6-1-2003

Distributed planning and control for modular robots with unit-compressible modules

Zack Butler
Dartmouth College

Daniela Rus
Dartmouth College

Distributed planning and control for modular robots with unit-compressible modules

Zack Butler and Daniela Rus
Dept. of Computer Science, Dartmouth

Dartmouth CS Technical Report TR2003-462

Abstract

Self-reconfigurable robots are versatile systems consisting of large numbers of independent 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 for systems with two- or three-dimensional unit-compressible modules. We give two versions of the algorithm along with correctness analysis. We also analyze the parallel actuation capability of the algorithm, showing that it will not deadlock and will avoid disconnecting the robot. We have implemented PacMan on the Crystal robot, a hardware system developed in our lab, and we present experiments and discuss the feasibility of large-scale implementation.

I. Introduction

The goal of self-reconfiguring robotics is to create more versatile systems: hundreds of small modules will autonomously organize and reorganize as geometric structures to best fit the shape the object the robot has to manipulate, the terrain on which the robot has to move, or the sensing needs for the given task. Self-reconfiguring robots are well-suited for tasks in hazardous and remote environments, especially when the environmental model and the task specifications are uncertain.

A collection of simple, modular robots endowed with self-reconfiguration capabilities could perform a wide variety of tasks. For example, a modular platform could carry a collection of self-reconfiguring modules to a site. The modules could then grow into a tower, enter the site through a small opening (such as a window) and reconfigure to survey the site. The modules could carry different sensors and collaborate to deploy these within their environment. The robots in such a modular system could also collaborate to perform sophisticated manipulation tasks. Such a system could even use the ability to change shape to act as a reusable 3-D printing device; that is, a physical CAD system that reuses its components to make physical shapes instead of computer models.

To create flexible, robust, and autonomous robotic systems capable of such applications is a considerable challenge, which can be met through (1) hardware designs for reconfigurable systems and (2) algorithmic planning and control that can confer autonomous reconfigurability. The robot hardware and algorithms are intertwined: the hardware dictates the primitive control motions as building blocks for the algorithms and the planners expose the biggest computational needs of the hardware leading to hardware design optimizations. In this paper we focus on planning and control for self-reconfiguring robots. We develop planning and control algorithms and analyze their correctness and parallel performance capabilities. We demonstrate the parallelism of the algorithms in hardware for a small modular robot that consists of 11 modules and project the scalability of these algorithms to larger robots in simulation.

We classify shape-changing for self-reconfiguring systems into two categories: static and dynamic. For a static goal, the system is required to make a particular shape in its current location, for example to generate a table or a sensor array. For dynamic goals, the requirement is for the system to move through space without concern for the exact shape of the group[1]. In this paper we present an algorithm intended for static reconfigurations in which the system is to achieve a particular goal configuration and that configuration overlaps with the current shape. We also show how to extend it to support locomotion via dynamic reconfiguration.

[1]It should be noted that *chain-based systems* such as PolyBot [27] or the CONRO system [5] (and in some circumstances unit-compressible modules [3]) can perform locomotion without reconfiguration, unlike other *lattice-based systems* (see Appendix I for definitions of system types).

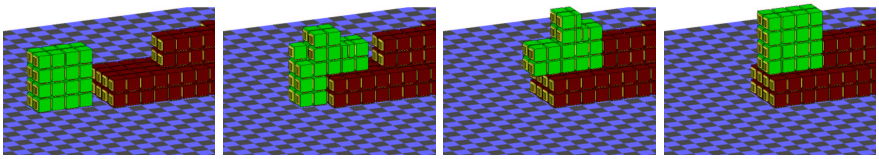

Fig. 1. In this simulation, the light gray modules represent a unit-compressible robot that uses morphing to climb stairs.

We develop algorithms that are matched to a specific type of hardware called *unit-compressible actuation*, where modules are actuated by expansion and contraction. Two specific instantiations of this type of module are the Crystal robot developed in our lab [18] and the TeleCube developed at PARC [20]. Unit-compressible actuation leads to computationally efficient algorithms for shape morphing. In this type of actuation modules can relocate by traveling though the volume of the robot [18] as compared to other existing systems that rely on surface relocation (e.g. [9], [14]). Thus the resulting plans for morphing one shape into another generally require $O(n)$ fewer moves than for surface-moving systems [18].

Our goal is to develop decentralized approaches to path planning and actuation for self-reconfiguring systems that are provably correct, because some of the most interesting applications of this work will employ thousands of modules working together. In a large system, this may mean dozens of modules moving at once, in which case distributed planning and control is important in allowing parallel communication and actuation. Distributed algorithms lead to more efficient and scalable systems by supporting parallelism, taking the burden of low-level motion planning and control off of a central controller. For example, during motion, the movement of a module may depend on the progress of a neighbor's motion. If each motion is controlled from a central location over a common communication link, this may induce significant latency for large systems. Distributed planning and control, in which neighboring modules can communicate and decide locally which motions should take place and when, allows for more scalable and more efficient systems. However, when each module is in control of its own actions, care must be taken to prevent deadlock and ensure that the correct overall shape is achieved.

This paper describes and presents analysis of the PacMan algorithm, a simple scheme for distributed planning and actuation for self-reconfiguring robots with unit-compressible modules, in which actuation takes place through expansion and contraction. The overall idea of PacMan is as follows: to change shape, some modules will be positioned correctly for the new shape and some will not, so the modules not in place will plan paths in parallel to fill in holes in the desired shape. These paths are then actuated by the modules asynchronously. Because of the unit-compressible actuation, a single module does not physically follow an entire path. Rather it will change identities with other modules along the path, such that the module appears to move globally while each physical unit moves only locally.

The natural parallelism of the PacMan algorithm allows for increased performance of the reconfiguration, making it potentially much faster than a central controller for systems with large numbers of modules. The PacMan algorithm consists of two components: a distributed planner that develops paths for individual modules and an actuation protocol that the modules execute in parallel. We characterize the class of self-reconfigurations achievable by PacMan and discuss extensions to the actuation protocol. In particular, we analyze a class of shapes that contains a sufficiently large central group of atoms and simple topology, although the system will work for many other shapes. The analysis of the extended actuation protocol (summarized in Theorem 9) shows that for a large class of simultaneous paths within this class of shapes, the robot does not deadlock and remains connected during actuation without requiring global synchronization.

An outline of the PacMan concept is given in Sec. III, followed by the distributed planning and basic actuation algorithms in Sec. IV and V. Analysis of these algorithms is given in Sec. VI. Extensions that allow for better parallel actuation are described in Sec. VII, along with analysis of these extensions. Software and hardware implementations and discussion of future prospects for these techniques are presented in Sec. IX. A glossary of terms relating to self-reconfiguring robots in general and PacMan in particular is included as Appendix I.

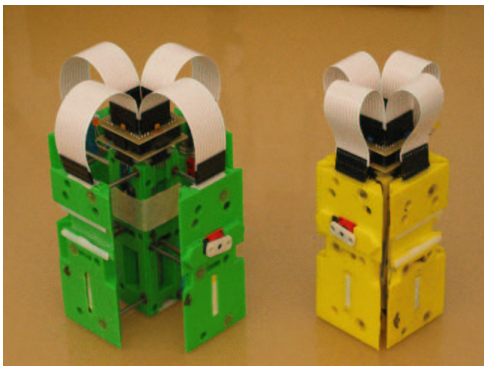

Fig. 2. Two modules of the Crystal robot: the one on the left is fully expanded and the one on the right fully contracted.

A. Related work

Previous work in self-reconfiguring robotics has included hardware development as well as centralized and distributed planning and control algorithms. Several lattice-based self-reconfiguring systems have been proposed using a wide variety of actuation modes (and therefore often requiring different planning algorithms). Systems using unit-compressible actuation include the Crystalline Atomic robot from Dartmouth [18] in 2D and the Telecube from PARC [20] in 3D. Systems which use actuation over the surface of the group include 2D systems such as the Fracta [12] and a small shape-memory alloy based module [30] from AIST and hexagonal deformable modules from Chirikjian's group [7], [16] and 3D systems such as the 3D Fracta from AIST [13] and the Molecule robot from Dartmouth [9]. Chain-based systems have also been implemented, including the Polybot and Polypod by Yim *et al.* [26], [27] and the CONRO system at USC [5]. More general actuation methods were used in the heterogeneous CEBOT system [8], the first proposed self-reconfiguring system; the M-TRAN, which can operate as both lattice-based and chain-based [14] and the bipartite I-Cubes system [22].

Centralized reconfiguration planning has been developed for many systems. These approaches have been applied for complex modules where distributed approaches are more difficult, such as the Molecule [10] or the M-TRAN system [29], to handle topologically challenging shapes [15] or to help achieve globally efficient plans [6], [17].

