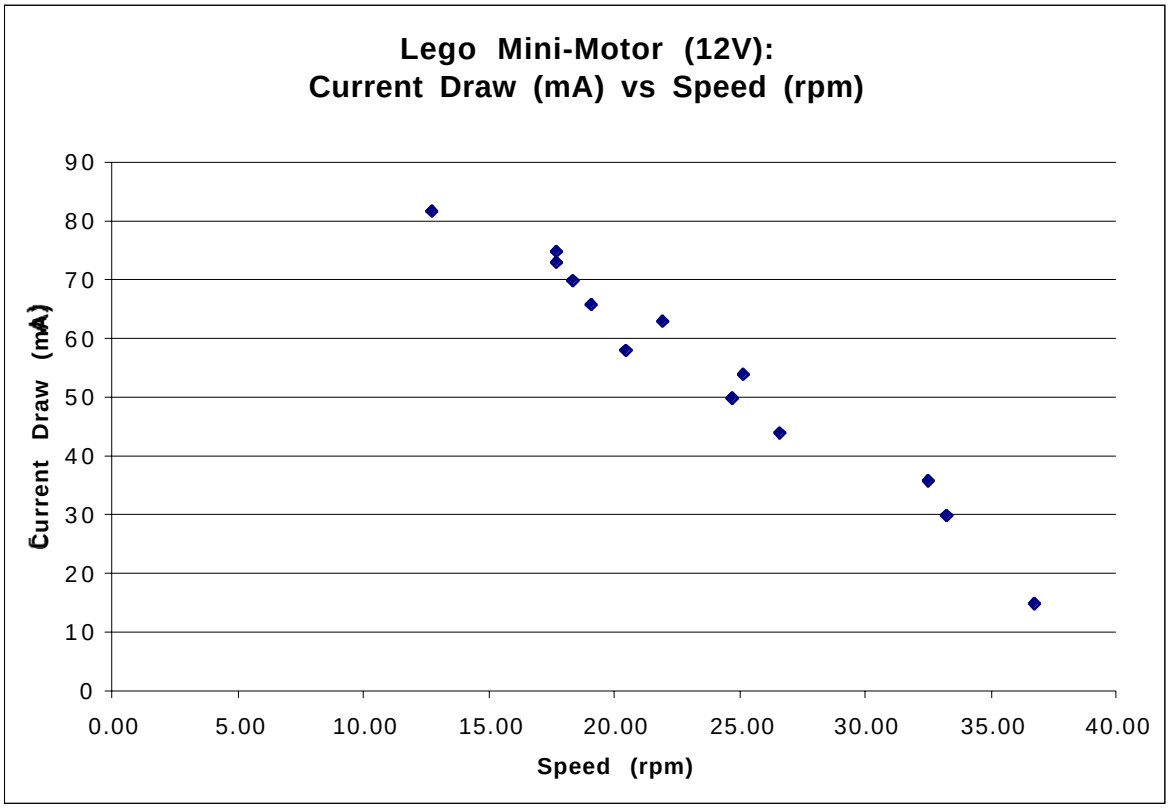


Figure A2.3: Lego Mini-Motor current-speed performance data.

The motor efficiency was calculated as the ratio of mechanical output power (torque · speed) to electrical input power (voltage · current), as shown in Figure A2.4.

