Distributed motion planning for modular robots with unit-compressible modules

Zack Butler, Sean Byrnes and Daniela Rus
Dept. of Computer Science, Dartmouth College
zackb@cs.dartmouth.edu, snb23@cornell.edu, rus@cs.dartmouth.edu

Abstract

The ability of self-reconfigurable robots to solve a variety of robot tasks comes in part from their use of a large number of modules. Effective use of these systems requires parallel actuation and planning, both for efficiency and independence from a central controller. This paper presents the PacMan algorithm, a technique for distributed actuation and planning. This algorithm was developed for systems with unit-compressible modules, such as the crystalline robot. We also describe some analytical properties of the PacMan planning and actuation, and discuss simulation and hardware experiments.

1 Introduction

Self-reconfigurable modular robots are systems that adapt to the task at hand by changing their overall shape through the use of many independently actuating modules. A self-reconfiguring robot could, for example, form into a legged robot for rapid locomotion, then reform into a snake to navigate a tunnel, while using some of its modules to form a gripper to manipulate an object. Several self-reconfigurable systems have been proposed [1, 4, 6, 2, 8, 10, 5]. In all these systems, the actuation is distributed. However, if the control is centralized, the overall efficiency of the system may be quite low. We seek to enable decentralized approaches to path planning and actuation for self-reconfigurable systems. These algorithms lead to more efficient systems by supporting parallelism, taking the burden of low-level motion planning off of a central controller.

Previous work in self-reconfigurable robotics has often concentrated on the difficult task of creating such machines. Once built, several algorithms have considered how to automatically generate a reconfiguration plan for a specific modular robot, given a centralized controller. For example, Kotay [3] uses a graph-search based planner that achieves self-reconfiguration by moving one module at a time. Pamecha *et al.* [7] use a simulated annealing approach to develop global reconfiguration plans. None of these algorithms explicitly exploit the parallel actuation capability[1].

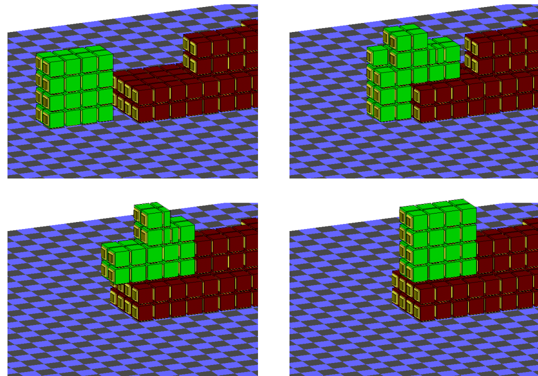

Figure 1: A sequence of screenshots of a manually planned crystal reconfiguration in which the light gray robot climbs stairs by shape morphing.

Previous work on distributed self-reconfiguration planning includes that of Tomita *et al.* [9], who described a method for shape formation for a simple modular system in which modules attempt to attain the correct sets of connections between modules. It is not clear how this approach might extend to more complex systems. Recent work by Yim *et al.* [11] presents a method for a class of systems that uses local greedy control to attempt to converge to a globally known goal shape. Because of the nature of the control, this method cannot guarantee global goal achievement, but seems to perform well in practice.

The contribution of this paper is the PacMan algorithm, a simple scheme for distributed planning and actuation for self-reconfiguring robots with unit-

[1] Kotay describes a method of overlapping sequential motion plans in time to create efficient parallel actuation.

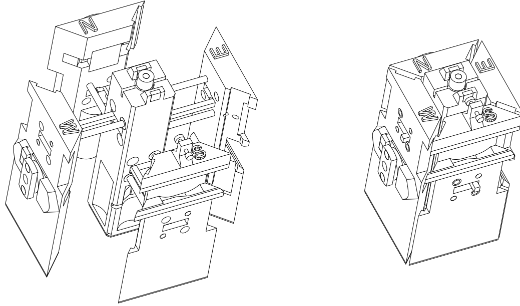

Figure 2: A single module (atom) of the crystalline robot, expanded and contracted.