There has also been significant research into distributed self-reconfiguration planning. Earlier work includes Tomita *et al.*'s [21] method for shape formation based on locally attaining the correct sets of connections between modules. This does not explicitly consider changing between two different shapes. Lee and Sanderson present [11] distributed control for the Tetrabot system. Several researchers have considered reconfiguration for various surface-moving systems. Yim *et al.* described methods for Proteo modules [28] that use simple distributed control rules, but that can sometimes get trapped in local minima. Salemi, Shen and Will developed the idea of digital hormones [19] which are propagated through a modular robot to control shape change and locomotion. Walter, Welch and Amato [25] presented a distributed reconfiguration algorithm (extended by Walter, Tsai and Amato [24]) for planar hexagonal modules. This work is notable for handling both planning (achieved without explicit communication) and actuation control for a large class of goal configurations.

For unit-compressible modules, the original 2-D version of the PacMan algorithm was presented in [1]. This was extended to the 3-D case along with the addition of parallel actuation analysis in [4]. Vassilvitskii *et al.* [23] present a planning scheme for 3-D systems based on meta-modules. In this work, which builds on the original PacMan algorithm [1], groups of eight modules ($2\times2\times2$) are treated as units for planning, and arbitrary reconfiguration between any shapes made of meta-modules can be achieved. Their work focuses on planning, and does not address parallel actuation or synchronization among the modules within each meta-module.

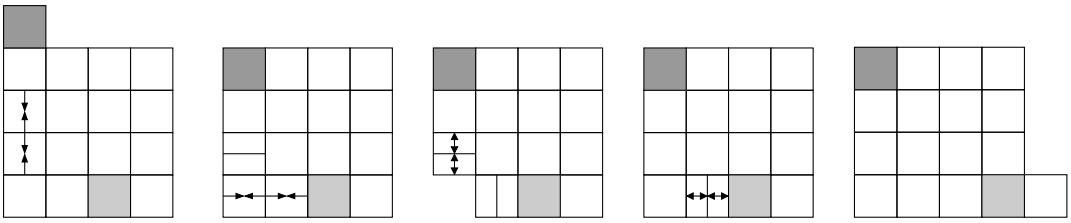

Fig. 3. An example of unit-compressible actuation. Two modules have been shaded to indicate the small motion performed by each physical model despite the large apparent motion of a single module.

II. The Crystal Robot hardware

The distributed algorithms described in this paper are applicable to robots with unit-compressible modules [18], [20]. One specific instantiation is the *Crystalline Atomic* robot, or *Crystal*, developed in our lab [18]. 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. Figure 2 shows two Atoms of the latest version of the Crystal. In the Crystal, each atom is completely autonomous, with computation, power and communication onboard. Each module has two degrees of freedom, so that it can expand its North/South axis independently of its East/West axis. Physical connection between modules uses a lock-and-key style of connector, with each module having two active connectors and two passive connectors. Communication between modules is implemented with an infrared transmitter and receiver on each face of each module, so that the module can only communicate with its immediate neighbors. Any algorithm for the Crystal must therefore be distributed. The unit-compressible module concept extends directly to the three-dimensional case, as do the algorithms presented here. The Telecube system developed at PARC [20] is a 3D unit-compressible system with six degrees of freedom (i.e. each face is independently actuated) which currently uses off-board power and computation.

In any unit-compressible system, a single module cannot move relative to the group on its own. The expansion and contraction of neighboring modules are required to cause it to move with respect to its neighbors, while attaching and detaching from neighbors allows for more complex global reconfigurations. An example of a simple reconfiguration is shown in Fig. 3. An important aspect of controlling unit-compressible robots is to ensure that the overall group does not fragment during actuation due to module disconnections.

Like most self-reconfiguring modular robots, the Crystal is homogeneous, meaning that all atoms are identical. Any module can therefore occupy any position in the goal configuration. Given a goal shape, assigning atoms to specific locations can be done at run-time. This effect enables the PacMan technique, and can lead to very efficient reconfiguration algorithms, as described previously [18].

III. The PacMan concept

PacMan is 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 develops paths for each module that will move. The paths are then actuated by the atoms in a parallel distributed fashion. We wish for the robot to remain connected throughout this process to ensure the integrity of the system and its future operation. A disconnected systems requires complex positioning and sensing to reconnect itself.

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 unit-compressible actuation, since motion of the modules takes place in the interior of the structure. However, a single physical module cannot move through the entire structure to follow its path, as can be seen in the example of Fig. 3. Instead, the 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. 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. 4). The existence of multiple paths in turn allows for several

modules to perform parallel reconfigurations.

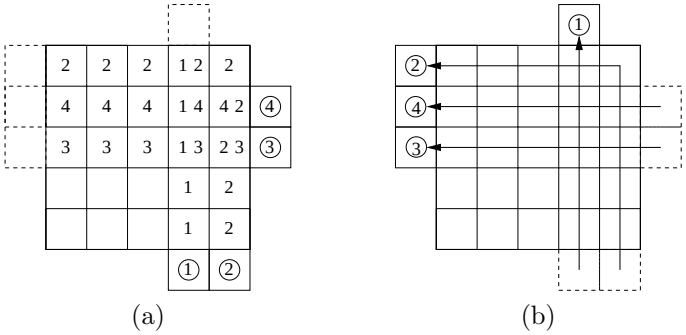

Fig. 4. An example of PacMan pellets: (a) a Crystal with four moving atoms (with circled ID numbers), their pellets shown by number in the other atoms, (b) the paths and resulting structure.

More specifically, the PacMan process is two-fold. First, a path is planned for each module in a distributed fashion, using a planner such as the one described in Sec. IV. The result of planning is a set of pellets distributed through the atoms of the robot. Once the pellets are in place, the actuation will happen asynchronously, directed by the algorithm presented in Sec. V. Each atom looks for pellets and "eats" them without adhering to a strict schedule. Along the way, it exchanges identities with its neighbors, as seen in the simple example in Fig. 5. This means that the intermediate structure of the Crystal will be undetermined, but the final structure will be correct. The distinction between planning and actuation suggests that a centralized planner could be used to create all necessary pellets. This is reasonable for a small system, since the constraints on paths can be complicated and it may be easier to generate paths in a centralized way. For larger systems, a distributed algorithm would likely be preferred. In either case, the modules would perform PacMan actuation in parallel using the protocol described below to allow for efficient simultaneous actuation. We will now describe a distributed planner, followed by the actuation protocol.

IV. Distributed planning

To plan a reconfiguration in a distributed fashion, we have developed a two-stage planning algorithm which develops paths for several modules in parallel, as in Fig. 4b. In the first stage, the modules determine the local difference between the current shape and the desired one, so as to decide which modules can move (i.e. potential starting points) and which locations should be filled (i.e. destinations). The individual paths are then planned in parallel from goal to start using a distributed depth-first search over the modules.

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 define *mobile* atoms as spare atoms that are on the edge of the system and can be moved without disconnecting the structure. These concepts are shown graphically in Fig. 6.

A. Initialization

To determine how the current configuration relates to the goal configuration, we propose a simple matching algorithm. We represent the desired shape with a binary matrix S_d. One atom is chosen as the initiator of the shape matching process, and this atom is given S_d along with its location

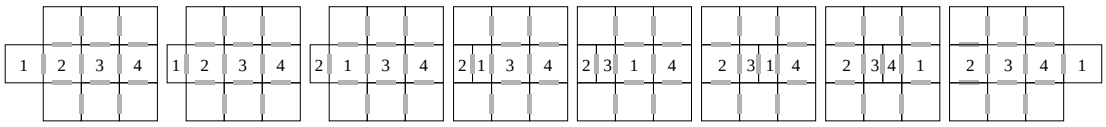

Fig. 5. A simple planar PacMan reconfiguration in which module 1 virtually moves from the left side of the group to the right side. The numbers represent module IDs, not physical modules. Gray bars represent connections between modules.

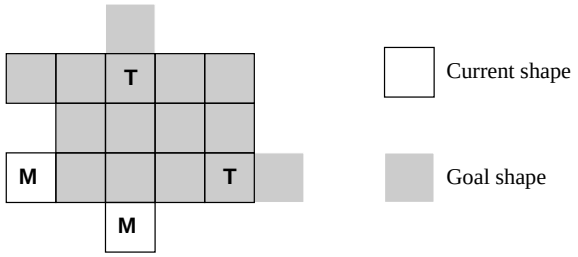

Fig. 6. An example of a robot configuration with a goal configuration. Mobile atoms (which are not needed for the goal) are marked M, while target atoms (those which are next to unoccupied portions of the goal) are marked T.

in S_d[2]. It then marks its location as visited and if its location is not filled in S_d, notes that it is a spare atom. This atom then checks each of its neighbor locations (in S_d and in hardware). For each location, if a neighbor is present and its location is not marked as visited, S_d is passed to that neighbor along with the neighbor's location. If a neighbor is not present but that location in S_d is set to 1, the current atom knows that it is a target atom. This process ensures that all spare and target atoms will be identified, and does not require that the current shape be explicitly calculated. Once an atom has completed its comparisons, it can proceed with the second part of the planning as described below.