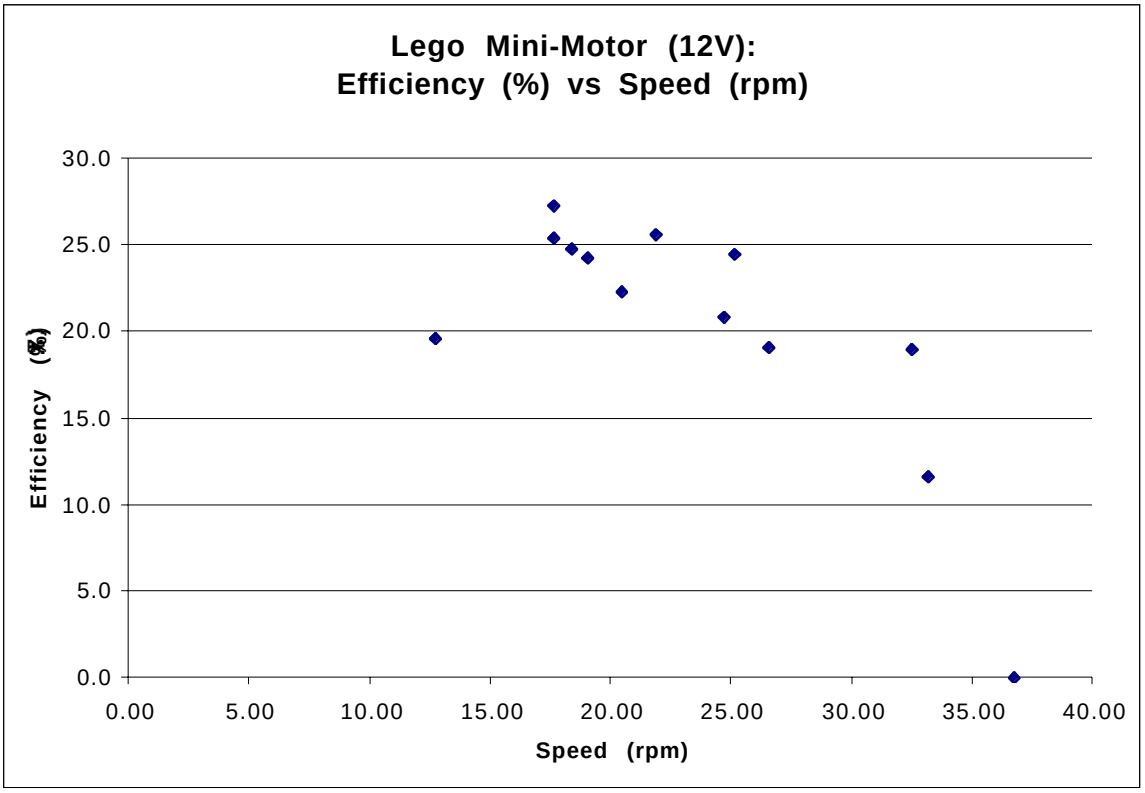