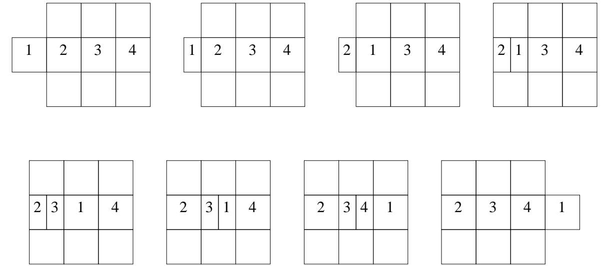

Figure 3: A simple PacMan reconfiguration. The numbers represent module IDs, not physical modules.

compressible modules. This algorithm allows for natural parallelism in the performance of the reconfiguration, making it potentially much faster than a central controller for systems with large numbers of modules.

2 The Crystalline Atomic robot

The distributed algorithms described in this paper are applicable to robots with unit-compressible modules. One specific instantiation is the *Crystalline Atomic* robot, or *crystal*, developed in our lab [8]. It is made up of a collection of *Atoms*, which are square modules that can connect to each other and can expand and contract by a factor of two. One atom is shown in Fig. 2. In this system, a single module cannot move on its own. The expansion and contraction of neighboring modules will cause it to move with respect to its neighbors, while attaching and detaching from neighbors allows for more complex global reconfigurations. A sample reconfiguration is shown in Fig. 1. Similar three-dimensional modules have been developed by Yim *et al.*, and we are currently extending our algorithms to the 3-D case.

One important feature of the crystal is that all atoms are identical. Any module can therefore occupy any position in the goal configuration. When a goal shape is specified, which atom occupies which location in that shape can be determined at run-time. This effect is what enables the PacMan technique, and can lead to very efficient reconfiguration algorithms, as described previously [8]. One important thing to keep in mind is that the style of actuation also means that disconnections occur in the interior of the crystal, and can easily lead to fragmentation (especially near small protrusions) if proper care is not taken.

3 The PacMan concept

In this paper we develop a distributed algorithm for self-reconfiguration, in which the individual modules plan and actuate a given reconfiguration in parallel using only local information and local communication. A planner run by the atoms will develop paths for each module that needs to move to implement the goal shape. The paths are then actuated by the atoms in a parallel distributed fashion.

The PacMan algorithm was inspired by the video game of the same name, in which the main character eats "pellets" as it moves around the board. The algorithm uses data structures called pellets as a way of marking the path that each module should follow to perform its part of the reconfiguration. This representation is well suited to the unit compressible actuation concept, in which motion of the modules takes place in the interior of the structure, rather than on the surface. Under PacMan, a single physical module does not follow the entire path. Instead, a module will virtually travel along the path marked by its pellets. To do this, it exchanges its identity with other modules along the path, while the physical modules move only locally. A simple PacMan reconfiguration is shown in Fig. 3. Additionally, by marking each pellet with the identity of the module that is to "eat" it, paths for several modules can coexist in the crystal (see Fig. 4a). The existence of multiple paths in turn allows for several modules to perform simultaneous reconfigurations without relying on a central clock.

More specifically, the PacMan process is two-fold. First, a path is planned for each module in a distributed fashion, using the planner described in Sec. 4. The result of planning is a set of pellets distributed through the atoms of the robot, as in Fig. 4. Once the pellets are in place, the actuation will happen asynchronously, directed by the algorithm presented in Sec. 5. Each atom looks for pellets and "eats" them without adhering to a strict schedule. This means that the intermediate structure of the crystal will be undetermined, but the final structure will be as specified.

