Technical Report TR03-002

Texas A&M University Parasol Lab

Algorithms for Filling Obstacle Pockets with Hexagonal Metamorphic Robots

Author: Mary E. Brooks, University of Oklahoma

Advisors: Dr. Jennifer E. Walter, Dr. Nancy M. Amato

Department: Computer Science

6319 Windmill Circle
Dallas, TX 75252

Abstract

The problem addressed is the distributed reconfiguration of a metamorphic robotic system composed of any number of identical hexagonal modules. A metamorphic robotic system is a collection of independently controlled, smaller robots that can move around adjacent robots to change the shape of the overall system. The shape-changing ability of these systems makes them more versatile than conventional mobile robots. Because the modules are homogeneous, the systems are also capable of self-repair.

As part of our research, we developed deterministic, distributed reconfiguration algorithms for planning the reconfiguration when a single obstacle with an irregular surface (e.g., containing pockets) is embedded in the goal. Current reconfiguration algorithms find a bisecting path of goal cells spanning the goal, called the substrate path. Modules fill in this substrate path and then move along the path to fill in the remainder of the goal. We present algorithms to 1) determine if an obstacle fulfills a simple admissibility requirement based on contact patterns, 2) include an admissible obstacle in a substrate path, and 3) determine the order to fill the pocket cells.

I. INTRODUCTION

A *self-reconfigurable* robotic system is a collection of independently controlled, mobile robots, each of which has the ability to connect, disconnect, and move around adjacent robots. *Metamorphic* robotic systems [5], a subset of self-reconfigurable systems, are further limited by requiring each module to be identical in structure, motion constraints, and computing capabilities and to have a regular symmetry so that they can densely pack the plane to form two and three dimensional solids.

In these systems, robots achieve locomotion by moving over a substrate composed of one or more other robots. The mechanics of locomotion depend on the hardware and can include module deformation [6], [12], [13] or rigid motion [1], [7], [10], [11], [19], [20].

Shape changing in these composite systems is envisioned as a means to accomplish various tasks, such as structural support or tumor excision [12], as well as being useful in environments not amenable to direct human observation and control (e.g. interplanetary space). The complete interchangeability of the robots provides a high degree of system fault tolerance. An example of the usage of a metamorphic system is demonstrated in Figure 1.

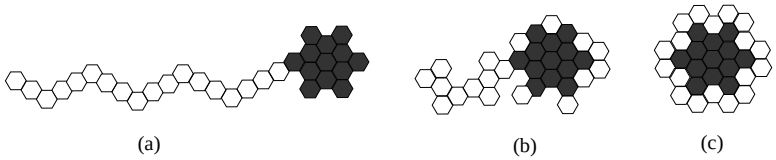

Fig. 1. Metamorphic system morphing from a bridge configuration to envelop an obstacle composed of faulty modules. Faulty modules are dark, and the mobile robots are white. Time is progressing from (a) to (c).

The motion planning problem for a self-reconfigurable robotic system is to determine a sequence of robot motions required to go from a given initial configuration (I) to a desired goal configuration (G). Many existing motion planning strategies rely on centralized algorithms to plan and supervise the motion of the system components [5], [8], [9], [12], [13], [18]. Others, such as [2], [10], [11], [19], and [20], use distributed approaches which rely on heuristic approximations or require communication between robots during the reconfiguration process.

We focus on a system composed of planar, hexagonal robotic modules as described by Chirikjian [6]. We consider a distributed motion planning strategy, given the assumption

of global knowledge of G and an initial configuration I that intersects G and is a straight chain. Our distributed approach offers the benefits of localized decision making, the potential for fault tolerance, and requires less communication between modules during reconfiguration than other approaches. Walter et al. has previously applied this approach to the problem of reconfiguring a straight chain to an intersecting straight chain [16] or to a goal configuration that satisfies a general "admissibility" condition [15]. In [14], they presented algorithms and heuristics to choose paths for fast reconfiguration.

In this paper, we present a new definition of an admissible obstacle. We consider the scenario in which a single obstacle is present in the goal configuration. Additionally, we allow the obstacle to contain "pockets" - informally, a depression in the obstacle surface. We provide a more formal definition of pockets later in the paper. We introduce algorithms to give an irregular obstacle a traversable surface and to incorporate the obstacle in the substrate path prior to distributed reconfiguration.