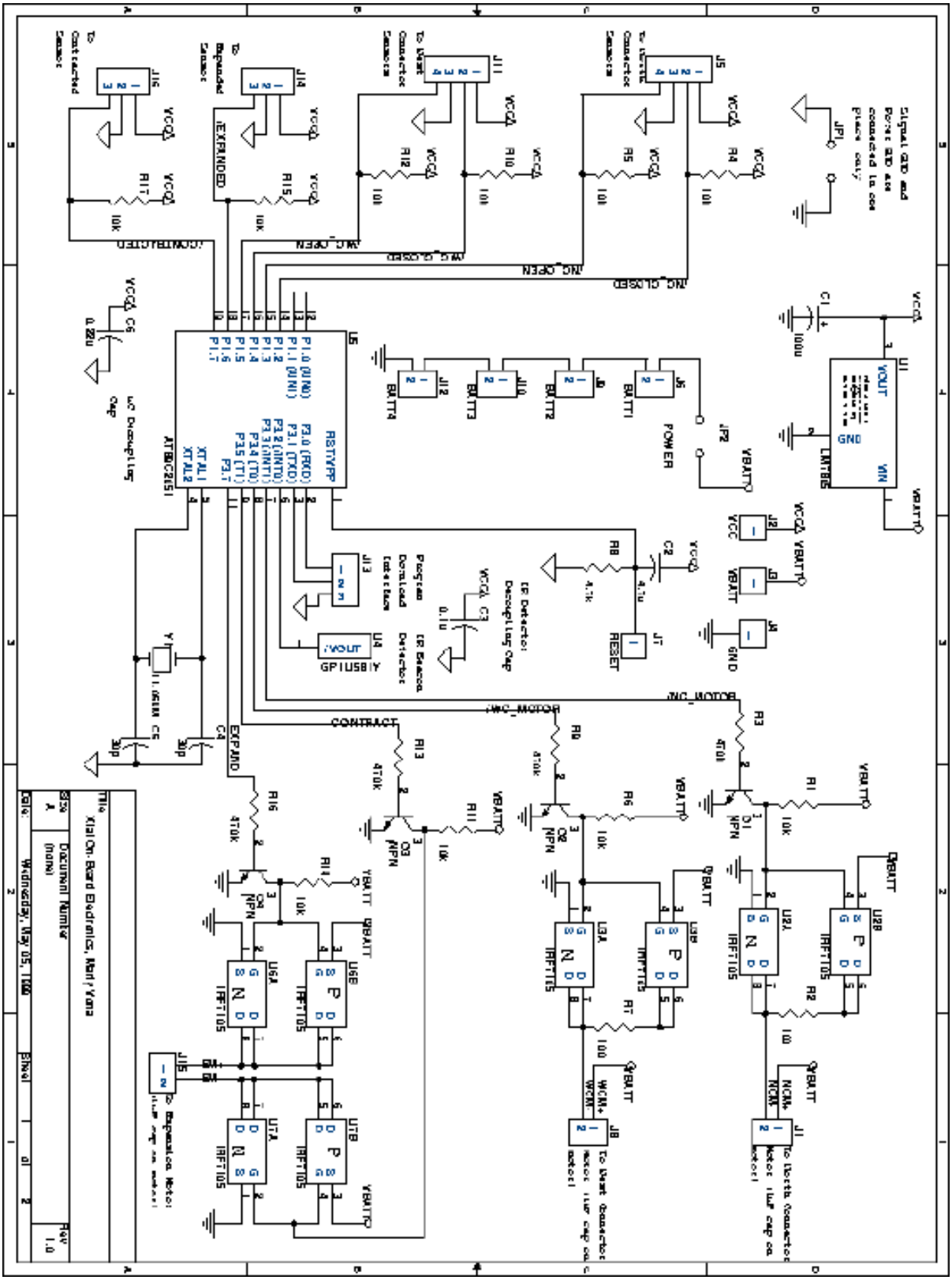


