Optimal Kinodynamic Motion Planning for 2D Reconfiguration of Self-Reconfigurable Robots

John Reif
Department of Computer Science, Duke University

Sam Slee
Department of Computer Science, Duke University

***Abstract*— A self-reconfigurable (SR) robot is one composed of many small modules that autonomously act to change the shape and structure of the robot. In this paper we consider a general class of SR robot modules that have rectilinear shape that can be adjusted between fixed dimensions, can transmit forces to their neighbors, and can apply additional forces of unit maximum magnitude to their neighbors. We present a kinodynamically optimal algorithm for general reconfiguration between any two distinct, 2D connected configurations of n SR robot modules. The algorithm uses a third dimension as workspace during reconfiguration. This entire movement is achieved within O($\sqrt{n}$) movement time in the worst case, which is the asymptotically optimal time bound. The only prior reconfiguration algorithm achieving this time bound was restricted to linearly arrayed start and finish configurations (known as the "x-axis to y-axis problem"). All other prior work on SR robots assumed a constant velocity bound on module movement and so required at least time linear in n to do the reconfiguration.**

I. Introduction

The dynamic nature of self-reconfigurable robots makes them ideally suited for numerous environments with challenging terrain or unknown surroundings. Applications are apparent in exploration, search and rescue, and even medical settings where several specialized tools are required for a single task. The ability to efficiently reconfigure between the many shapes and structures of which an SR robot is capable is critical to fulfilling this potential.

In this paper we present an O($\sqrt{n}$) movement time algorithm for reconfiguration between general, connected 2D configurations of modules. Our algorithm accomplishes this by transforming any 2D configuration into a vertical column in a third dimension in O($\sqrt{n}$) movement time. By the Principle of Time Reversal mentioned in [4], we may then reconfigure from that vertical column to any other 2D configuration in the same movement time. Thus, we have an algorithm for reconfiguration between general 2D configurations by going through this intermediate column state.

We begin with a general, connected 2D configuration in the x/y-axis plane, with the z-axis dimension used as our workspace. The configuration is represented as a graph with a node for each module and an edge between nodes in the graph for pairs of modules that are directly connected. The robot modules are then reconfigured in O(1) movement time so that they remain in the x/y-axis plane, but a Hamiltonian Cycle (a cycle that visits each module exactly once) is formed through the corresponding graph and is known.

Using this cycle we create a Hamiltonian Path which may be reconfigured in O($\sqrt{n}$) movement time so that each module has a unique location in the z-axis direction. It is in this stage that our kinodynamic formulation of the problem is necessary to achieve the desired movement time bound. Finally, in two further stages all modules will be condensed to the same x-axis location and y-axis location in O($\sqrt{n}$) movement time. This forms the intermediate configuration of a z-axis column of modules. Reversing this process allows general reconfiguration between connected, 2D configurations.

Organization The rest of the paper is organized as follows. Work related to the results given in this paper is summarized in Section II. In Section III the problem considered in this paper is more rigorously defined and the notation that will be used in the remainder of the paper is given. In Section IV we transform a general, connected 2D configuration of modules into a 2D configuration with a known Hamiltonian Cycle through its modules in O(1) movement time. In Section V that Hamiltonian Cycle is used to allow reconfiguration into the z-axis column intermediate stage. Finally, we conclude with Section VI.

II. Related Work

The results given in this paper primarily apply to lattice or substrate style SR robots. In this form, SR robot modules attach to each other at discrete locations to form lattice-like structures. Reconfiguration is then achieved by individual modules rolling, walking, or climbing along the surfaces of the larger robotic structure formed by other modules in the system. Several abstract models have been developed by various research groups to describe this type of SR robot.

In the work of [7] these robots were referred to as metamorphic robots (another common term for SR robots [2, 3, 8]). In this model the modules were identical 2D, independently controlled hexagons. Each hexagon module had rigid bars for its edges and bendable joints at the 6 vertices. Modules would move by deforming and "rolling" along the surfaces formed by other, stationary modules in the system. In [1] the sliding cube model was presented to represent lattice-style modules. Here each module was a 3D cube that was capable of sliding along the flat surfaces formed by other cube modules. In addition, cube modules could make convex or concave transitions between adjacent perpendicular surfaces.