A. Related work

In this section, we include a discussion of reconfiguration algorithms for when obstacles are present, but, due to limited space, we do not discuss related work on general metamorphic robots. Reconfiguration algorithms have been developed for hexagonal [6], [12], [10], square [4], [12], cubic [13], [2], [11], and rhombic dodecahedral [19], [20], [1] metamorphic robots.

Obstacle-related reconfiguration: The presence of obstacles in the reconfiguration environment is briefly considered by Chirikjian in [5]. In this approach, a heuristic is used to attract modules to an obstacle so that they converge around it. Bojinov et al. [1] provide a distributed strategy for grasping objects in the environment using rhombic dodecahedral modules by probabilistically "growing" extensions to grasp the obstacle. Butler et al. [3] present a rule set for distributed locomotion of layers of cubic modules over obstacles on the traversal surface.

B. Our approach and problem definition

Our objective is to design a distributed algorithm that will cause the modules to move from an initial straight chain configuration, I, in the plane to a known goal configuration, G. This algorithm should ensure that modules do not collide with each other, and the reconfiguration should be accomplished in a minimal number of rounds.

The major differences between our approach and those of other researchers are summarized below:

1) *Our algorithms require no message passing.*
2) *Our algorithms are deterministic, ensuring that the reconfiguration can be accomplished without deadlock or collision provided I is a straight chain and G and the obstacle fulfills simple admissibility requirements.*
3) *Our algorithms are particular to the motion constraints of planar, hexagonal modules.*
4) *Our algorithms are the first to consider the complete envelopment of obstacles.*

In Section II, we describe the system assumptions and the problem definition. Section III describes our algorithm for determining admissibility of a traversal surface. Section IV introduces simple admissibility conditions for obstacles and presents a method for reconfiguration in the presence of obstacles. Section V presents algorithms for reconfiguration when complex obstacles containing *pockets* are embedded in the goal. Section VII

discusses the simulation modeling our algorithms. Section VIII provides a discussion of our results and future work.

II. System model

The plane is partitioned into equal-sized hexagonal cells and labeled using the same coordinate system as described by Chirikjian [5].

A. Assumptions About the Modules

Our model provides an abstraction of the hardware features and the interface between the hardware and the application layer.

- Each module is identical in computing capability and runs the same program.
- Each module is a hexagon of the same size as the cells of the plane and always occupies exactly one of the cells.
- Each module knows at all times:
  - its location (the coordinates of the cell that it currently occupies),
  - its orientation (which edge is facing in which direction), and
  - which of its neighboring cells is occupied by another module.

Modules move according to the following rules.

1) Modules move in lockstep rounds.
2) In a round, a module M is capable of moving to an adjacent cell, C_1, iff
   (a) cell C_1 is currently empty,
   (b) module M has a neighbor S that does not move in the round (called the *substrate*) and S is also adjacent to cell C_1, and
   (c) the neighboring cell to M on the other side of C_1 from S, C_2, is empty.
3) Only one module tries to move into a particular cell in each round.
4) Modules cannot carry, push, or pull other modules, i.e., a module is only allowed to move itself.

If the algorithm does not ensure that each moving module has an immobile substrate, as specified in rule 2(b), then the results of the round are unpredictable. Likewise, the results of the round are unpredictable if the algorithm does not ensure that only one module will move into a particular cell in each round, rule 3.

III. Admissible traversal surfaces

In this section we briefly present a classification of traversable surfaces in a hexagonal system under the motion constraints presented in Section II. These definitions are described in detail in [17].

A. Admissible Traversal Surfaces

In the following definition, let set T represent a particular set of cells in the plane, such as the set of all goal cells.

Definition 1: The *d-clearance* of a cell $i \in T$ is the number of consecutive cells $\notin T$ that are aligned along the outgoing vector originating in the center of cell i and normal to side d, $d \in \{\text{N, S, NW, SW, NE, SE}\}$.

Figure 2 shows a cell with a *SE-clearance* $= 3$.

Informally, the d-clearance is the distance from a cell in direction d to the next cell of the same type.

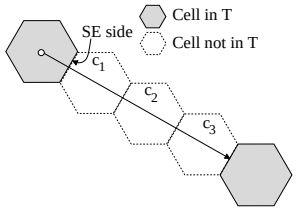

Fig. 2. Cell with *SE-clearance* $= 3$.