The distinction between planning and actuation in this algorithm suggests that a centralized planner

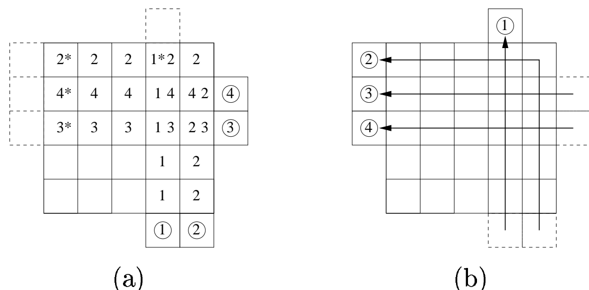

Figure 4: An example of PacMan pellets: (a) a crystal with four moving atoms (with circled ID numbers), their pellets shown by number in the other atoms, (b) the resulting structure.

could be used to create pellets and assign them to the appropriate modules. The modules could then perform PacMan actuation in parallel. This is reasonable for a small system, since the constraints on paths can be complicated and it is easier to generate paths in a centralized way. However, this would require the arrangement of the atoms to be known at all times.

4 Distributed planning

To plan a reconfiguration, the algorithm operates in two stages. In the first stage, the modules determine the local difference between the current shape and the desired one. In the second stage, paths are planned for modules that are not currently part of the desired shape to fill in the holes. These stages may happen a number of times, resulting in a series of waves of reconfiguration. In each wave, a layer of modules moves to the edge of the robot toward the desired shape.

For a given reconfiguration, we define *target* atoms as those adjacent to unfilled goal locations, and *spare* atoms as those not currently part of the goal shape. We further define *mobile* atoms as spare atoms that are on the edge of the system and can be removed without disconnecting the structure.

4.1 Initialization

First of all, it must be determined how the current configuration relates to the goal configuration. To do this, we propose a simple algorithm which was originally used in a centralized fashion. This technique can be easily implemented in a distributed way, although it requires significant data transfer between modules.

In this technique, a matrix representation of the desired shape S_d is passed throughout the modules and compared to the current system. One atom is chosen as the initiator of the process, and this atom is given a description of S_d along with its location in S_d. It then marks its location as visited and checks each of its neighbors (in S_d and in hardware). This check results in one of three cases: if an atom is present, it is given S_d, and will mark itself as a spare atom if not in S_d. If an atom is not present but one is specified for that location in S_d, the atom performing the checks knows that it is a target atom. This process ensures that all spare and target atoms will be identified.

4.2 Path planning

In the second phase, each target atom initiates a search through the atoms for an available mobile atom to become its neighbor. The search is done using *plan-pellets*, which represent one step of a potential path. They are propagated through the atoms of the crystal, performing a depth-first search for a mobile atom in a distributed fashion. When a mobile atom is found, the plan-pellets (now forming a chain from mobile to target atom) are turned into *path-pellets*, which the mobile atom will "eat" to virtually move to the goal location.

Due to the structure of the modules, an atom cannot follow arbitrary paths through the crystal, and some classes of paths that are physically feasible cannot be followed under PacMan actuation, so the planning process must take these restrictions into account. First of all, when a path turns a corner, there is a minimum number of additional atoms that must be present, as well as a minimum length for each path segment leading to and from the corner. Path-pellets contain two counters t_l and t_r to keep track of these data, as described below. Secondly, when a path travels along the edge of the crystal, any atoms that lie by themselves adjacent to that edge (referred to as *dangling* atoms, examples of which are shown in Fig. 5) will become disconnected as the atoms along the path contract and expand. Therefore, the planner must (a) provide pellets for the dangling atoms to move off of that edge and (b) leave one edge of the crystal free of paths as a place for these extra atoms to hide. The latter is done through the use of a set of edge counters E in each plan-pellet, while the former is handled by spawning additional plan-pellets.