Most recently, in [4] an abstract model was developed that set explicit bounds on the physical properties and abilities of robot modules. According to those restrictions, matching upper and lower bounds of O($\sqrt{n}$) movement time for an example worst case reconfiguration problem were given. The algorithm for that problem, the *x-axis to y-axis problem*, was

designed only for that example case and left open the question of finding a general reconfiguration algorithm. This paper gives an O($\sqrt{n}$) movement time algorithm for reconfiguration between general, connected 2D configurations. This algorithm satisfies the abstract model requirements of [4] and matches the lower bound shown in that paper.

While the results of this paper are theoretical, the abstract modules used do closely resemble the compressible unit or expanding cube hardware design. Here an individual module can expand or contract its length in any direction by a constant factor dependent on the implementation. Modules then move about in the larger system by having neighboring modules push or pull them in the desired direction using coordinated expanding and contracting actions. Instances of this design include the Crystal robot by Rus et. al. [5] and the Telecube module design by Yim et. al. [6]. In [5, 6] algorithms are given that require time at least linear in the number of modules.

III. Notation and Problem Formulation

Bounds and Equations We assume that each module has unit mass, unit length sides, and can move with acceleration magnitude upper bounded by 1. This acceleration is created by exerting a unit-bounded force on neighboring modules so that one module slides relative to its neighbors. Each module in any initial configuration begins with 0 velocity in all directions. We also assume that each module may exert force to contract itself from unit length sides to 1/2 unit length or expand up to 3 units in length in any axis direction in O(1) time. These assumptions match the abstract module requirements stated in [4]. Friction and gravitational forces are ignored. Our analysis will make use of the following physics equations:

$$F_i = m_i a_i \quad (1)$$

$$x_i(t) = x_i(0) + v_i(0)t + \frac{1}{2}a_i t^2 \quad (2)$$

$$v_i(t) = v_i(0) + a_i t \ . \quad (3)$$

In these equations F_i is force applied to a module i having mass m_i and acceleration a_i. Similarly, $x_i(t)$ and $v_i(t)$ are module i's position and velocity after moving for time t.

Problem Formulation Define the coordinate location of a module in a given axis direction as being the smallest global coordinate of any point on that module. This must be the location of some face of the module since all modules are assumed to be rectangles with faces aligned with the 3 coordinate axes. Let A and B be two connected configurations, each of n modules with each module having a z-axis coordinate of 0 and unit-length dimensions. Let at least one module δ in A and one module β in B have the same coordinate location. The desired operation is reconfiguration from A to B while satisfying the requirements of the abstract model given in [4].

IV. Creating A Hamiltonian Cycle

We begin to lay the groundwork for our algorithm by first showing that a Hamiltonian Cycle may be formed through a collection of modules in a general, connected 2D configuration along the x/y-axis plane. We consider this configuration as a graph, with the modules represented as nodes and connections between touching, neighboring modules represented as edges in the graph. First a spanning tree for this graph will be found. Then the modules will be reconfigured to form a Hamiltonian Cycle through them that essentially traces around the original spanning tree.

A. Finding A Spanning Tree

To facilitate the creation of such a cycle, we will form a spanning tree by alternating stages of adding rows and stages of adding columns of modules to the tree. We later show how this method of creating a spanning tree is useful. We define a *connected row* of modules as a row of modules $i = 1, \ldots, k$ each having the same y-axis coordinate and with face-to-face connections between adjacent modules i and $i+1$ along that row. A *connected column* is defined in the same way as a connected row, but with each module having the same x-axis coordinate. An *endpoint module* has only one adjacent neighbor in the spanning tree.

Define a *free module* as one that has not yet been added to the spanning tree. Consider a case where we have a partially formed spanning tree T and a given module p that has been added to T. Let the *free-connected row* about p be the longest possible connected row of modules which includes p and is composed only of modules, other than p, that are free. Similarly, the *free-connected column* about p is the longest possible connected column of modules which includes p and is composed only of free modules (with p possibly not free). These definitions are used in the algorithm below and examples are given in Figure 1.

CREATE_SPANNING_TREE(Module m)

Input: A module m in the 2D configuration.
Output: A spanning tree of all configuration modules.
Initialize: Add m to spanning tree T.
Add m to queue R.
Add m to queue C.