Figure A2.4: Lego Mini-Motor efficiency performance data.

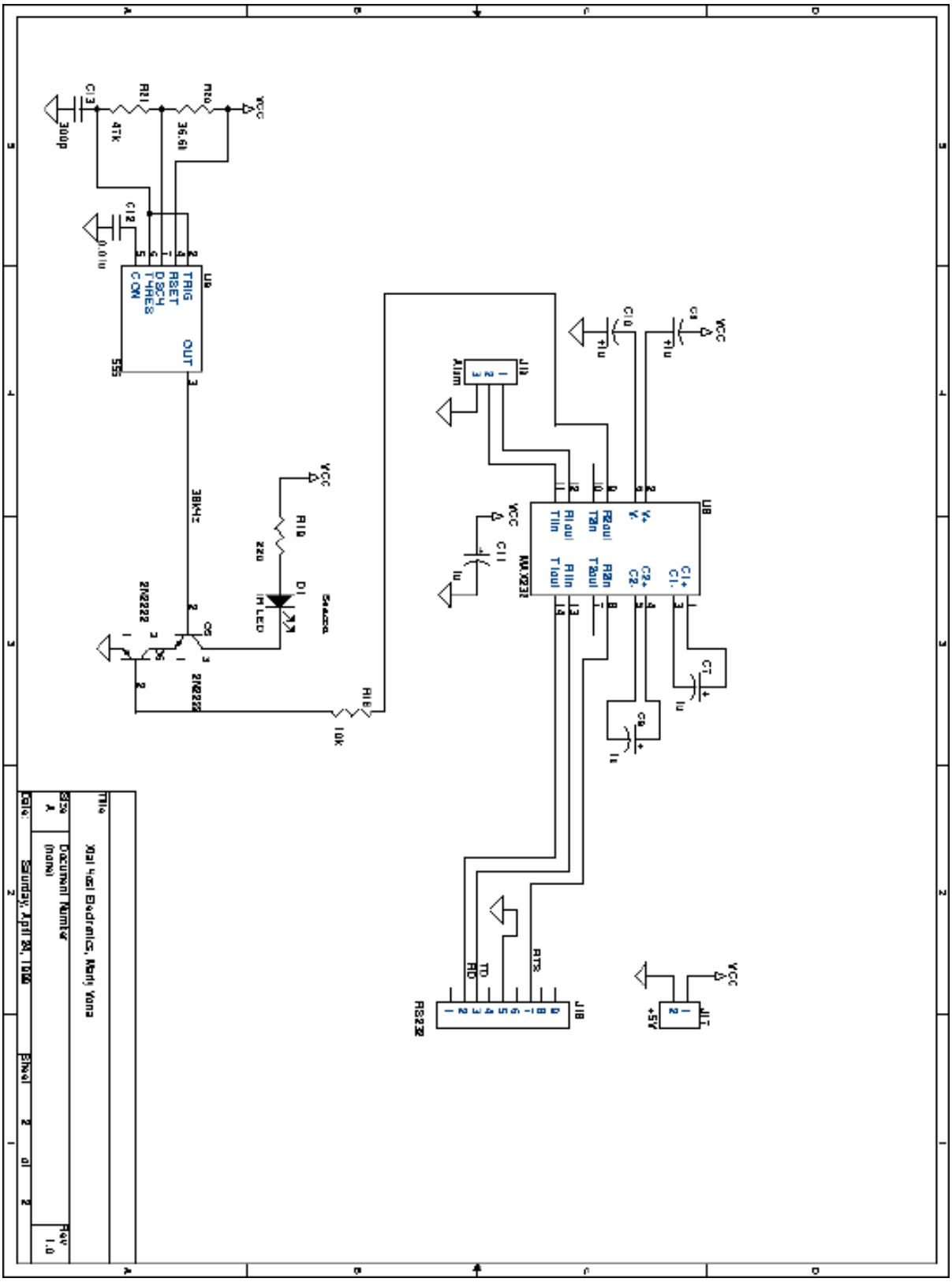
A3 Schematic Diagrams

In this Appendix we present two schematic diagrams: one for the Atom on-board electronics and one for the Atom host electronics. The diagrams follow on the next two pages.

A3.1 On-Board Electronics



A3.2 Host Electronics



A4 Listings

This Appendix includes three listings: the dogcouch relative deformation, the Atom hardware state sequences used for experiments, and the Melt-Grow planner input for the table-to-chair reconfiguration.