B. Path planning

In the second phase of planning, each target atom initiates a search through the robot for a mobile atom to fill its neighboring empty goal location. This search is opportunistic in that a mobile atom will fulfill the first path request that it receives. 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[3] for a mobile atom in a distributed fashion as detailed in Algorithm 1. 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.

Under the PacMan actuation and due to the nature of unit-compressible actuation, an atom cannot follow arbitrary paths through the Crystal, and the planning process must take these restrictions into account. Specifically, there are two constraints that the actuation forces on the planning. First, when a path turns a corner, there is a minimum amount of surrounding structure that must be present. This is due to the disconnections required to turn a corner under unit-compressible actuation, as can be seen in Fig. 3. This constraint can be posed as requiring a number of modules adjacent to the path and a minimum path length to and from the corner. In order to keep track of these data, each path-pellet contains the last direction in which it was propagated (variable d in Algorithm 1), and four counters (T in Algorithm 1) for the number of atoms adjacent to the path in each of the four directions perpendicular to the current segment. The second constraint on planning is that when a path travels along the edge of the Crystal, any *dangling* atoms (see Fig. 7) will become disconnected as the atoms along the path actuate. For purposes of reliability, we require that the system remain connected at all times. The planner therefore forces the dangling atoms to move off of that edge by creating appropriate pellets for the dangler when the original path is instantiated. The original path is then put on hold until the dangler has moved from its original location. This guarantees that the Crystal will remain connected throughout the PacMan actuation.

To plan a path, a target atom will initiate a depth-first search by passing a plan-pellet to its neighbor. This plan-pellet is then propagated by the modules of the system according to rules which generate a correct and complete search procedure including backtracking. To correctly perform the depth-first search, we define a canonical ordering of the four (in the 2D case) or six (in the 3D case) lattice directions. When a path cannot continue straight (which is always preferred due to the physical complexity of turning corners), the path is propagated in the next available canonical direction (lines 8-11 of Algorithm 1). When a path is rejected (for example when it has reached

[2]It should also be possible to give the modules only S_d and have them dynamically determine the best alignment to the current shape.

[3]In practice, inspired by [23], we use an iterative deepening search which minimizes the number of turns (but not the length) of the path.

Algorithm 1 PacMan path planner

```
1:  Pellet p contains:
    d - direction in which it was last passed
    T - set of four counters used to store number of modules adjacent to each side of the current
    path segment
    type - either plan-pellet or path-pellet
    ID - ID of path originator (for plan-pellet) or module that will execute path (for path-pellet)
2:  if Pellet p received then
3:    if p.type = plan-pellet then
4:      if (I am mobile) and (max(p.T) ≥ 2) then
5:        set p.ID to my ID
6:        set p.type to path-pellet
7:        send p to sender
8:      if neighbor exists in p.d then
9:        for each direction perp perpendicular to p.d do
10:         if neighbor exists in perp then
11:           increment p.T_perp
12:         else
13:           reset p.T_perp to zero
14:       propagate pellet in p.d
15:     else
16:       for each direction perp perpendicular to p.d do
17:         if p.T_perp ≥ 2 and there is a neighbor in perp then
18:           set p.d = perp and set p.T = 0
19:           send plan-pellet to the neighbor in perp
20:       send rejection in direction opposite p.d
21:   else {p is a path-pellet}
22:     Find plan-pellet pp with pp.ID = p.ID
23:     Replace pp with p
24:     Send p in direction opposite p.d
25: if Rejection received (from dir rd) then
26:   Find pellet p that matches rejection notice
27:   for each perpendicular dir perp canonically after rd relative to p.d do
28:     if p.T_perp ≥ 2 and there is a neighbor in perp then
29:       set p.d = perp and set p.T = 0
30:       send p to the neighbor in perp
31:       break
32:   return rejection in direction opposite p.d.
```

the corner of the group and cannot turn), the pellet is sent back in the direction it came. In this case, the canonical ordering also allows us to determine the next successor direction using only the information in the pellet and the direction of the rejection message (lines 15-18). Finally, when a plan-pellet is received by a mobile atom, the mobile atom accepts the 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 plan-pellets.

This procedure is guaranteed to find a path if one exists (as shown in Sec. VI), while respecting the actuation constraints inherent in the Crystal under PacMan. In addition, when a path that passes a dangling atom is confirmed, a path is planned for the dangler to reach a safe location. This is somewhat different in that it is planned in the same direction in which it will be executed, but is done in the same depth-first manner. To guarantee that several path searches can simultaneously be completed, atoms will not begin their actuation while searches are in progress. This is ensured through a distributed counter (handled in the same way as the traffic counter described above) that is incremented by each atom starting a search and decremented when the search ends in success or

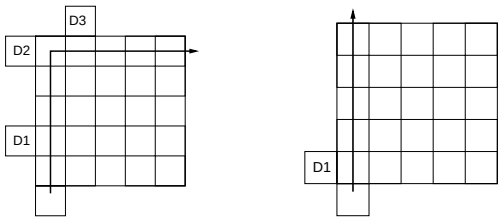

Fig. 7. Examples of *dangling* atoms, atoms that lie by themselves adjacent to the edge of the robot. (a) D1, D2 and D3 can successfully move into the Crystal so as to not become disconnected when the given path actuates. (b) D1 cannot move into the Crystal without disconnecting the atom with the given path, and so cannot be correctly handled under PacMan.

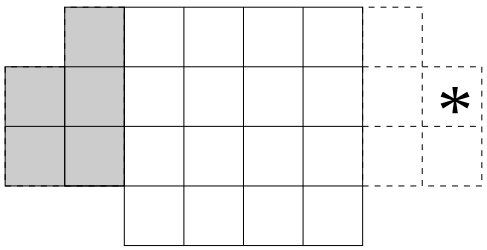

Fig. 8. An example of a reconfiguration requiring multiple waves of planning and actuation: unoccupied goal locations are given by dashed lines and spare atoms are shaded.

failure. No atom will actuate while this counter is greater than zero.

C. Multiple-wave reconfiguration

For some reconfigurations, the start and goal configurations will have large areas which do not overlap, such as in Fig. 8. In these instances, there are locations in the goal shape (such as the starred cell in Fig. 8) that are not next to any atom in the initial configuration. To perform these reconfigurations, the PacMan technique uses multiple *waves* of planning and actuation. In each wave, a layer of atoms is moved from the "back" of the group (areas that are not needed in the goal shape) to the "front." If multiple paths are executed during a single wave, it is necessary for all of them to complete before the next wave is planned so the structure of the group is stable and communication between all pairs of neighbors is possible. To keep the structure synchronized from one wave to the next, we use a distributed traffic counter that represents the number of currently active paths in the group. Each atom keeps a copy of the counter's value, and when one atom needs to increment or decrement the counter, a message is broadcast through the robot to that effect. Message IDs ensure that no increment or decrement message is handled twice, so that all modules will agree on its value. In this particular case, when an atom finishes its path, it decrements the counter and checks its new location in the goal shape to see if it is a target atom. If so, it waits until the counter reaches zero before beginning a path search in the next wave.

ID	Module's current PacMan ID
prev_ID	Module's ID before most recent ID trade
prev_dir	Direction of most recent ID trade
busy	Indicates module is involved in an ID trade
dibs	Set if module has been claimed by a neighbor for ID trade
waiting_for_corner	Set by module near a corner in the PacMan path
corner_nbr	Used with previous; direction to the corner of the path
at_goal	Module has completed its path
nbr_at_goal	Module involved in last ID trade is at its goal

TABLE I

VARIABLES USED IN PACMAN ACTUATION PROTOCOL (ALGORITHM 2) AND MESSAGE HANDLER (ALGORITHM 3).

Algorithm 2 PacMan actuation algorithm

```
1:  if not(busy or dibs or waiting_for_corner) then
2:    Send "Ask-pellet" message to each neighbor
3:    Wait for "Have-pellet" message
4:    Set d to direction of neighbor that has pellet with my ID
5:    if d ≠ ∅ then
6:      while "OK" message not received do
7:        Send("request transfer") in d
8:      Busy = 1
9:      if !contracted(d) then
10:       disconnect sides perpendicular to d
11:       contract(d)
12:     Send("trading ID",ID) in d
13:     Send("trading pellets",pellets) in (prev_dir)
14:     if d ≠ prev_dir then
15:       corner_neighbor = 1
16:       corner_dir = opposite(d)
17:       Send("wait for corner") in (dir)
18:   else
19:     wait for message
20: if at_goal then
21:   Disconnect(perp(d))
22:   busy = 0; at_goal = 0
23: if busy and contracted then
24:   if !(Is_nbr(prev_ID)) or nbr_at_goal then
25:     Expand(prev_dir)
26:     Connect(perp(prev_dir))
27:     busy = 0
28:   if nbr_at_goal then
29:     Send("Path complete" to nbr at goal)
30:     nbr_at_goal = 0
31:   if corner_neighbor then
32:     while not(connected(corner_dir)) do
33:       connect(corner_dir);
34:     corner_nbr = 0
35:     Send("Corner connected") in opp(corner_dir)
```

V. 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 or its extension in Sec. VII. The atom virtually moves along the path given by its pellets. At each step it communicates with the next atom along its path and exchanges its identity with that atom. 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, fragmentation and conflict at path intersections may all occur. To prevent these occurrences, two flags, *busy* and *dibs*, are used to ensure that the atoms complete their PacMan paths. These perform traffic control, so that two paths that need to use the same module do not interfere with each other and cause disconnection of the overall robot. Another constraint appears in the special case of turning a corner in a PacMan path. When turning a corner, a hole is opened up, as can be seen in the lower-left corner of the group in Fig. 3 so a pair of flags are introduced that ensure that the hole does not grow large enough to cause fragmentation.