While: R is nonempty or C is nonempty.
Repeat:
For-each module x in R
Remove x from queue R.
For-each module y in x's free-connected row
Add y to tree T as part of that row.
Add y to queue C.
End-for-each
End-for-each

For-each module x in C
Remove x from queue C.
For-each y in x's free-connected column
Add y to tree T as part of that column.
Add y to queue R.
End-for-each
End-for-each

The algorithm above simply adds modules to the spanning tree through alternating stages of adding free-connected rows and columns of modules. The **For-each** loops are executed synchronously so rows and columns are added one at a time (i.e. no time overlap). The fact that we add these synchronously and that we alternate between row stages and column stages will be used in later proofs.

Spanning Tree Property 1: *If two modules a and b have a face-to-face connection, the same y-axis (x-axis) coordinate, and were each added to the spanning tree as part of rows (columns), then a and b were added as part of the same free-connected row (column).*

An example run of this tree creation algorithm is given in the top row of pictures in Figure 1. The bottom row of pictures shows how the spanning tree may be converted to a *double-stranded* spanning tree, which will soon be discussed.

Lemma 1: *The algorithm described above finds a spanning tree through a given 2D connected configuration along the x/y plane and satisfies Spanning Tree Property 1.*

Proof: To prove that a spanning tree is formed we must show that all modules are added to the tree and that no cycles are created. First, consider a case where the algorithm for creating the spanning tree has finished, but one or more modules have not been added to the spanning tree. Since the given 2D configuration is assumed to be connected, there must be some module q in the group, not yet added to the spanning tree, that has a face-to-face connection to another module d which has been added to the spanning tree.

Yet, in this case q would be a part of either module d's free connected row or its free connected column. So q would have been added to the spanning tree when d was added or in the next stage when the free connected row or column about d was added. Hence, we cannot have that the algorithm has finished and module q is not a part of the spanning tree. Thus, all modules must be part of the spanning tree.

Furthermore, with the exception of the first module added to the tree (module p_0) modules are added to the spanning tree through either a free connected column or a free connected row. Then each module m has at most one 'parent' module in the spanning tree: the module preceding m along the row or column with which m was added. Hence, no cycles are formed and a complete spanning tree has been created. Also, each module is added to the spanning tree exactly once and is considered for inclusion in the spanning tree at most 4 times (once when each of its 4 possible neighbors are added). Thus, the algorithm terminates and, from a centralized controller viewpoint, it takes O(n) computation time.

Finally, to prove that Spanning Tree Property 1 holds, consider two modules a and b that were added to the spanning tree from different rows r_1 and r_2, respectively, but share a face-to-face connection and have the same y-axis coordinate. Since rows are added synchronously one at a time, either r_1 or r_2 was added first. Without loss of generality let it be r_1. Then at the time that module a was added as part of r_1, module b would have been free. Thus, module b would have been included in r_1, the same free-connected row as module a, instead of r_2. Thus, Spanning Tree Property 1 holds for rows. A matching argument for columns also holds (replacing rows r_1 and r_2 with columns c_1 and c_2 and letting a and b have the same x-axis coordinate instead of the same y-axis coordinate). Thus, Spanning Tree Property 1 holds for both rows and columns. □

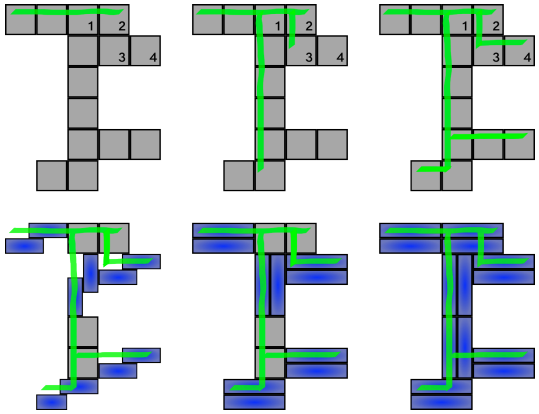

Fig. 1. **Top Row:** Creating a spanning tree by successive rows and columns. **Bottom Row:** Transforming into a double-stranded spanning tree.

B. Doubling the Spanning Tree

With a spanning tree through our given 2D configuration now found, we may focus on turning that into a Hamiltonian Cycle. A basic step for this is to divide the modules along the spanning tree into adjacent pairs of modules. We then reconfigure each of these pairs of consecutive 1×1 dimension modules along that spanning tree into $\frac{1}{2} \times 2$ dimension modules that run parallel to each other in the direction of the original spanning tree. An example is shown in the 3 pictures along the bottom row of Figure 1.