Pellet propagation works as follows, where d is the direction in which the pellet was last sent:

- If mobile, set ID to my ID and return to sender.
- Check edge flags: if the atom belongs to an edge currently not flagged, set the flag. If $\text{sum}(E) = 4$, return rejection to sender.
- If a neighbor exists in d, propagate pellet in d, increment t_l if there is a neighbor on the left (and reset it to zero if not), similarly for t_r.
- Otherwise, if $t_l \geq 2$ and there is a neighbor on the left, propagate the plan-pellet to that atom, with

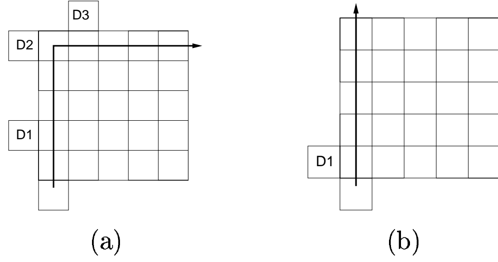

Figure 5: Dangling atoms that (a) can and (b) cannot be correctly handled under the current algorithm.

d set appropriately and $t_l = t_r = 0$.

- Similarly for a neighbor on the right.
- If none of these are possible, return rejection.

Backtracking (i.e. upon receipt of a rejection notice) works as follows:

- If rejection direction is opposite d: turn left as described above if possible, otherwise turn right if possible, otherwise return rejection in opp(d).
- If rejection direction is counter-clockwise from d, try a right turn if possible, otherwise return rejection in opp(d).
- Otherwise return rejection in opp(d).

When a path-pellet is received by a mobile atom, and either t_l or t_r is at least 2, the mobile atom fulfills this path request, and sends a message back to the atom that gave it the plan-pellet. This message causes the plan-pellet to be turned into a path-pellet with the mobile atom's ID, and the message is propagated along the sequence of path pellets. This procedure guarantees to find a path if one exists (as shown in Sec. 6), while also respecting the actuation constraints inherent in the crystal under PacMan.

In addition, when a path is confirmed that passes a dangling atom, a second path is planned from the dangler's neighbor to any free edge of the crystal. This is somewhat different in that it is planned in the same direction in which it will be executed, but the planning uses the same depth-first technique. This path will then be executed first to ensure that the dangler does not get disconnected.

5 Distributed actuation

In this section we describe how the pellets are used to create motion in a parallel distributed context. This is the heart of the PacMan algorithm — while the pellets could be determined and placed by a variety of mechanisms, the use of the pellets is fixed to the algorithm given here. The basic idea is that in order to virtually move, an atom will follow the path given by its pellets, exchanging its identity with the neighbor that has its pellet at each step. In addition, the atom will also contract toward the neighbor holding its pellet. The result is an inchworm motion that produces physical motion toward the goal configuration.

Since many atoms may be actuated simultaneously using PacMan actuation, deadlock and fragmentation may occur. To prevent these occurrences, two flags, *busy* and *dibs*, are used to ensure that the atoms complete their PacMan paths. Another constraint appears in the special case of turning a corner in a PacMan path. When turning a corner, a hole is opened up, so a pair of flags are introduced that ensure that the hole does not grow too large and cause fragmentation.

The main part of the PacMan algorithm is given as Algorithm 1, while a message handler that also runs is listed in Algorithm 2 (some trivial handlers are left out for simplicity). The eating of pellets and identity trading happens in the first part of Algorithm 1 (lines 1-19). The atom first asks its neighbor for "dibs" on a trade, then contracts toward it, and finally exchanges identities. The second half of Algorithm 1 (lines 20-28)

Algorithm 1 PacMan algorithm

```
1: if not(busy or dibs or waiting_for_corner) then
2:    if neighbor(dir) has pellet with my ID then
3:       while "OK" message not received do
4:          Send("request transfer") in (dir)
5:       Busy=1
6:       if contracted then
7:          Send("trading ID",ID) in (dir)
8:          Send("trading pellets",pellets) in (prev_dir)
9:          if dir ≠ prev_dir then
10:            corner_neighbor = 1
11:            Send("wait for corner") in (dir)
12:      else
13:         disconnect(perp(dir))
14:         contract(dir)
15:   else
16:      wait for message
17: if at_goal then
18:    Disconnect(perp(dir))
19:    busy = 0
20: if busy and contracted then
21:    if !(Is_nbr(prev_ID)) or at_goal(prev_ID) then
22:       Expand(prev_dir)
23:       Connect(perp(prev_dir))
24:       busy = 0
25:    if corner_neighbor then
26:       while not(connected(corner_dir)) do
27:          try_connect(corner_dir)
28:       Send("Corner connected") in opp(corner_dir)
```