Definition 2: A *segment* of s is a contiguous subsequence of s of length ≥ 2. In a *d-segment*, each cell is direction d of the previous.

During our reconfiguration algorithm, modules travel from west to east, so we present our definition of an admissible traversal surfaces for traversing west to east. Then we make a similar definition for the situation where traversal is from east to west.

Definition 3: An *east-monotone admissible traversal surface* is a contiguous sequence $s = c_1, c_2, \ldots, c_k$ of distinct cells from a set T such that

1) no c_i is the end of a NW or SW segment (i.e., each cell is adjacent to the previous, but not to its west),
2) the north side of s is such that
   a) every c_i in a N segment with NW-clearance > 0 has NW-clearance ≥ 3, and
   b) every c_i in a S segment with NE-clearance > 0 has NE-clearance ≥ 3; and
3) the south side of s is such that
   a) every c_i in a S segment with SW-clearance > 0 has SW-clearance ≥ 3, and
   b) every c_i in a N segment with SE-clearance > 0 has SE-clearance ≥ 3.

A *west-monotone admissible traversal surface* is defined exactly as the east-monotone admissible surface except that in Definition 3, part 2, NW is substituted for NE and vice versa, and SE is substituted for SW and vice versa in Definition 3, part 3.

B. Admissible goal configurations

In this section, we define admissible goal configurations and describe a centralized algorithm that tests whether a given configuration is admissible, i.e., whether it contains an *admissible substrate path*.

Without loss of generality, assume I is a straight chain that intersects G in exactly one cell on the perimeter of G. The number of modules in I and the number of cells in G is n.

Let $G_1, G_2, \ldots, G_m$ be the columns of G, such that G_1 is the column in which I intersects G and G_m is the column furthest from column G_1. Suppose that G is oriented such that column G_1 is the westmost column, G_m is the eastmost column, and each column of G is a contiguous straight chain oriented north-south.

An admissible substrate path was defined formally in [15]. Informally, this is a path of cells that spans G and forms an east-monotone admissible traversal surface.

Definition 4: G is an *admissible goal configuration* if there exists an admissible substrate path in G.

IV. OBSTACLES

Sections IV and V describe the major part of the work I did for this research project. In this section, we consider the presence of a single obstacle in the coordinate system and present a strategy for reconfiguration when obstacles are present in the goal.

An obstacle is a sequence of one or more "forbidden cells" that modules cannot enter. We consider obstacles that are composed of hexagons of the same size as the cells of the plane and we assume each hexagon in the obstacle occupies exactly one of the cells. Since at this point, we consider only the presence of obstacles that are contained completely within G, the cells surrounding the obstacle are goal cells. Modules may touch obstacle perimeter cells and may use them as substrate for movement.

Definition 5: An obstacle contains a *pocket* if, for any obstacle cell i contacting one or more goal cells, i has non-infinite *d-clearance* in any direction d (see Figure 3), where T is the set of obstacle cells. We call any goal cell contained within a pocket a *pocket cell*.

Pocket cells are identified by checking the *d-clearance* of all obstacle cells present in G. Informally, any goal cell found on a straight-line path between two obstacle cells is a pocket cell.

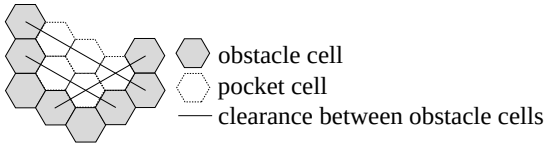

Fig. 3. Pocket cells within obstacle.

Definition 6: An obstacle is *admissible* if

1) it is completely enclosed by the cells of G,
2) all pocket cells, if any exist, have *free* contact patterns with obstacle cells (cf. Figure 4), and
3) all pocket cells are contiguous (i.e., there is only one pocket in the obstacle).

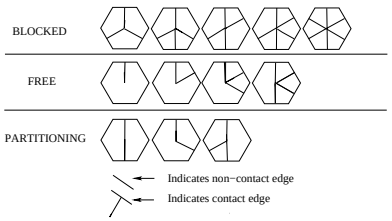

Fig. 4. Contact patterns possible in algorithm.

Notice that part 3 of Definition 6 restricts the set of admissible obstacles to be only those that contain a single pocket. We relax this restriction to allow multiple pockets in a single obstacle in Section V-B.