A4.1 Dogcouch Relative Deformation

In this section we present the xtalsim source code for the dog-couch simulation illustrated in Chapter 2. This code is included here to demonstrate how the simulator input language is used. The code is contained in two files: *dogcouch.dfm* is the main simulation file with the initial Crystal specification (the dog) and the list of updates which change the dog into a couch. *dfmcommon.h* is an auxiliary file that defines some routines used in dogcouch.dfm.

```
/* File: dogcouch.dfm
 * Created: 7/18/98
 * Author: Marty Vona
 *
 * Abracadabra!
 *
 */

#include "dfmcommon.h"

(
  (
    /* the initial Crystal (dog) */

    /* body */
    (atom (-4 0 0) fbb eee)
    (atom (0 0 0) bbf eee)
    (atom (4 0 0) bbb eee)
    (atom (8 0 0) bbb eee)
    (atom (12 0 0) bbb eee)

    (atom (-4 4 0) fbf eee)
    (atom (0 4 0) bbf eee)
    (atom (4 4 0) bbf eee)
    (atom (8 4 0) bbf eee)
    (atom (12 4 0) bbf eee)

    /* rear paws */
    (atom (4 0 4) fbb eee)
    (atom (8 0 4) bbb eee)
```

```
  (atom (12 0 4) bbb eee)

  (atom (4 0 -4) fbf eee)
  (atom (8 0 -4) bbf eee)
  (atom (12 0 -4) bbf eee)

  /* front paws */
  (atom (-12 0 4) fbf eee)
  (atom (-8 0 4) bbf eee)
  (atom (-4 0 4) bbb eee)

  (atom (-12 0 -4) fbf eee)
  (atom (-8 0 -4) bbf eee)
  (atom (-4 0 -4) bbf eee)

  /* neck */
  (atom (-4 8 0) fbf eee)

  /* head */
  (atom (-8 12 0) fff eee)
  (atom (-4 12 0) bbb eee)

  /* ears */
  (atom (-4 12 4) ffb eee)
  (atom (-4 12 -4) fff eee)

  /* tail */
  (atom (12 8 0) fbf eee)

)

(
  /* the updates (dog->couch) */

  /* move tail up */
  SCRUNCH_COLN(<12 8 0>, y)
  ((bond <12 4 0> x))
  SCRUNCH_COLP(<0 4 0>, x)
  ((expand <12 * 0> y))
  ((bond <12 * 0> x))
  ((free <[3 4] * 0> x))
  ((free <0 * 0> x))
  ((contract <0 * 0> y))
  RELAXN(<8 4 0>, x)
  ((free <[0 4] 4 0> x))
  ((expand <0 [-1 3] 0> y))
  ((bond<0 * 0> x) (bond <4 * 0> x))

  /* pull paws back */
  ((free <[inf 0] 0 *> y) (free <[4 8] 0 0> y) (free <[4 8] 0 * ??b> z))
  ((contract <[4 8] * 0> x))
  ((bond <* 0 [-4,8 8] ?f?> y))
  ((free <0 0 * ??b> z))
  ((expand <[7 9] * 0> x))
  ((bond <* 0 * f??> x) (bond <* 0 * ?f?> y) (bond <* 0 * ??f> z))

  /* pull head in */
  SCRUNCH_COLN(<12 0 0>, x)
  ((free <[0 4] [0 4] 0 b??> x) (free <0 0 [0 4] ??b> z))
  ((contract <0 [0 4] 0> y))
  ((bond <-4 0 0> y))
```

```
    ((free <0 4 0> x))
    ((expand <0 [-1 3] 0> y))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    RELAX_COLN(<12 0 0>, x)
    ((free <-4 0 0> x) (free <4 * 0> x) (free <[-4 0] 0 * ??b> z))
    ((contract <[-4 0] [0 4] 0> y))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))

    /* move ears to either end to make armrests */
    SCRUNCH_COLN(<12 4 0>, x)
    ((free <-4 4 0> z) (free <-4 4 4> y) (free <[inf 4] 4 0> y))
    ((expand <[7 9] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <-4 4 0> x) (free <[-4 8] 4 *> y))
    ((contract <[4 8] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <[0 4] 4 0> y) (free <0 4 0> z))
    ((expand <[7 9] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <0 4 0> x) (free <[0 8] 4 *> y))
    ((contract <[4 8] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <4 4 0> y) (free <4 4 0> z))
    ((expand <[7 9] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <[-4 8] 4 0> y) (free <4 4 0> z) (free <-4 4 0> x))
    ((contract <[-4 8] 4 0> x))
    ((expand <[-4 0] [-1 1] 0> y))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <[-4 0] 4 0> y))
    ((contract <[-4 0] 4 0> x) (expand <[7 9] 4 0> x))