Algorithm 2 PacMan message handler

```
1: Switch (message type):
2: if "Request transfer" then
3:   if not(busy or dibs) then
4:     Dibs = sender's ID
5:     Send("OK") to sender
6:   else
7:     Send("Not OK") to sender
8: if "Trading ID" then
9:   Prev_ID = ID, ID = received ID.
10:  Dibs = ∅
11:  Remove path-pellet with received ID.
12:  Send("Return ID",Prev_ID,prev_dir) to sender.
13:  Prev_dir = dir to sender
14: if "Return ID" then
15:  Prev_ID = ID, ID = received ID
16:  Prev_dir = received prev_dir
```

takes care of a contracted atom that has traded identities with another atom. The contracted atom will wait for the atom it traded with to move on, and then expand and reconnect to its neighbors to complete the PacMan motion. If the contracted atom is next to a corner in the path, it will instead wait for the hole at the corner to be filled before expanding.

6 Analysis

The analysis of the PacMan algorithm falls into two categories. The first portion (Theorems 1-3) defines what class of reconfigurations are possible with these distributed planning and actuation algorithms in the case where only one PacMan path is executed at a time. The second set of results (Theorems 4-6) describes results for sets of paths that execute in parallel (that is, at any arbitrary relative time).

Theorem 1 *The PacMan planning and actuation schemes will allow any peripheral atom relocation around a square of at least five atoms on a side.*

This initial result describes the action of a crystal that is made of a square of atoms (of size ≥ 5) plus a single additional atom, which is to relocate arbitrarily around the square. It can be proven in two parts: first, by concatenating path segments that obey the turn radius constraint, it can be shown that there exists a set of pellets that will cause an atom to move from any point on the edge of the square to any other point on the edge. Then, since the planner operates depth-first without pruning[2], it will necessarily find a path since one exists. Since the planner enforces the turn constraint, the resulting path will be feasible under the actuation primitive.

[2] In order for the search to terminate, the planner prunes loops, where a loop is defined by the path moving through an atom at two distinct points *and* in the same direction both times.

The crystal in Theorem 1 is a special case of a *stem cell*, which is a square of atoms (the *core*) with arbitrary linear projections. The next two results apply to all *well-behaved* stem cells — crystals in which no atom is adjacent to two dangling atoms (as in Fig. 5b).

Theorem 2 *The planner and actuation scheme can generate an arbitrary peripheral relocation in any well-behaved stem cell.*

The difference between Theorem 2 and Theorem 1 is that the presence of additional peripheral atoms has the potential to lead to fragmentation. Therefore, it must be shown that any path can be planned and executed without causing fragmentation. This is shown in three parts: first, a path can be made for any peripheral relocation while avoiding one edge of the core. Second, it is possible for all dangling atoms to move to that edge without causing fragmentation of the crystal. Finally, the path planning done for the danglers will cause them to reach a safe edge and have them wait until it is safe to return to their original locations.

Theorem 3 *The PacMan actuation scheme can generate an arbitrary relocation in any connected group of well-behaved stem cells.*

Generalizing the structure around the reconfiguration requires two more things to be shown. First, an arbitrary relocation in a single stem cell can be composed of a linear relocation to the edge of the core, a peripheral relocation, and a linear relocation away from the core, each of which will be successful. Then, for any crystal that is made of stem cells, it is possible for the cores to line up on a grid themselves through linear relocations. In such a structure, any atom can move into its stem cell, through the set of stem cells, and out to its final location. However, it should be noted that (a) there is no PacMan-style planner to achieve this yet and (b) as specified, it would be quite inefficient.