The main part of the PacMan algorithm is given as Algorithm 2, while its supporting message

Algorithm 3 PacMan message handler

```
1:  Switch (message type):
2:  if "Ask-pellet" then
3:    if I have path-pellet p with p.id = received ID then
4:      Send"Have pellet" to sender
5:  if "Request transfer" then
6:    if  not(busy or dibs) then
7:      Dibs = sender's ID
8:      Send("OK") to sender
9:    else
10:     Send("Not OK") to sender
11: if "Trading ID" then
12:   Prev_ID = ID
13:   ID = received ID.
14:   Dibs = ∅
15:   if path-pellet with received ID is last in path then
16:     at_goal = 1
17:     set new position in goal matrix
18:   Remove path-pellet with received ID.
19:   Send("Return ID",Prev_ID,prev_dir,at_goal) to sender.
20:   Prev_dir = dir to sender
21: if "Return ID" then
22:   Prev_ID = ID, ID = received ID
23:   Prev_dir = received prev_dir
24:   nbr_at_goal = received at_goal
25: if "Path complete" then
26:   Decrement path counter
27:   if all paths in wave complete then
28:     Start planner with new position
29: if "Wait for corner" then
30:   waiting_for_corner = 1
31: if "Corner connected" then
32:   waiting_for_corner = 0
```

handler is listed in Algorithm 3. The atom first asks its neighbor for "dibs" on a trade, then contracts toward it, and finally exchanges identities, as detailed in the first part of Algorithm 2 (lines 1-19). The busy flag is also set to indicate that it is undergoing an identity transfer. Once an atom has traded, that physical atom is no longer actuating a path, but it is contracted and must still assist in the PacMan actuation. 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. The second half of Algorithm 2 (lines 20-35) details these processes.

VI. Analysis

In this section we analyze the correctness of the basic PacMan algorithm and define a class of reconfigurations that are possible in the case where only one PacMan path is executed at a time. In Section VII we extend this analysis and describe a class of paths that can be executed in parallel (that is, at arbitrary relative times). To build our arguments, we define a class of configurations called *stem cells*, and restrict our analysis to this class, although reconfigurations of other shapes are possible. A stem cell is a group of modules consisting of a *core*, which is a square or cube of a given size, along with arbitrary projections from the edges/faces of the core. The robots shown in Fig. 4 and Fig. 7 are all 2-D stem cell robots. We first show that the PacMan actuation is sufficient for both 2-D and 3-D systems despite the constraints described in Sec. IV. The remaining analysis proves that these paths can be planned and executed over a class of robot shapes. These latter

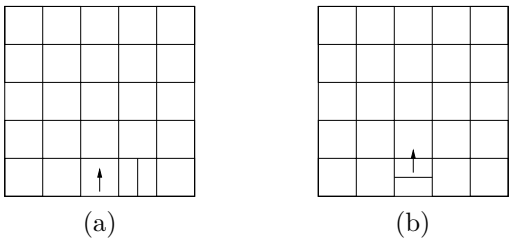

Fig. 9. Two ways in which a centered atom can exist.

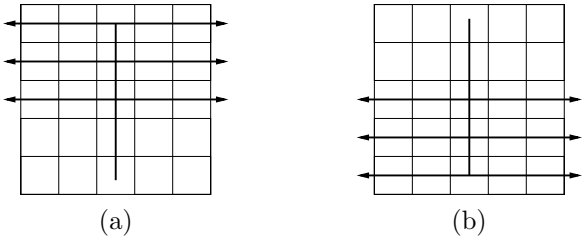

Fig. 10. Peripheral location reachable from (a) the southern centered atom and (b) the northern centered atom.

results apply equally to the 2-D and 3-D cases.

Theorem 1: [2D Actuation] For any two-dimensional stem cell of size 5 consisting of a core and a single extra atom, a PacMan path exists for the extra atom to move to any other location on the periphery.

In both 2-D and 3-D, we show that PacMan actuation is sufficient for stem cell reconfiguration by defining a set a canonical path segments, and showing that a path for any peripheral atom relocation can be assembled from the canonical segments.

Proof: To develop these canonical paths in two dimensions, we first define a *centered* atom as one in the center location of a side of the core grain and able to move toward the center of the core. In the context of the particular situation here, a centered atom will take one of the two forms shown in Fig. 9: either having just turned the corner and so having a contracted pair beside it, or having just entered the core. We then prove this theorem in two steps: (1) showing that any atom on the periphery can become a centered atom and (2) that any centered atom can reach any other location on the periphery.

(1) It can be shown that any peripheral atom can "become" (in the virtual sense) a centered atom. For atoms adjacent to the center atom of an edge, they simply compress and swap identities to become centered. Atoms adjacent to the corner of the core can "travel" along the edge and turn at the middle of the edge. Finally, atoms between the center and corner must travel in a P shape as shown in Fig. 11. Next it can be shown that any centered atom can become a centered atom on any other side, through the use of the paths shown in Fig. 12, so all centered locations can be reached from any initial location.

(2) We then note that a centered atom on the south edge of the core can travel (using the PacMan primitive) to locations attached to the east and west edges in the north half of the core without fragmenting the Crystal. This holds similarly for a centered atom on the other three edges. Adding these four together, and noting that any centered atom can move to any other, mean that any

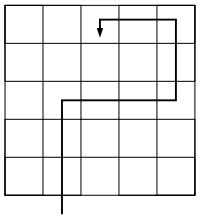

Fig. 11. Path used to get from an "off-center" position to a centered atom.

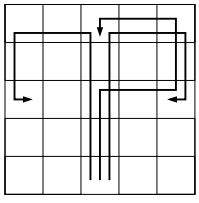

Fig. 12. Relocation of a centered atom to other centered locations.

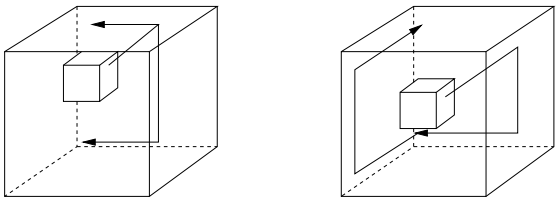

Fig. 13. From a given start location for a PacMan path adjacent to the -Y (close) face of the group, a pair of corners can be reached as shown for (a) a start location in the top or bottom layer and (b) a start location in the middle layers. Other start positions can be handled by generalizing the given paths.

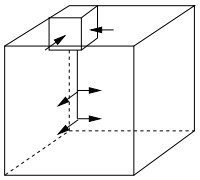

Fig. 14. A PacMan path that enters a top (or bottom) corner can continue in Z and depart in different layers. If a path can reach both the top and bottom corner, all layers can subsequently be reached.

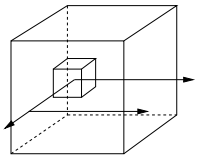

Fig. 15. A PacMan path coming from a corner of a given XY plane of a 3D stem cell can reach any peripheral location in that plane (only some simpler paths are shown for clarity).

centered atom will be able to reach all peripheral locations. Since any atom on the periphery can become centered, any single atom relocation around the core is possible. ■

Theorem 2: [3D Actuation] For any three-dimensional stem cell of size 4 consisting of a core and a single extra atom, a PacMan path exists for the extra atom to move to any other location on the periphery.

Proof: We wish to show the existence of a path for a module located at (x_0, y_0, z_0) to relocate to any position (x_g, y_g, z_g) on the surface of a $4 \times 4 \times 4$ stem cell. The stem cell is a cube lying between $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$. Intuitively we will show that a PacMan path exists that can be used to move the module to some (x, y, z_g) first and then move the module in that XY plane to the desired (x_g, y_g) in a similar fashion to Theorem 1. We show this by constructing a path from canonical segments. Without loss of generality, we can assume that neither the start nor goal location are on the +Z or -Z faces of the cube and that the start location is adjacent to the -Y face of the stem cell (i.e. $y_0 = y_{min} - 1$).