This reconfiguration is done so that each pair of modules maintains the same 1×2 length dimension bounding box around those modules. So, all such reconfigurations may be done locally and simultaneously in two stages. Using two stages allows alternating pairs of modules to be kept stationary so that the overall configuration does not become disconnected. At all times adjacent modules along the spanning tree will share at least a corner connection from the 2D 'bird's eye' view (which is really an edge connection since each module has a 3D rectilinear shape).

The effect of this reconfiguration is to transform our original tree into a "double-stranded" spanning tree. That is, a spanning tree which has two parallel sets of modules along each edge of the tree, but with single modules still permitted at the endpoints of the tree. This allows traversal of the tree by using one module from each pair to travel 'down' a tree branch, and the other module in the pair to travel back 'up' the branch. Further reconfiguration work will transform this intuition into an actual Hamiltonian Cycle.

Lemma 2: *Given the spanning tree T formed in Lemma 1, the modules in that tree may be paired such that the only modules remaining single are endpoint modules in that tree.*

Proof Sketch: Begin with the root module p_0 and pair

modules while moving away from p_0. At the end of any row/column if a module p is left single it is either: (1) an endpoint of the spanning tree, or (2) included in some other column/row where pairs have not yet been assigned. In case (1) no more work is needed and in case (2) recursively solve the subtree that has module p as its root. □

Now that we have the modules in our spanning tree paired together, the next step is to reconfigure those pairs so that the desired double-stranded spanning tree is formed. This reconfiguration and its proof are similar to the *confined cubes swapping problem* introduced in [4].

Lemma 3: *Consider a row of n modules in the x/y plane, labeled $i = 1, \ldots, n$ along the row, each having unit length dimensions. Each pair of adjacent modules along the row may be reconfigured in O(1) movement time so that each has $1/2\times2$ unit dimensions in the x/y plane while maintaining the same 1×2 unit dimension bounding box around each pair in that plane throughout reconfiguration.*

Proof Sketch: Same as given in the *c. c. s. problem* in [4]. □

With the reconfiguration step proven for a single row, we may now state that the entire double-stranded spanning tree may be formed in $O(1)$ movement time.

Lemma 4: *Given the spanning tree formed in Lemma 1, and the pairing of modules along that tree given in Lemma 2, that spanning tree may be reconfigured into a double-stranded spanning tree in O(1) movement time.*

Proof Sketch: Module pairs reconfigure just as in Lemma 3, but here all pairs reconfigure simultaneously in one of two stages. Placing adjacent module pairs into different stages keeps the total configuration connected throughout. □

C. Forming a Hamiltonian Cycle

With the lemmas above we have shown that any connected, 2D configuration of modules may be reconfigured into a double-stranded (DS) spanning tree in O(1) movement time. The next step is to form a Hamiltonian Cycle. Note that a single DS module pair, or a module left single, trivially forms a "local" cycle of 2 or 1 module, respectively. Thus, all that remains is to merge local cycles that are adjacent along the DS spanning tree. This will form a Hamiltonian Cycle that effectively traces around the original tree.

Single modules only occur at the endpoints of the spanning tree. So, we only have 2 types of adjacent local cycle merge cases to consider: (1) DS module pair adjacent to another DS pair, and (2) DS pair adjacent to a single module. Thus, it is sufficient to consider these merge cases from the viewpoint of a given DS pair. Figure 2 illustrates an example DS pair and the 6 possible locations, A-F, of adjacent local cycles. These 6 neighbor locations have 2 types: (1) at the endpoint of the DS pair (locations A and D) and (2) along the side of the DS pair (B, C, E, and F).

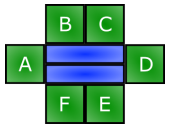

Fig. 2. The 6 possible locations for neighbors adjacent to a double-stranded module pair.

Since the original spanning tree was made with straight rows and columns, adjacent DS pairs along the tree will typically form straight lines or make perpendicular row-column connections. The lone exception is a short row-column-row or column-row-column sequence where the middle section is only 2 modules long. This can sometimes cause adjacent DS pairs that are parallel but slightly offset. An example of this situation, which we refer to as the *kink case*, occurred in the example in Figure 1 (modules 1,2,3 and 4). Later it is also shown more clearly in Figure 4.