Theorem 4 *The PacMan actuation primitive will avoid deadlocks in the reconfiguration.*

Deadlock can occur when all atoms that are executing a path are required to wait for another atom that is also executing a PacMan path. Deadlock is avoided by noting that this can only occur when paths form cycles. Then, for an atom to virtually advance along its path, it must get dibs from the next atom. When a set of atoms in a circle are all on paths, it will necessarily be the case that one will ask for dibs first. Since an

atom cannot become busy until it gets dibs (busyness stays with the physical atom, not the virtual one), this request will be satisfied, and so one path in the cycle will be able to proceed. At this point, the cycle will no longer be present, and deadlock is avoided.

Theorem 5 *Any number of non-intersecting paths can be executed at arbitrary relative times if paths through neighboring atoms adjoin for at least five atoms.*

If it was not for the disconnections between atoms due to the actuation method, non-intersecting paths could always be executed without regard for one another. However, atoms along a PacMan path will disconnect from their neighbors, causing a break in the crystal along the path as long as two expanded atoms. Two such paths in adjacent rows or columns could cause a fracture four atoms long, so as long the paths are adjacent for at least five atoms (such as any linear paths through a stem cell with a core of size five), they cannot cause the crystal to fragment.

Theorem 6 *Any number of paths can execute at arbitrary relative times provided that no two path intersections are within three (or fewer) atoms of each other.*

When multiple straight-line paths intersect in a stem cell, there will be always be one path p that intersects with two others such that the intersections along p will be closer than any other two intersections. Like any path, p may cause disconnection of two atoms along its line of action. Then, atoms along each of the other two paths may disconnect in the row above p (and will do so on the side closer to each other). Therefore, if the points of intersection are three atoms apart, it may be possible for the disconnections from p to touch both other paths (whose disconnections may in turn link to those from other paths, eventually fragmenting the crystal). However, if the distance between intersections is four atoms or greater, there will remain connection of the crystal through the line defined by p and so the paths can be successfully executed. For non-straight paths, if all corners are treated as intersections, the same criterion can be used.

For some complex reconfigurations, paths will be required that cannot conform to this constraint. In this case, several stages can be used, each of which satisfies the constraint, giving rise to the idea of a neighborhood in time as well as space. It is simple to detect intersecting paths by the presence of two pellets in a single atom. Any potential nearby paths could then be delayed (moved out of the time neighborhood) or replanned (moved out of the space neighborhood).

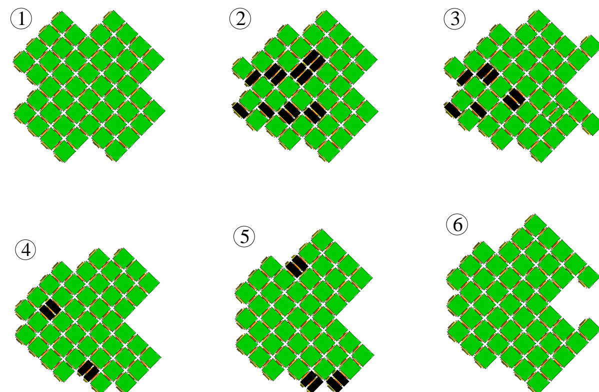

Figure 6: A sequence of screenshots of a PacMan simulation. The black modules are contracted and are currently involved in the actuation process.

7 Experiments

7.1 Simulation

The PacMan primitive has been implemented in a distributed simulation, and can perform reconfigurations such as the one shown in Fig. 6. In the current simulation, a separate process is run for each atom, to obtain reasonable fidelity to the hardware. An additional process is run to simulate physics and facilitate communication between simulated atoms.