For the first part of the path construction, we first need to create a path from the starting position to each of two corners of the stem cell (the need for both paths is explained below). Specifically, we create paths from (x_0, y_0, z_0) to (x_1, y_1, z_{min}) and (x_1, y_1, z_{max}), where (x_1, y_1) is the farthest corner (in XY) from (x_0, y_0). This construction can be shown for two cases: (1) for start locations in the top and bottom layers ($z_0 = z_{min}$ or $z_0 = z_{max}$) and (2) locations in the middle two layers. In the first case, the corner in the same XY plane as z_0 can be reached with a path that goes in +Y and then ±X, while the matching corner in the far layer can be reached by a path in +Y, ±Z, ±X. Examples of these paths are shown in Fig. 13a. For the second case, the far corner in the farther layer (two layers away) can be reached as in the first case, but the corresponding corner in the layer adjacent to the start layer requires a path in +Y, ∓Z, ±X, -Y, ±Z, +Y, as shown in Fig. 13b.

Once the path has reached a corner of the stem cell, it should turn to reach the plane containing the goal ($z = z_g$), and turn into that plane to reach its final destination. As long as both corners at (x_1, y_1) can be reached, the path can turn in Z to reach (x_1, y_1, z_g) as shown in Fig. 14. Finally, it then turns into that plane and can reach any (x_g, y_g) as described in Fig. 15. ■

We note that for both 2-D and 3-D, any stem cell of a greater size than specified in the theorem can be analyzed in a similar way to show that the extra atom can be arbitrarily relocated around the periphery. Finally, while a *valid* (i.e. obeying the actuation constraints) path can always be created by the concatenation techniques presented, in many cases a shorter path can be found.

Theorem 3: [Planning] For any relocation as described above, the distributed planner will find a

valid PacMan path and leave appropriate pellets for the atom to use.

Proof: Theorems 1 and 2 shows that a single continuous valid path exists for any relocation. Since the search for a moving atom is a complete DFS of the cells, a path from start to goal will be found. In addition, the search respects the PacMan turning constraint by using the set of counters T, so that any path found will be valid. The search process places pellets (incrementally) with the ID of the start atom when a path is found. Thus, the pellets created can be followed to perform the relocation. ■

We now continue our analysis by showing what happens when the stem cell robot has additional units we call dangling atoms. We would like to ensure that the dangling atoms do not get disconnected during PacMan actuation. In many cases (such as D1 in Fig. 7a), these atoms can simply translate across the core before the path is traveled and back to their original location once the path is completed. If a dangling atom is next to a corner of the core, its path might travel next to other dangling atoms (see D2 and D3 in Fig. 7a). This is possible as long as the two atoms do not share a corner such as in Fig. 7b. We define any stem cell that does not contain such pairs of dangling atoms as *well-behaved*. The following theorems capture some observations about well-behaved stem cells.

Theorem 4: [Dangler handling] Any relocation as described in Theorem 2 can be performed in any well-behaved stem cell.

Proof: For any planar reconfiguration, there will always be at least one edge that is not adjacent to the path, and therefore "safe" for dangling atoms. The existence of a free edge can be shown on a case-by-case basis. For any non-planar reconfiguration, the presence of a cubic core gives even more safe locations. By the analysis of Theorem 2, at most eight linear path segments are required for any path. Even if all these segments are on the surface of the core, at most 25 modules can be involved in the path for a core of size four. There are 52 total surface modules with adjacent free space (i.e. not counting those on the floor), so there will always be sufficient places for dangling atoms. Since a path can be found from any peripheral location to any other, the planner will always be able to generate a path for any dangler. ■

Thus, any path through a single stem cell can always be performed successfully. We can now extend this analysis to a collection of well-behaved stem cells.

Theorem 5: Any arbitrary reconfiguration of a well-behaved stem cell can be performed with the PacMan primitive and appropriate pellets.

Proof: This analysis applies independently to actuation and actuation plus planning. In either case, an arbitrary reconfiguration can be decomposed into a series of atom relocations (only for analysis, the planner will produce a single plan). For each relocation, the atom can move linearly to the edge of the stem cell, through the stem cell and linearly out to its final location. ■

Theorem 6: Any arbitrary reconfiguration of a set of connected convex stem cells can be accomplished with the PacMan primitive and appropriate pellets.

Proof: It is possible for any connected set of convex stem cells to move such that they form a configuration that includes a tree of a given width, rather than just a square, with all additional atoms linearly contiguous with an atom in the tree. To build this construction, one stem cell acts as the core for the composite stem cell and the other cores will move to form a connected tree. The extra atoms from each core grain will either stay in their original position relative to their core grain or will have to move to the opposite side (if the edge they are on will connect to another core). From this configuration, any relocation can be performed — an atom can move to the edge of its core grain, through the set of core grains, and out at any other location. Once the atoms have relocated, the original stem cell structure (or any other connected stem cell structure) can be recreated as desired. ■

At this point we can also show that the basic PacMan actuation protocol will not result in deadlock of the system — that is, for any set of paths, at least one atom will be able to move at all times. Note that this does not guarantee that the robot will not fragment, which is addressed by the additions in the following section.

Theorem 7: The PacMan actuation primitive will avoid deadlocks in the reconfiguration.

Proof: 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, such as in Fig. 16. Then, for an atom to virtually advance along

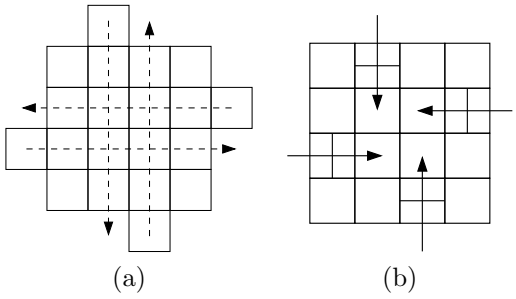

Fig. 16. (a) A set of paths containing a cycle, (b) A likely intermediate configuration for this reconfiguration, at which point each path is waiting for a module that is itself on a path.

its path, it must get dibs from the next atom. When a set of atoms in a circle are all on paths and reach a configuration such as Fig. 16b where each one has reached another path, it will necessarily be the case that one atom will ask for dibs first. Note that an atom cannot become busy until it gets dibs since the busy flag is fixed to the physical atom, not the virtual one. Therefore, the first dibs request within the cycle will necessarily 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. ■

One additional consideration for 3-D reconfigurations is the presence of gravity. The nature of the unit-compressible actuation makes it possible to ensure stability for most shapes. In particular, the motion of a module during reconfiguration never leaves the boundaries of the initial shape. Assuming the initial and final shapes are stable, this is generally sufficient to ensure that the reconfiguring shape is stable, since the moving module leaves the edge of the group, moves through the center, and out to the perimeter. For some paths, it is possible for a module to go across the center of mass only during the middle of the path, in which case the initial and final shapes cannot be marginally stable — they must be stable with the addition of one module mass anywhere in the system.

VII. Enabling Parallelism

The basic PacMan algorithm, while compact and straightforward, has some limitations in the ability to support many intersecting paths without undergoing disconnection of the Crystal, both in 2-D and in 3-D. Theorem 7 shows that the basic PacMan algorithm will not produce deadlocks during parallel actuation, but at the expense of potential disconnections. That is, the class of paths that can be guaranteed to execute in parallel without disconnection of the group is extremely limited. In this section, we introduce two additions to the basic PacMan algorithm which can greatly increase the class of simultaneous paths that can be assured to complete successfully. We first present and analyze an extension under which any set of paths that does not include a cycle can be executed at arbitrary relative times without fragmenting the system. A set of paths is said to contain a cycle if the directed graph formed by the paths (with vertices at each atom) contains a cycle (Fig. 16 shows such a set of paths). The second extension guarantees that PacMan will still never deadlock while avoiding fragmentation whenever possible.

A. Using State to Avoid Fragmentation

The first addition to the PacMan algorithm addresses how to limit the disconnections adjacent to path intersections in the Crystal. Under the original algorithm, when a path approaches a module that a different path is already using, the atom executing the second path will not disconnect. Thus, the disconnections in the Crystal will remain separate (see Fig. 17a). However, once the first path is through the intersection, the second can use the intersection immediately, as in Fig. 17b, and cause the disconnections to link up, potentially fragmenting the robot. If the first path then turns as in Fig. 17c, its atoms can be disconnected regardless of the size or dimension of the Crystal.

To fix this problem, we introduce a boolean state variable and an additional message type. This will force the disconnections around the intersection to be symmetric and not link up. The added state is a boolean *nbr_wait* that signifies whether an atom has recently been used by a path and has a contracted neighbor. This is set to 1 when a pellet is received with an ID return message. Such a message signifies an intersection, as this additional pellet belongs to a path yet to use this atom. The *nbr_wait* flag will remain set until a "Neighbor OK" message comes from the atom that

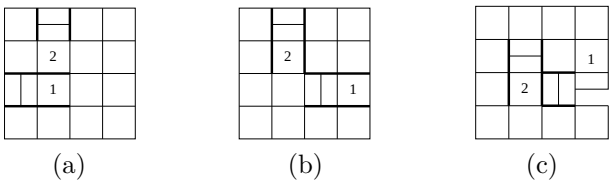

Fig. 17. Two intersecting paths that lead to fragmentation under basic PacMan. Thick lines indicate disconnected atoms.

it swapped with, which will happen as detailed in Algorithm 4.