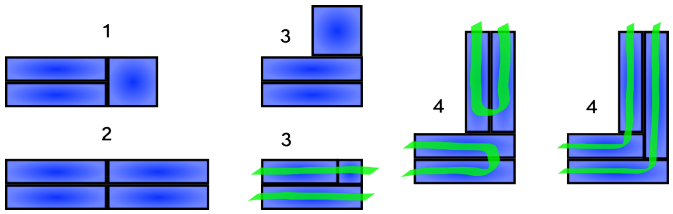

Fig. 3. **(1,2):** Adjacent neighbors occurring at 'endpoint' locations A or D. **(3,4):** Adjacent neighbors occurring at 'side' locations B, C, F, or E.

We now have only 5 types of local cycle merges to handle: (1) pair-single merge at the end of the pair, (2) pair-pair merge at the end of each pair, (3) pair-single merge at the side of the pair, (4) pair-pair merge at the side of 1 pair and end of the other pair, and (5) pair-pair merge at the side of each pair (kink case). A given DS pair must be able to simultaneously merge with all neighboring local cycles adjacent to it along the DS spanning tree. In order to merge adjacent cycles, there needs to be sufficient face-to-face module connections between the 2 cycles to allow one cycle to be "inserted" into the other cycle. In the 2D viewpoint of this paper's figures, this means shared edges between modules rather than just a shared point. We now present 5 reconfiguration rules for handling all 5 cases as well as 4 further rules for resolving potential conflicts between the first 5 rules. Typically, a rule first shows the cycles to be merged and then shows the reconfigured modules after their cycles have been merged in $O(1)$ movement time. In the following descriptions, a module's *length* refers to its longest dimension and a module's *width* refers to its shortest dimension in the x/y-plane.

Rules 1, 2: The leftmost two pictures of Figure 3 show examples of merge types 1 and 2: neighbors at the ends of the DS pair. No reconfiguration of the modules is necessary as the adjacent modules already share sufficient edges to allow cycle merges. Thus, no conflict can arise between multiple applications of rules 1 and 2.

Rules 3, 4: In Figure 3 rules are also given for handling merge types 3 and 4. The two middle pictures in Figure 3 show how to insert a single module into a DS pair while the two rightmost pictures show how to insert another DS pair. In reconfigurations for this rule modules travel O(1) distance and do not exceed the original bounding boxes of the modules shown. Thus, reconfiguration takes O(1) movement time and does not interfere with other modules in the system. Note that rules 3 and 4 may simultaneously be applied at each of the four possible side locations along this DS pair without conflict.

The only possible conflict occurs with the module end that was extended in rule 4, when this same module end is also faced with another rule 4 application along its side (a rule 3 application at this same side location is not a problem). This

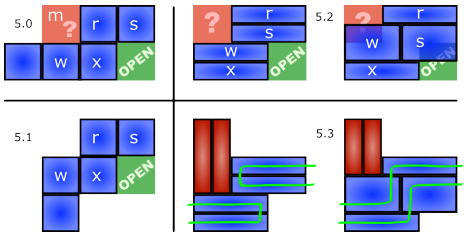

Fig. 4. **Rule 5:** Double-stranded module pairs occurring in a 'kink' case.

second rule would require the module end to contract, creating a direct conflict with the first rule. In this case the conflicted module does contract and the void it was supposed to extend and fill will be handled later by reconfiguration rule 8.

Rule 5: In Figure 4 we present a reconfiguration rule for handling the last type of local cycle merge, the type 5 'kink case'. Dark blue modules are those adjacent to each other along the kink case. Light red modules are from a different part of the spanning tree not directly connected to any of the blue modules. In this example the blue modules were added in a left-to-right, row-column-row order sequence. We now prove two small properties about this kink case arrangement.

Proposition 1: *The green square labeled "open" is not occupied by any module in the original 2D configuration.*

Proof: Since the blue modules in this kink case were added to the original spanning tree in a left-to-right order, modules w and x were added first as part of a left-to-right row (before r and s were added). If the green square were occupied by some module g then it would have been added either as part of a row or part of a column. If added as part of a row then by Spanning Tree Property 1 module g would be part of the same row as modules w and x and thus would be added before modules r and s. Yet, in this case in the next part of the sequence for creating the kink case, modules r and s would have been added as parts of different columns and therefore could not have been paired together as shown.