7.2 Hardware

In previous experiments with the crystal, the actions of the atoms were synchronized with an external clock. Under the PacMan primitive, it is assumed that the atoms operate asynchronously, and must still be able to physically perform the actuation. For example, there are situations under PacMan where one atom is required to contract while its neighbor expands, such that the two atoms (which are fixed at both ends) always take up the space of 1.5 expanded atoms. Simple experiments were performed with the crystal hardware to determine whether such asynchronous actions are feasible. A row of four atoms were connected to each other and the atoms at both ends fixed to simulate a larger structure, as shown in Fig. 7. Of the two center atoms, one began expanded and the other contracted, and they were commanded to swap states, but not at the same time. Another experiment involved the center row of a 3×3 crystal expanding and contracting asynchronously. These experiments were each successful several times, but did put significant stress on the crystal. The atoms have some compliance which may have hindered (or helped) the experiments, but atoms with a new stiffer design are under construction.

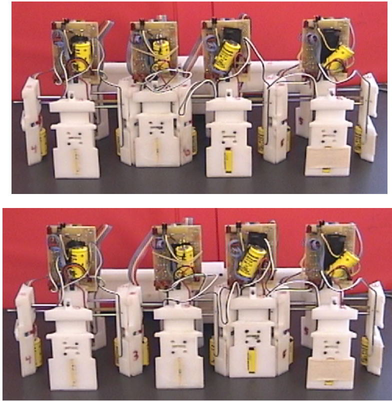

Figure 7: In this experiment, the two center atoms are commanded to swap expansion states while the outer atoms remain fixed.

8 Conclusions / future work

This paper has presented a set of algorithms with which a modular robot system can plan and execute self-reconfigurations in a parallel and distributed fashion. In addition, analysis shows that nearly any self-reconfiguration can be achieved with this technique. We have also presented some guarantees about the parallelization of this technique, but the restrictions are fairly strong, and developing softer requirements is a priority for future work. This seems possible, since the restrictions under which parallel actuation can be proven are not implemented in the current simulations, and yet for most cases (including many in which this constraint is violated) fragmentation does not occur.

In addition, it is possible to use the PacMan algorithm to perform locomotion with a group of atoms, by designating a "front" (where atoms are targets) and "back" (where atoms are mobile). However, the completeness of the algorithm relies on a convex shape which is difficult to maintain during locomotion. (Many non-convex shapes can be successfully created, but the class of such shapes has yet to be determined.) For the same reason, the algorithm does not work well for goal shapes that are extremely convoluted or have internal holes. This also tends to be the case with other distributed reconfiguration planners [9, 11], although perhaps not to the extent present in PacMan.

Finally, we are very interested in extending the PacMan technique to other systems. The most obvious extension is to three-dimensional unit-compressible modules, and in fact this work is well underway. The algorithms transfer almost directly (at least in concept) to three dimensions, with appropriate redefinition of terms. For other systems that use motion over the surface to perform reconfiguration, the extension is not as straightforward. However, we can imagine using similar path planning over the surface, rather than through the interior, with the surface modules obtaining similar pellets. For systems with more complex actuation, such as the molecule [2], the path planning would have a larger branching factor, and the pellets a bit more configuration information, but could be handled much the same way. Some of the actuation constraints of the crystal are not present in other systems, so that the PacMan algorithm may be overly complex, but it may still be a useful method for planning and performing self-reconfiguration.

Acknowledgments

Support for this work was provided through the NSF CAREER award IRI-9624286 and NSF awards IRI-9714332, EIA-9901589, IIS-9818299, and IIS-9912193. Murphy Stein worked on the simulation and developed the reconfiguration shown in Fig. 1.

References

[1] T. Fukuda and Y. Kawakuchi. Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. In *Proc. of IEEE Int'l Conf. on Robotics and Automation*, pages 662–7, 1990.

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

[3] K. Kotay and D. Rus. Algorithms for self-reconfiguring molecule motion planning. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, 2000.

[4] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D self-reconfigurable structure. In *Proc. of the IEEE Int'l Conf. on Robotics and Automation*, pages 432–9, May 1998.

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

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

[7] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Trans. on Robotics and Automation*, 13(4):531–45, 1997.

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

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

[10] M. Yim, D. Duff, and K. Roufas. PolyBot: a modular reconfigurable robot. In *Proc. of IEEE Int'l Conf. on Robotics and Automation*, 2000.

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