Algorithm 4 Extension to PacMan

```
14: if busy and contracted then
15:   if !(Is_nbr(prev_ID)) or at_goal(prev_ID) then
16:     Expand(prev_dir)
17:     Connect(perp(prev_dir))
18:     busy = 0
19:     Send("Neighbor OK") in opp(prev_dir)
```

As long as the *nbr_wait* variable is set, the atom will refuse to give dibs to the atom executing the second path. With this additional restriction, the intersection will evolve over time as shown in Fig. 18.

A.1 Analysis

The additional state and message described above enables correct planning and actuation for any set of paths not containing a cycle. To prove this, we look at the immediate neighborhood of each path intersection and show that the atom(s) that occupy the location of the intersection will remain attached to all other atoms in the Crystal throughout the actuation of the two paths. We can then induce from this result that no set of intersections can disconnect the Crystal given a restriction on the trading of IDs.

We show the correctness of a single intersection by analyzing the connectedness of various components of the Crystal during actuation. We use the term *mover* to refer to a virtual atom that is executing a path, that is, an atom with an ID for which pellets exist in the Crystal.

Lemma 1: Any Crystal with two intersecting paths will remain connected during actuation when using PacMan with the extension in Algorithm 4.

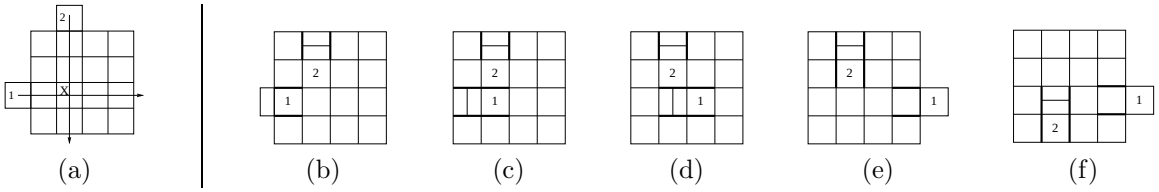

Fig. 18. (a) Two intersecting paths and (b-f) one of two actuation sequences for these paths under Algorithm 2 together with Algorithm 4.

Proof: Consider two intersecting paths as shown in Figure 18a. The only possible point of fragmentation during the parallel execution of the two paths is around the intersection point marked in Figure 18a by X. Suppose path 1 obtains dibs from X first, we will have the situation shown in Fig. 18. At each of these positions, note that the structure remains globally connected despite local disconnections. Also, between each step, connection is also maintained. If path 2 obtained dibs first a similar process would take place, also retaining global connectivity.

Finally, if one of these paths turns near the intersection, the actuation still succeeds. If the corner is before the intersection, then the path would disconnect near the intersection only when it got dibs on the intersection, and its disconnections due to the corner would not touch the intersection otherwise. If the corner is after the intersection, then the "Neighbor OK" message would not be sent until after the corner had been filled and the nearby atoms expanded, and so the intersection would behave the same way. ■

Algorithm 5 Augmented PacMan message handler

```
1:  Switch (message type):
2:  if "Request transfer" then
3:    if not(busy or dibs or nbr_wait) and
      (not mover or (waiting_for = sender)) then
4:      Dibs = sender's ID
5:      Send("OK") to sender
6:    else if mover then
7:      if waiting_for < ID then
8:        Send("Wait for", ID) to sender
9:      else
10:       Send("Wait for", waiting_for) to sender
11:   else
12:     Send("Not OK") to sender
13: if "Wait for" then
14:   waiting_for = received ID.
15: if "Trading ID" then
16:   Function from Algorithm 3
17:   Mover = 1
18: if "Return ID" then
19:   Function from Algorithm 3
20:   Mover = received mover
```

Theorem 8: Any set of paths will be executed by the atoms without fragmenting the Crystal provided a mover is never displaced.

Proof: For ease of exposition, we first examine the case when all the paths are straight by an inductive case analysis. Lemma 1 guarantees no fragmentation for two intersecting paths. Consider the effect of a third intersecting path. The same process will take place around the added intersection point. Note that all disconnections in the immediate neighborhood of a single intersection are only in one direction at any given time. The disconnections due to the two intersections can only adjoin if they belong to the common path or if the two intersections are adjacent and the active paths parallel (in which case the minimum segment length restrictions ensure connectivity). This inductive argument applies for each additional path and intersection in the absence of cycles. As long as the mover does not trade identities with a neighbor, it will wait to use the intersection until it can do so safely.

For paths with corners, there will be disconnections that are perpendicular and touching adjacent to the corner. However, as long as each path segment is of the required length, these disconnections alone cannot fragment the Crystal. Any additional path intersecting the turning path will be subject to the restriction above, so the disconnection due to the corner cannot extend beyond the neighborhood of the corner and cause fragmentation. ■

B. Using State to Handle Cycles and Deadlocks

The restriction in the previous theorem points out an inherent compromise in the original PacMan algorithm. In the original algorithm, deadlock is avoided by allowing one mover to displace another to break a cycle of waiting paths. However, this can cause a group of atoms to become disconnected from the main group. A simple alternative is to disallow cycles completely (or equivalently, to allow deadlock) by not allowing a mover to give dibs[4]. This will guarantee that the Crystal stays connected by Theorem 8, but at the expense of potential deadlock. The extension presented here allows both cycles and guaranteed connection in the absence of cycles, but at the expense of additional messages and atom state.

The additional information and computation serves two purposes: (1) to allow each atom to know explicitly whether it is a mover or not, and (2) to detect cycles in the active paths. With this

[4]This can be done without any additional state by simply changing the times when the "busy" flag is set and cleared.

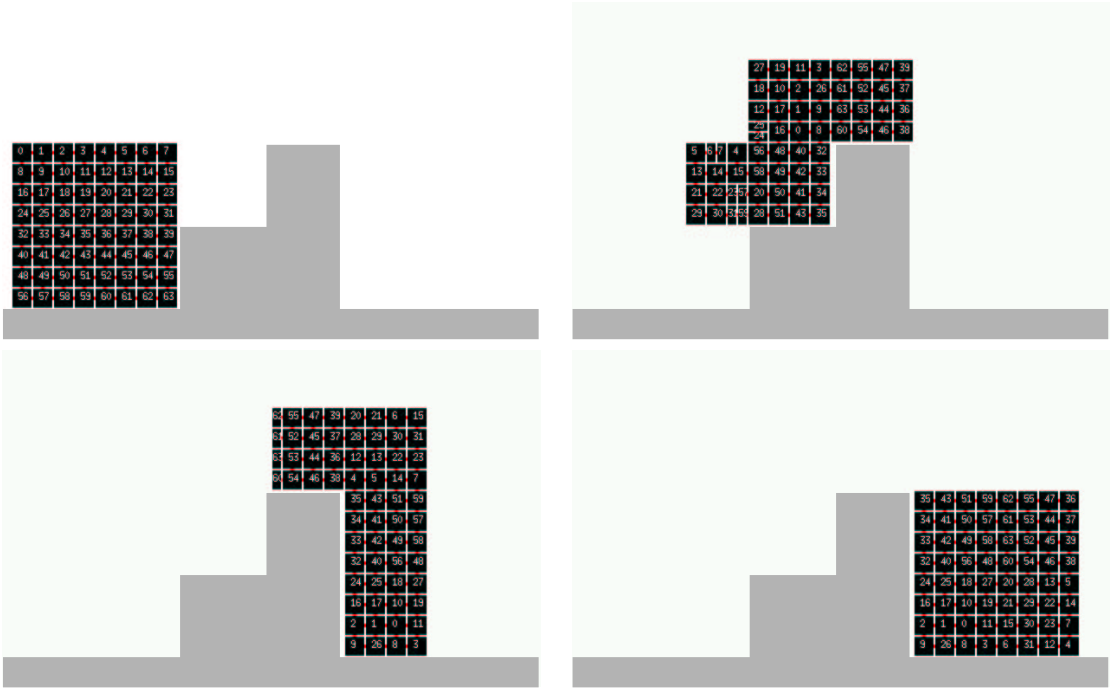

Fig. 19. Simulation of dynamic reconfiguration using PacMan. Each module is labeled with its PacMan ID.

information, we allow the default behavior to be for a mover to not give dibs to another atom, to avoid fragmentation as detailed in Theorem 8. However, to avoid deadlock, each mover that is denied dibs will determine the identity of the mover at the head of the chain. If that mover is itself, this clearly indicates a cycle, and so the atom can override the rejection and get dibs anyway (as it would be able to immediately in the basic algorithm). The changes are only in the message handler, which is augmented as shown in Algorithm 5.

B.1 Analysis

Theorem 9: The augmented PacMan algorithm will prevent fragmentation in the absence of cycles, and deadlock in all cases.