    ((bond <4 4 0> z) (free <4 4 -4> y) (free <12 4 0> y))
    ((expand <[-5 1] 4 0> x))
    ((bond <12 4 -4> y) (free <12 4 0> z))
    ((contract <[0 12] 4 0> x))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))
    ((free <12 4 0> x) (free <12 0 [-4 0]> x) (free <12 0 [-4 0]> y)
     (free <12 4 0> y))
    ((contract <12 0 [-4 0]> z))
    ((free <8 0 4> z) (free <8 0 0> y) (free <8 [0 4] 0> x))
    ((contract <8 0 0> z))
    ((bond <12 4 -1> x) (free <12 4 -1> y))
    ((expand <[8 12] 0 [-1 1]> z))
    ((bond <* * * f??> x) (bond <* * * ?f?> y) (bond <* * * ??f> z))

    /* pull in rear underside of couch */
    ((free <[-8 12] [0 4] [-4 0] ?b?> y))
    ((contract <[-8 12] 0 [-4 0]> z))
  )
)
```

```
/* File: dfmcommon.h
 * Created: 7/17/98
 * Author: Marty Vona
 *
 * the C preprocessor: LOVE IT
 */

#ifndef DFMCOMMON_H
```

```
#define DFMCOMMON_H

#define ODIMx0 y
#define ODIMx1 z
#define ODIMy0 x
#define ODIMy1 z
#define ODIMz0 x
#define ODIMz1 y

/* return one of the dimensions orthogonal to dim */
#define ODIM(dim, which) ODIM ## dim ## which

#define DTOx(in_dim, in_odim0, in_odim1) in_dim in_odim0 in_odim1
#define DTOy(in_dim, in_odim0, in_odim1) in_odim0 in_dim in_odim1
#define DTOz(in_dim, in_odim0, in_odim1) in_odim0 in_odim1 in_dim

/* order a dimension triple */
#define DTO(dim, in_dim, in_odim0, in_odim1) \
  DTO ## dim (in_dim, in_odim0, in_odim1)

#define AINxb b??
#define AINyb ?b?
#define AINzb ??b

#define AINxf f??
#define AINyf ?f?
#define AINzf ??f

#define AINxc c??
#define AINyc ?c?
#define AINzc ??c

#define AINxe e??
#define AINye ?e?
#define AINze ??e

/* build an active-in wildcard */
#define AIN(dim, action) AIN ## dim ## action

/* create a two-atom scrunch behind a in dim */
#define SCRUNCHN(a, dim) \
  _SCRUNCHN(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _SCRUNCHN(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [-8 -1], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [-8 -1], 0, [0 4]) AIN(od1, b)>> od1) \
    (free <@ a <DTO(dim, -8, 0, 0) AIN(dim, b)>> dim) \
  ) \
  ((contract <@ a <DTO(dim, [-8 -1], 0, 0) AIN(dim, e)>> dim))

#define SCRUNCHP(a, dim) \
  _SCRUNCHP(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _SCRUNCHP(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [1 8], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [1 8], 0, [0 4]) AIN(od1, b)>> od1) \
    (free <@ a <DTO(dim, 12, 0, 0) AIN(dim, b)>> dim) \
  ) \
  /* shrink */ \
```

```
  ((contract <@ a <DTO(dim, [1 8], 0, 0) AIN(dim, e)>> dim))

/* relax a two-atom scrunch behind a in dim */
#define RELAXN(a, dim) \
  _RELAXN(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _RELAXN(a, dim, od0, od1) \
  ((expand <@ a <DTO(dim, [-5 -1], 0, 0) AIN(dim, c)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [-8 -1], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [-8 -1], 0, [0 4]) AIN(od1, f)>> od1) \
    (bond <@ a <DTO(dim, -8, 0, 0) AIN(dim, f)>> dim) \
  )

#define RELAXP(a, dim) \
  _RELAXP(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _RELAXP(a, dim, od0, od1) \
  ((expand <@ a <DTO(dim, [1 5], 0, 0) AIN(dim, c)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [1 8], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [1 8], 0, [0 4]) AIN(od1, f)>> od1) \
    (bond <@ a <DTO(dim, 12, 0, 0) AIN(dim, f)>> dim) \
  )

/* create a two-atom scrunch behind a in dim pulling along the whole
trailer */
#define SCRUNCH_COLN(a, dim) \