In Figure 4, the contact patterns labeled "free" represent any pocket cell with at least 2 adjacent sides that are not in contact with an obstacle cell. Recall that the deformable modules we model can move into, or out of, an opening formed by two adjacent non-contact sides. The "partitioning" and "blocked" contact patterns represent openings in the obstacle that are either too small for a module to move into, or that would cause the blocking of modules from filling all cells in the pocket.

It is important to note that the pockets in an obstacle are not necessarily part of an admissible east-monotone traversal surface (although they may satisfy that definition). For obstacle pockets, we require only that the pockets can be filled in without collision or deadlock, i.e., modules will move into, but not out of, the pockets.

In [14], we present algorithms for reconfiguration with obstacles that do not contain pockets. These algorithms transform the obstacle surface into a traversable surface by *repairing* certain cells. These algorithms will not be covered in this paper.

V. Algorithm to Fill Pocket

If the obstacle contains pockets, these pockets must be filled in so that the obstacle surface can be traversed. After ordering the cells for the western substrate path, we determine the order to fill the pocket. Our algorithm fills the pocket before modules traverse the obstacle surface or fill in goal cells to the east of the pocket because the pocket may not have an admissible traversal surface.

A. Algorithm to Fill a Single Pocket

The FillPocket algorithm determines the ordering in which cells will be filled in an admissible obstacle pocket by considering the number of free sides each pocket cell has, the distance of the cells from the entrance of the pocket, and whether they are separated from the pocket opening by any *two-cell gaps*. Two-cell gaps provide sufficient clearance for modules to move through, but do not provide sufficient clearance for two modules moving different directions in the gap to pass each other.

To precisely calculate the cells' positions within the obstacle pocket and to find two-cell gaps, our algorithm orders the currently reachable pocket cells in an array called *perimeter*. The ordering begins at the *entrance cell* of the pocket. The *entrance cell* is the closest pocket cell to the substrate path on the *pocket opening* (i.e., the set of all pocket cells adjacent to at least one goal cell that is not a pocket cell). We pick the *entrance cell* from the *pocket opening* by considering which cell on the pocket opening is closest, in terms of distance along the outside obstacle perimeter, to where the western substrate path meets the obstacle. The *perimeter* array contains all the pocket cells a module would roll into, in order, while rotating in the same direction (CW or CCW) on the inside of the obstacle pocket, starting at the *entrance cell*.

Once the *perimeter* array has been created, the two-cell gaps can be easily identified by locating adjacent cells in the *perimeter* that have nonconsecutive perimeter numbers. If two adjacent cells have nonconsecutive perimeter numbers, modules will be crawling to reach the perimeter cells in betweeen the two as other modules are trying to crawl out of those cells causing blockage as the two adjacent cells are not separated by enough distance to accomodate simultaneous movement in both directions.

To avoid this situation, our algorithm finds these two-cell gaps and fills them before sending modules beyond those cells. The algorithm NonconsecutivePerimeterCell (cf. Figure 8) finds the range of cells (i, j) safe to fill. The boundaries i and j, where $i < j$, denote the lower and higher perimeter numbers of the *deepest two-cell gap*. The deepest two-cell gap has the smallest difference between its nonconsecutive perimeter cells and lies within the range of the two-cell gap closest to the entrance cell, possibly including the entrance cell itself. To find the deepest two-cell gap, we first find the closest two-cell gap to the entrance cell by progressing in order along the perimeter until we reach a cell i with a neighboring, nonconsecutive perimeter cell j (i.e., j is on perimeter and $|j - i| > 1$). The lower perimeter number is saved as i and the higher perimeter number is saved as j. Then, the algorithm looks for a two-cell gap within the range (i, j). If a two-cell gap is found, i and j are set to the lower and higher perimeter numbers of those two cells respectively. The process continues until we reach j meaning we have checked the entire range of the last two-cell gap found for deeper two-cell gaps.

At each step, our FILLPOCKET algorithm (cf. Figure 6) chooses the pocket cell in *perimeter* that has the highest number of sides in contact with the obstacle or a filled pocket cell and has a perimeter number within the range (i, j).