Proof: The augmented algorithm centers on the detection of cycles in the current paths. When a cycle occurs, each atom will be denied dibs by the one ahead of it, as all will be movers and none will be able to move. Since the atoms ask repeatedly for dibs, even if the cycle is entered at multiple points, eventually all of them will be waiting for the atom with the largest ID. This atom will then assert priority and trade IDs with its neighbor, alleviating deadlock. On the other hand, if a cycle is not present, there will always be one atom capable of advancing along its path. Any atom that must wait for that mover (or wait for one waiting for that mover, etc) will not be able to wait for itself and therefore assert priority. This will enforce serialization of paths around intersections and prevent movers from trading IDs, thus preventing fragmentation. ■

VIII. Dynamic reconfiguration

Although PacMan is designed for static reconfigurations, it can also be applied to create dynamic structures, where the goal is to move the robot from one place to another, perhaps over uneven ground. The exact intermediate shapes are generally not of concern. We would like to take advantage of the distributed actuation protocol within PacMan to efficiently perform locomotion.

For simple shapes without obstacles, this can be done very simply by designating a “front” and “back” to the group, and running a series of waves in which the modules at the front initiate plans and the modules at the back consider themselves mobile. In this case, the overall shape of the group will not change during locomotion.

If we want the group to move in the presence of large obstacles, the key is to ensure that the intermediate shapes are amenable to the PacMan algorithm. In addition, since the initial location

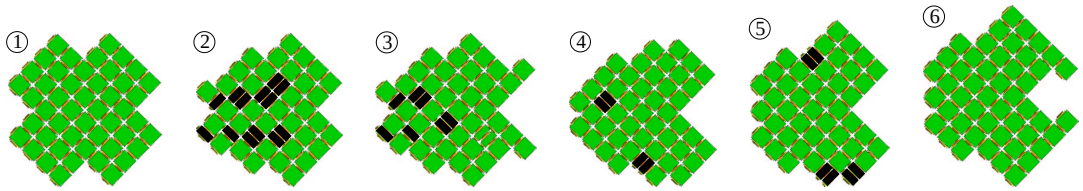

Fig. 20. A sequence of screenshots of a PacMan simulation. The black modules are contracted and are currently involved in the actuation process.

and final location will generally not overlap, we must run PacMan several times, since the planner assumes overlap between the current shape and the goal. To do this, we have currently implemented a simple centralized high-level planner. This planner keeps blocks of modules together so that the overall group is viable for PacMan paths, and submits new goals to the system every four waves (after one block has virtually moved from the back of the group to the front). Assuming that the modules can sense the presence of nearby surfaces, we can use that information in the path planner to treat any surface topology as an obstacle, thus our algorithm supports locomotion on uneven ground. Snapshots of a simulation based on this technique are shown in Fig. 19. An interesting issue that arises in planning for dynamic structures is the stability of the robot during motion. Our dynamic high-level planner can be augmented with a check to ensure that the center of mass of the robot always projects to a point of support on the ground.

IX. Implementation / Experiments

A. Simulation

The PacMan primitive has been implemented in simulation, and can perform 2-D reconfigurations such as the one shown in Fig. 20 as well as 3-D reconfigurations. In this 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. The physical model is discrete, since the atoms are in a lattice.

Any time an atom needs to send a message to a neighbor or perform an actuation, it puts a message in a FIFO that is read by the physics process. For communications, the atom builds a packet with the message type and any appropriate data (e.g. its ID for a "Trading ID" message). The packets need not be a constant size for all messages, as long as the atoms have a common protocol — each atom can look at the incoming message type, and expect the appropriate amount of data to follow. This packet is sent to the physics process, which determines which atom is adjacent to the sending atom, if any, and relays the message, along with the direction from which it came. Each atom process has a separate FIFO associated with it with which it can receive messages from the physics.

For actuation requests, the atom simply builds a packet that tells the physics what type of actuation it is attempting (expansion, contraction, connection or disconnection) and in what direction, and the physics must determine if the motion is feasible. For connection and disconnection requests, the physics can easily determine success or failure (disconnections always succeed), updates its model of the robot, and returns the value to the atom. For expansion and contraction, it must be determined whether the motion would be constrained by the other atoms in the lattice. This is done with a breadth-first search of the connections of the robot, starting from one connection of the moving atom. If the search reaches the opposite side of the moving atom, a loop is detected, and the actuation would not be feasible. In addition, if an expansion would cause internal collisions between modules regardless of their connections, it is denied by the physics. If the requested action succeeds, the model is updated and the atom informed through its FIFO that it was successful.

If an action is not immediately feasible, the assumption is that the atom will continue attempting actuation, so the physics adds the actuation to a list of pending actions. After each new attempted action by any atom, the physics process examines the list of pending actions and sees if any of them can now be taken. In addition, the physical model attempts to match pairs of expansions and contractions within a row or column. These matched pairs of actuations naturally occur during PacMan, and so it was necessary to enable them. When two actions are matched, the two requesting atoms are notified.

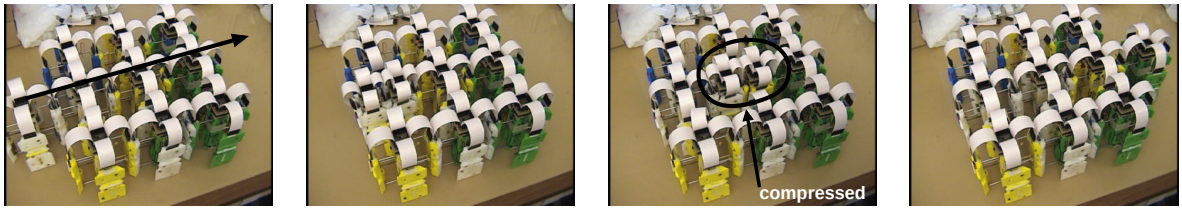

Fig. 21. Crystal hardware performing a PacMan experiment. The reconfiguration being executed is the same as the one shown schematically in Fig. 5.

The atoms themselves run the PacMan planning, actuation, and message handling algorithms as outlined in this paper. The PacMan algorithm presented in Algorithm 2 is run as the main loop, with the message handler called after each action or cycle of the loop.

B. Hardware

In order to verify the performance of PacMan on a physical robot system, we have implemented the planning and basic actuation protocols on the Crystal robot. In the second generation of the Crystal (seen in Fig. 21), of which 18 modules have been constructed, each module is completely independent, having computation, communication and power all onboard. In this system, all communication and actuation is asynchronous, and since all communication is between neighboring modules, there is no inherent capability for broadcast messages. Each atom of the Crystal has four UARTs[5], one dedicated to each of its four faces, and so can accept messages from each of its neighbors with the direction of the incoming message known. Detailed descriptions of the hardware and low-level communication protocol (all developed in our lab) are presented in [2].

One aspect of the initial communication library developed for the Crystal is that although message passing is implemented at the low level, the message size is currently limited to two bytes, including the message type. This means that some of the message formats had to be trimmed and packed to fit into this framework. For all messages, four bits were used for the message type, which was sufficient to allow both planning and actuation messages. The remaining twelve bits were used for data, with a common framework for the data where possible. For example, in the planning messages, we use the bottom four bits for the ID of the pellet in both the plan-pellet propagation and rejection messages. The path-pellet message uses four bits for the ID of the plan-pellet (the plan originator) and four bits for the ID of the mobile atom that has accepted the path request. For almost all message types, we were able to fit all the required data into the twelve bits available provided that only four bits are used for a module ID. This means that the current protocol can handle only 16 atoms, which is not a severe limitation for the current system, and can be easily changed with a more complete communication protocol. The one message that could not be limited to twelve data bits was the "giving pellets" message, since an arbitrary number of pellets may need to be sent. Instead, we send a new message for each pellet, with a "last pellet" bit set in the last message.

Previously, we have implemented several distributed algorithms for the Crystal using message loops as the basic architecture. In this framework, the processor polls the four UARTs in turn, adding any received messages to a queue, then processes the messages from the queue one at a time using the appropriate message handler. We were able to use the same infrastructure for PacMan. Message handlers were written as described in Sec. V. The PacMan actuation protocol was then written as a separate function which is called by the message handlers after relevant messages.

Our implementation was run on an 11-module robot, and was able to both plan and execute various single paths as well as intersecting straight-line paths, showing that the algorithm (including traffic control) can perform correctly in a physical distributed asynchronous system. Snapshots of one experiment are given in Fig. 21. The algorithms worked correctly, although some slight mechanical assistance was required to allow the modules to reach their intended goal configurations. These failures are solely due to compliance in the module faces and connectors, and we are currently working to address this hardware problem in a new version of the system, as described in [2].

[5]A UART is a device which performs standard serial communication, both transmitting and receiving.

C. Lessons Learned

Our physical experiments with the PacMan algorithm have taught us several lessons about the feasibility of large scale distributed control in such systems. The implementation has demonstrated the feasibility of the PacMan distributed algorithm and our simulation projections have demonstrated its scalability and efficiency through parallelism. However, hardware remains the bottleneck in realizing large scale applications with self-reconfiguring systems.