  _SCRUNCH_COLN(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _SCRUNCH_COLN(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [inf -1], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [inf -1], 0, [0 4]) AIN(od1, b)>> od1) \
  ) \
  ((contract <@ a <DTO(dim, [-8 -1], 0, 0) AIN(dim, e)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [inf -6], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [inf -6], 0, [0 4]) AIN(od1, f)>> od1) \
  )

#define SCRUNCH_COLP(a, dim) \
  _SCRUNCH_COLP(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _SCRUNCH_COLP(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [1 inf], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [1 inf], 0, [0 4]) AIN(od1, b)>> od1) \
  ) \
  ((contract <@ a <DTO(dim, [1 8], 0, 0) AIN(dim, e)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [6 inf], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [6 inf], 0, [0 4]) AIN(od1, f)>> od1) \
  )

/* relax a two-atom scrunch behind a in dim pushing along the whole
trailer */
#define RELAX_COLN(a, dim) \
  _RELAX_COLN(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _RELAX_COLN(a, dim, od0, od1) \
```

```
  ( \
    (free <@ a <DTO(dim, [inf -6], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [inf -6], 0, [0 4]) AIN(od1, b)>> od1) \
  ) \
  ((expand <@ a <DTO(dim, [-5 -1], 0, 0) AIN(dim, c)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [inf -1], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [inf -1], 0, [0 4]) AIN(od1, f)>> od1) \
  )

#define RELAX_COLP(a, dim) \
  _RELAX_COLP(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _RELAX_COLP(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [6 inf], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [6 inf], 0, [0 4]) AIN(od1, b)>> od1) \
  ) \
  ((expand <@ a <DTO(dim, [1 5], 0, 0) AIN(dim, c)>> dim)) \
  ( \
    (bond <@ a <DTO(dim, [1 inf], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [1 inf], 0, [0 4]) AIN(od1, f)>> od1) \
  )

/* relax a two-atom scrunch behind a and simultaneously create a two atom
 * scrunch in front of a */
#define SCRUNCH_SWAPN(a, dim) \
  _SCRUNCH_SWAPN(a, dim, ODIM(dim, 0), ODIM(dim, 1))

#define _SCRUNCH_SWAPN(a, dim, od0, od1) \
  ( \
    (free <@ a <DTO(dim, [0 8], [0 4], 0) AIN(od0, b)>> od0) \
    (free <@ a <DTO(dim, [0 8], 0, [0 4]) AIN(od1, b)>> od1) \
  ) \
  ( \
    (contract <@ a <DTO(dim, [1 8], 0, 0) AIN(dim, e)>> dim) \
    (expand <@ a <DTO(dim, [-5 -1], 0, 0) AIN(dim, c)>> dim) \
  ) \
  ( \
    (bond <@ a <DTO(dim, [-4 4], [0 4], 0) AIN(od0, f)>> od0) \
    (bond <@ a <DTO(dim, [-4 4], 0, [0 4]) AIN(od1, f)>> od1) \
  )

#endif
```

A4.2 Atom Hardware Experiment State Sequences

Here we present the state sequences used for the two Atom hardware experiments. The code for these sequences is as follows:

Even *states are connection updates:*
0 : Free North, Free West
1: Bond North, Free West
2: Free North, Bond West
3: Bond North, Bond West

Odd *states are expansion updates:*
0: Contracted
1: Expanded

A4.2.1 Experiment 1

Atom **a**:`111111`
Atom **b**:`113110`

A4.2.2 Experiment 2

Atom **a**:`10111111001`
Atom **b**:`30202131303`

A4.3 Table-To-Chair Melt-Grow Planner Input

The Melt-Grow planner input for the table-to-chair reconfiguration consists of two lists of Grain centroid coordinates. The first list defines the table, and the second defines the chair:

```
(
  (-1 -2)
  (0 -2)
  (1 -2)
  (2 -2)
  (3 -2)
  (4 -2)
  (5 -2)
  (0 -1)
  (0 0)
  (4 -1)
  (4 0)
)
(
  (0 -2)
  (1 -2)
  (2 -2)
  (3 -2)
  (0 -1)
  (0 0)
  (3 -1)
  (3 0)
  (3 -3)
  (3 -4)
  (3 -5)
)
```