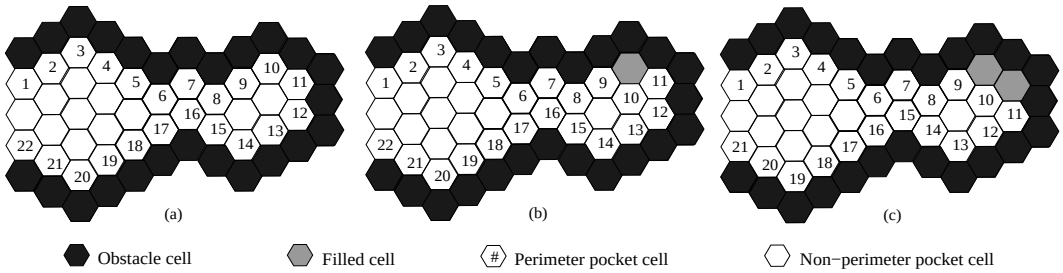

Fig. 5. Sample admissible obstacle with numbered cells in *perimeter*. Non-pocket goal cells not shown. Initial perimeter numbering (a), followed by the first two iterations of the FILLPOCKET algorithm, (b) and (c).

By filling the cells within the boundaries of the deepest two-cell gap, we ensure no modules will crawl back through this two-cell gap causing collisions with the modules behind it. If no two-cell gaps occur in the pocket, the first cell in *perimeter* is i and the last cell in *perimeter* is j.

Figure 5(a) shows an admissible obstacle with cells in the *perimeter* array numbered in the order they would be placed in *perimeter* if cell 1 was chosen as the entrance cell and modules moved with CCW rotation through the pocket. In part (a) of this figure, the cells in *perimeter* numbered 6 and 17 would be the first two-cell gap found in the obstacle pocket. Next, the algorithm would identify cells 7 and 16 and then cells 8 and 15 as being part of a two-cell gap. Since no two-cell gaps occur in the range $[9, 14]$, the cell with the highest number of contacts in the range $[9, 14]$ would be filled first. The cell numbered 10 would be chosen as the first to fill in this case.

After cell 10 in *perimeter* is added to the front of the (previously empty) list of pocket cells to fill in Figure 5(b), the non-perimeter pocket cell that is a neighbor of cell 10 in part (a) is added to *perimeter*. Note that filled cells are treated the same as obstacle cells when renumbering perimeter and determining contact patterns. On the next iteration, in part (b), the FILLPOCKETS algorithm would identify cells 10 and 13 as part of a two-cell gap and would fill the cell in the range $[11, 12]$ with the highest number of sides contacting an obstacle (or filled cell), cell 11. Part (c) of Figure 5 shows the renumbered perimeter after cell 11 is filled.

The entire algorithm to fill in pocket cells is shown in Figures 6 through 8.

```
 1.   entranceCell := pocket opening cell closest to substrate path/obstacle intersection
 2.   Create perimeter array, starting at entranceCell.
 3.   x := FINDNEXTTOFILL()
 4.   while(perimeter.size > 0) // While pocket not full.
 5.     Put x in vector of cells to fill.
 6.     Remove x from perimeter
 7.     Add appropriate neighbors of x to perimeter
 8.     Decrement free side count of neighbors of x
 9.     x := FINDNEXTTOFILL()
10.   end while
```

Fig. 6. Pseudocode for Algorithm FILLPOCKET.

The FILLPOCKET algorithm starts by finding the closest cell on the pocket opening, in terms of distance along the outside of the obstacle from the cell in which the western

substrate path meets the obstacle. Then, the *perimeter* array is created. Next, the algorithm determines the next cell to fill (cf. Figure 7) based on the number of contact sides of each cell and whether the perimeter up to that point has any two-cell gaps (cf. Figure 8).

Once a cell i is found and added to the end of the vector of cells to fill, the perimeter array and the free side count of the neighbors of i are updated.

```
-->  Procedure FINDNEXTTOFILL()
 1.   lowestFound := 6
 2.   (lowestAllowed, highestAllowed) := NONCONSECUTIVEPERIMETERCELL()
 3.   for(i := lowestAllowed to highestAllowed)
 4.      if(perimeter[i].freeSides < lowestFound)
 5.         next := perimeter[i]
 6.         lowestFound := perimeter[i].freeSides
 7.   end for
 8.   return next
```

Fig. 7. Pseudocode for Procedure FINDNEXTTOFILL.

```
-->  Procedure NONCONSECUTIVEPERIMETERCELL()
 1.   high := perimeter.size
 2.   low := i := 0
 3.   while(i < high)
 4.      w := perimeter[i]
 5.      if(|w.perimeterNum − (any neighbor y of w).perimeterNum| ≠ 1)
            // y and w are nonconsecutive neighbors in a two-cell gap
 6.         if(y.perimeterNum < high && y.perimeterNum > low)
 7.            low := i // lowest cell in interior pocket
 8.            high := y.perimeterNum // highest cell in interior pocket
 9.         end if
10.      end if
11.      i++
12.   end while
13.   return (low+1, high −1)
```

Fig. 8. Pseudocode for Procedure NONCONSECUTIVEPERIMETERCELL.