The main source for error in our experiments has been the connection mechanism. During expansions and contractions, modules do not advance along a straight line and often accumulate some error in their trajectory, as the actuators and associated linear bushings are not stiff enough to ensure proper motion. Because the module faces are stiff the mobile unit's connector can get jammed in the face of its neighbor. The trajectory error also affects the alignment between the connection key of a module and the neighbor's slot for the connection. The current key design mechanism has very little tolerance to this kind of error. In response to these observations, we have designed a different connection mechanism that has been inspired by the gripper-like mechanism used by our molecule robot [2]. Because making and breaking connections is the most frequent operation for actuating self-reconfiguration plans, it is important to develop new approaches to connections that are strong, small, simple, efficient, and tolerate misalignment.

Other important design issues that were brought to light by our experiments include the communication required between modules, actuator performance, and electrical power supply. We have observed that the simple communication infrastructure of the Crystal robot is sufficient for complex coordination of actuation, although more capable protocols involving handshaking and larger messages will make future implementations more straightforward. The number of messages required for PacMan is fairly large, although we have not made an explicit effort to optimize our implementation. It should also be noted that all messages are neighbor-to-neighbor, so that many messages can be sent in parallel. For the simple example shown in Fig. 21, a simulation of this reconfiguration required 124 actuation-related messages for the 18 total actuations (expansions, contractions and connections). A simulation of the example of Fig. 20 required 4594 messages and had 402 total actuations. This is actually a worst-case example for communication requirements; the large number of intersecting paths means that more traffic control messages are required. Since communication in the hardware system is much faster than actuation, the number of messages has not had a significant impact on operational speed. However, since communication requires power, which is a limited resource, it should be optimized.

Supplying power to modules is difficult. If the modules supply their own power using batteries, their weight and size increases, which requires more power to move them around. In the Crystal, a set of batteries allows a module to operate for over 10 hours[6]. This is partly because during most applications only a small fraction of the units move at the same time. With increased parallelism, the lifetime of the system may go down. Communication uses power and should be optimized. However, the actuators utilize the largest amount of power in the system. The processors also utilize power. At the moment the processors are constantly on, independent of whether the modules are participating in the reconfiguration or not and this causes significant battery drainage. A better approach that would reduce power utilization would control when a processor is on or off, depending on whether its module is active or not. One possibility for improving the power resource is to use solar cells that can be recharged automatically regularly.

X. Conclusions

This paper has presented the PacMan algorithm, with which unit-compressible modular robot systems can plan and execute self-reconfigurations in a parallel and distributed fashion. Analysis shows that nearly any self-reconfiguration can be achieved with this technique. We have also presented an extension to the actuation protocol to enable more reliable parallel execution of paths. Analysis of these extensions shows that PacMan can then successfully perform in parallel any set of paths that does not contain a cycle, and will not deadlock in cases where a cycle exists. We have also presented initial implementation on the Crystal robot hardware, and we hope to continue this work by implementing the extensions to the actuation as well as improving the hardware itself.

[6]We collected this data based on 60 hours of demos at SigGraph 2002 and AAAI 2002.

References

[1] Z. Butler, S. Byrnes, and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, pages 790–6, 2001.

[2] Z. Butler, R. Fitch, and D. Rus. Distributed control for unit-compressible robots: Goal-recogition, locomotion and splitting. *IEEE/ASME Trans. on Mechatronics*, 7(4):418–30, Dec. 2002.

[3] Z. Butler, R. Fitch, and D. Rus. Experiments in distributed control of modular robots. In *Experimental Robotics VIII*, pages 307–16, 2002.

[4] Z. Butler and D. Rus. Distributed motion planning for 3-D robots with unit-compressible modules. In *Workshop on the Algorithmic Foundation of Robotics*, 2002.

[5] A. Castano, W.-M. Shen, and P. Will. CONRO: Towards deployable robots with inter-robots metamorphic capabilities. *Autonomous Robots*, 8(3):309–24, 2000.

[6] C.-H. Chiang and G. Chirikjian. Modular robot motion planning using similarity metrics. *Autonomous Robots*, 10(1):91–106, 2001.

[7] G. Chirikjian. Kinematics of a metamorphic robot system. In *Proc. of IEEE ICRA*, pages 449–55, 1994.

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

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

[10] K. Kotay and D. Rus. Algorithms for self-reconfiguring molecule motion planning. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems*, pages 2184–93, 2000.

[11] W. H. Lee and A. Sanderson. Dynamic analysis and distributed control of the tetrabot modular reconfigurable robot system. *Autonomous Robots*, 10(1):67–82, 2001.

[12] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. of IEEE ICRA*, pages 442–8, 1994.

[13] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D self-reconfigurable structure. In *Proc. of IEEE ICRA*, pages 432–9, May 1998.

[14] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: Self-reconfigurable modular robotic system. *IEEE/ASME Trans. on Mechatronics*, 7(4):431–41, 2002.

[15] A. Nguyen, L. Guibas, and M. Yim. Controlled module density helps reconfiguration planning. In *Algorithmic and Computational Robotics: Proceedings of WAFR 2000*, pages 23–35, 2000.

[16] 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 Conf. and Computers in Engineering Conf.*, pages 1–10, 1996.

[17] 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.

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

[19] B. Salemi, W.-M. Shen, and P. Will. Hormone-controlled metamorphic robots. In *Proc. of IEEE ICRA*, pages 4194–9, 2001.

[20] J. Suh, S. Homans, and M. Yim. Telecubes: Mechanical design of a module for self-reconfigurable robotics. In *Proc. of IEEE ICRA*, pages 4095–4101, 2002.

[21] 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, 1999.

[22] C. Ünsal, H. Kiliççöte, and P. Khosla. A modular self-reconfigurable bipartite robotic system: Implementation and motion planning. *Autonomous Robots*, 10(1):23–40, 2001.

[23] S. Vassilvitskii, M. Yim, and J. Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of IEEE ICRA*, pages 117–22, 2002.

[24] J. E. Walter, E. M. Tsai, and N. M. Amato. Choosing good paths for fast distributed reconfiguration of hexagonal metamorphic robots. In *Proc. of IEEE ICRA*, pages 102–9, 2002.

[25] J. E. Walter, J. L. Welch, and N. M. Amato. Concurrent metamorphosis of hexagonal robot chains into simple connected configurations. *IEEE Trans. on Robotics and Automation*, 18(6):945–56, 2002.

[26] M. Yim. A reconfigurable modular robot with multiple modes of locomotion. In *Proc. of JSME Conf. on Advanced Mechatronics*, pages 283–288, Tokyo, 1993.

[27] M. Yim, D. Duff, and K. Roufas. PolyBot: a modular reconfigurable robot. In *Proc. of IEEE ICRA*, pages 514–20, 2000.

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

[29] E. Yoshida, S. Murata, A. Kamimura, K. Tomita, H. Kurokawa, and S. Kokaji. Motion planning of self-reconfigurable modular robot. In *Proc. of Int'l Symposium on Experimental Robotics*, 2000.

[30] E. Yoshida, S. Murata, S. Kokaji, A. Kamimura, K. Tomita, and H. Kurokawa. Get back in shape! *IEEE Robotics & Automation Magazine*, 9(4):54–60, 2002.

Appendix

I. Glossary

A. General terms

modular robot A robot that consists of multiple independent units (modules).
reconfigurable robot A modular robot whose geometry can change.
self-reconfiguring robot A robot that can change its own geometry.
chain-based modular robot A modular robot in which the modules are arranged as a linked set of linear structures.

lattice-based modular robot A modular robot in which the modules are arranged in a space-filling lattice.
static self-reconfiguration The task of changing the robot's shape in place.
dynamic self-reconfiguration The task of moving in a particular direction using self-reconfiguration to generate the locomotion gait.
surface-moving Actuation for lattice-based self-reconfiguring robots where the modules are constrained to travel on the surface of the robot.
volume-moving Actuation for lattice-based self-reconfiguring robots where the modules are constrained to travel through the volume of the robot.
unit-compressible modules Lattice-based modules that actuate by expansion and contraction.
meta-modules Groups of modules that are treated as atomic units for planning purposes.

B. Terms relating specifically to PacMan

Crystal, Crystal robot The name of the unit-compressible robot system developed at Dartmouth (see Figure 2). Also, a group of modules of this robot.
Atom One module of the Crystal robot system.
Pellet, Path-Pellet A data structure within a module used during actuation to represent one step of a path for one module. Each pellet contains the path id.
Plan-Pellet A data structure within a module used during the planning algorithm to represent one step of a potential path.
Spare module A module which is not required at its current location for the given goal.
Mobile module A spare module which can move without disconnecting the robot. (See Figure 6.)
Stem cell A configuration consisting of a core (square or cube of modules) with projections off of the core. (See Figure 4.)
Core A square (for 2-D systems) or cube (for 3-D systems) of modules within a stem-cell robot.
Dangling atom, dangler A module that is attached to only one other module. (See Figure 7.)
Well-behaved stem cell A stem cell in which no atom is adjacent to two dangling atoms.
Centered atom A module at the center of the edge of the core of a 2-D stem cell. (See Figure 9.)
Mover A module that is currently executing a PacMan path; equivalently, a module for which pellets exist within the system.