B. Algorithm to Fill Multiple Pockets

As an extension of filling a single pocket, we consider filling multiple pockets in one obstacle. We present simple algorithms for finding multiple pockets in an obstacle and for determining the order to fill the pockets.

To group the pocket cells of one obstacle according to the pocket they are in and also to count the number of pockets, we use depth first search (DFS) on the set of all pocket cells where the pocket cell centers are nodes in the DFS graph and adjacent pocket cells are connected by edges. The vector *pocketCells* initially contains every pocket cell of an obstacle. DFS is executed starting from the first cell in *pocketCells*. In the first execution of DFS, each cell found by the search is removed from *pocketCells* and inserted into the vector at index zero of a two-dimensional vector called *pockets*. The vector *pockets* contains one vector for every pocket in the obstacle. After exhausting DFS, if *pocketCells* is not empty, DFS is run again from the first cell remaining in the vector. During this execution, the cells found are removed from *pocketCells* and inserted into the vector at index one of *pockets*. This process is repeated until *pocketCells* is emptied.

If an obstacle has multiple pockets, we examine the obstacle perimeter outside of the pockets to determine the order to fill the pockets. The algorithm traverses two chains of cells called the *CWPerimeter* and the *CCWPerimeter* starting at the cell in the western substrate path contacting the obstacle. The *CWPerimeter* counts the cells a module would roll into while traversing the perimeter of the obstacle in the CW direction and stops

at the first pocket cell encountered. The *CCWPerimeter* counts the cells traversed in the CCW direction around the obstacle until it reaches a pocket cell.

The counts made by the *CWPerimeter* and *CCWPerimeter* are compared. If the count for the *CWPerimeter* (resp. *CCWPerimeter*) is smaller, the first pocket in that direction is chosen to fill next.

Once we select the pocket to fill first, we run the FILLPOCKET algorithm on that pocket to order the cells inside. The cells of that pocket are marked as filled, and on the next run, that pocket is treated as part of the obstacle. Then, the algorithm to find the next pocket to fill is executed again, which recomputes the counts of *CWPerimeter* and *CCWPerimeter*. This process continues until all pockets are filled.

VI. DISTRIBUTED RECONFIGURATION ALGORITHM

In this section, we give a brief overview of the distributed algorithm that performs the reconfiguration of I to G after the goal and the obstacle are determined to be admissible and the obstacle surface is repaired.

The overall reconfiguration proceeds as outlined in Figure 9.

if obstacle contained in G is admissible and repairable
1. find *admissible substrate path* from G_1 to westmost column of obstacle.
2. find *admissible substrate path* from eastmost cell of repaired obstacle to G_m.
3. fill in cells of G in the following order:
   a) *substrate* west of obstacle, west to east,
   b) pocket cells in order determined by algorithm,
   c) repaired cells north and south of obstacle, west to east,
   d) repaired cells east of obstacle, west to east,
   e) *substrate* east of obstacle (if it exists), and
   f) north and south of *substrate*/obstacle, from east to west.

else report failure.

Fig. 9. Schema for reconfiguration with obstacles.

As the algorithms for calculating the rotation of the modules were presented thoroughly in [14], the rotation of modules will not be discussed in this paper.

VII. SIMULATION RESULTS

We developed an object-oriented, discrete event simulator to test the reconfiguration algorithm. As part of my research, I implemented the algorithms to fill pockets and added other functions to the simulator. The simulator handles goals configurations with simple and complex obstacles and without obstacles.

For all configurations tested, the simulator was successful in reconfiguring the systems using the algorithms presented in this paper. Figure 10 shows three of the obstacle configurations our simulator reconfigured successfully.

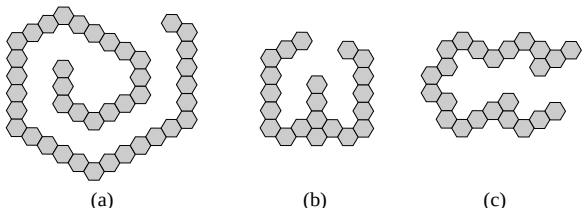

Fig. 10. Examples of obstacle configurations successfully filled by the simulation.

VIII. Conclusions and future work

We have presented a classification of admissible traversal surfaces in the hexagonal grid which allows for generality and coherence when defining admissibility conditions for various objects in the environment. We also addressed the problem of reconfiguration in the presence of a single obstacle enveloped by the goal, using the definition of admissible traversal surfaces to repair the surface of the obstacle prior to distributed reconfiguration.

In this paper, we extended the admissibility requirements of the obstacle to include embedded obstacles with more complex surfaces. We believe that the ability to repair an obstacle surface will also be helpful in designing reconfiguration algorithms for goal configurations containing multiple obstacles.

IX. Biography

Mary Brooks is an undergraduate student at the University of Oklahoma. She expects to graduate in May 2004 with a B.S. in Computer Science and a minor in Mathematics. She is currently participating in the Distributed Mentor Program at Texas A&M.

X. References

[1] H. Bojinov, A. Casal, and T. Hoag. Emergent structures in modulular self-reconfigurable robots. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, Vol. 2, pages 1734–1741, 2000.

[2] Z. Butler, S. Byrnes, and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. In *Proc. of IEEE Intl. Conf. on Intelligent Robots and Systems*, pages 790–796, 2001.

[3] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for a class of self-reconfigurable robots. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 809–816, May 2002.

[4] C.-J. Chiang and G. Chirikjian. Similarity metrics with applications to modular robot motion planning. *Autonomous Robots Journal, Special Issue on Self-Reconfiguring Robots*, Vol. 10, No. 1, pages 91–106, Jan. 2001.

[5] G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 449–455, 1994.

[6] G. Chirikjian, A. Pamecha, and I. Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of Robotic Systems*, Vol. 13, No. 5, pages 317–338, May 1996.

[7] K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H. Asama, Y. Kuroda, and I. Endo. Self-organizing collective robots with morphogenesis in a vertical plane. In *IEEE Intl. Conf. on Robotics and Automation*, pages 2858–2863, May 1998.

[8] K. Kotay and D. Rus. Motion synthesis for the self-reconfiguring molecule. In *IEEE Intl. Conf. on Robotics and Automation*, pages 843–851, 1998.

[9] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfiguring robotic molecule: design and control algorithms. In *Workshop on Algorithmic Foundations of Robotics*, pages 376–386, 1998.

[10] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 441–448, 1994.

[11] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. A 3-D self-reconfigurable structure. In *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pages 432–439, 1998.

[12] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997.

[13] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots Journal, Special Issue on Self-Reconfigurable Robots*, Vol. 10, No. 1,pages 107–124, Jan. 2001.

[14] J. Walter, B. Tsai, and N. Amato. Choosing good paths for fast distributed reconfiguration of hexagonal metamorphic robots. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation*, pages 102-109, May 2002.

[15] J. Walter, J. Welch, and N. Amato. Concurrent metamorphosis of hexagonal robot chains into simple connected configurations. Accepted to *IEEE Transactions on Robotics and Automation*, 2002.

[16] J. Walter, J. Welch, and N. Amato. Distributed reconfiguration of metamorphic robot chains. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pages 171–180, 2000.

[17] J. Walter, B. Tsai, and N. Amato. Enveloping obstacles with hexagonal metamorphic robots. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation, to appear*, Sept. 2003.

[18] M. Yim. A reconfigurable modular robot with many modes of locomotion. In *Proc. of Intl. Conf. on Advanced Mechatronics*, pages 283–288, 1993.

[19] M. Yim, J. Lamping, E. Mao, and J. G. Chase. Rhombic dodecahedron shape for self-assembling robots. SPL TechReport P9710777, Xerox PARC, 1997.

[20] Y. Zhang, M. Yim, J. Lamping, and E. Mao. Distributed control for 3D shape metamorphosis. *Autonomous Robots Journal, Special Issue on Self-Reconfigurable Robots*, 2000.

XI. Acknowledgements

Jennifer Walter of Vassar College and Nancy Amato of Texas A&M served as mentors for Mary Brooks. Brooks is supported by the CRA-W Distributed Mentor Program.