So, if module g exists it must have been added to the spanning tree as part of a column. In this case, the column would have to be added before modules w and x were added as part of a row. Otherwise module g would be part of that row instead. Yet, this means that module s would have been part of the same column. Thus, modules r and s could not have been paired together as shown and follow modules w and x as part of a left-to-right kink case. Thus, module g could not have been added as part of a column and hence could not exist. Thus, the green square must be unoccupied. □

Proposition 2: *The red 'question mark' square, if occupied by a 'red' module, must have been added to the spanning tree as part of a column and must not have a neighbor to its left adjacent to it in the spanning tree.*

Proof Sketch: Let the supposed red module be labeled m. If m is added to the tree before module w then it would be part of the same column as w or part of the same row as r. Either would prevent the kink case shown. Also, m cannot be added after module s because then it would be part of a column coming up from w – meaning it would not be a red module. So m is added to the tree after w but before s, and thus must

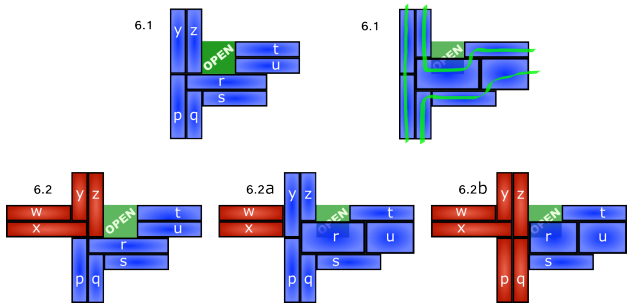

Fig. 5. **Rule 6:** Correcting a conflict between a rule 4 and a rule 5 case.

be part of a column. Also, it cannot have a red neighbor to its left (added from a row) because such a module would first be added as part of a blue column. □

These properties allow rule 5 to work. The green square in the bottom-right corner is open, so module s can expand into that space. If the upper-left corner (question mark space m) is filled by a blue module, by rules 3 and 4 it will first be inserted to the left of module w and will not be a problem. Otherwise that corner is filled by a red module which may be pushed upwards as shown in picture 5.3.

Only two minor issues remain: (1) rule case 4 requires a module with width $1/2$ to extend its length and fill a void, but modules w and s in the kink case have width 1, and (2) module s expands its width into an open space below it, but what if another module from a different kink case also expands into that space? These conflicts are handled by rules 6 and 7.

Rule 6: Figure 5 gives a reconfiguration rule for handling one of the possible conflicts caused by the previous kink case. In all pictures shown the green square marked "open" must be unoccupied. If the kink case in Figure 5 (modules r, s, t and u) was formed right-to-left then by Proposition 1 that green square must have been unoccupied for the kink case to form. Otherwise the kink case is formed left-to-right. Consider if some module m did occupy the green space. Module m cannot be part of a row because by Spanning Tree Property 1 it would be part of the same row as modules t and u, preventing the kink case from forming.

The only remaining possibility is that module m was added to the tree as part of a column. This column must have been added after the bottom row of the kink case. Otherwise either module r or s would have been part of that column and not part of the kink case shown. Thus, r and s are added to the tree before any module in the green space (module m). Yet, then in the same stage when r and s are added as part a row (left-to-right), module m would have been added as part of a row along with modules t and u (left-to-right) since the column to its left was added before modules r and s. Again, this prevents the kink case from forming and so is not possible. Hence, the green square must be unoccupied in all pictures shown. If we only have blue modules (all adjacent along the spanning tree) then reconfiguration may proceed as in 6.1.

The only other conflict from combining the kink case with rule 4 is shown in Figure 5, picture 6.2, and is handled by either 6.2a or 6.2b. In this arrangement, the conflict occurs if

we require DS pair (w, x) to be adjacent to DS pair (y, z) in the spanning tree. It is not important that these module pairs be adjacent, just that each module pair be attached to the tree at exactly one location. So, the tactic used is split and steal: break apart adjacent DS pairs and add one pair, along with its subtree, to a different portion of the spanning tree. First, consider the case where pair (w, x) is the parent of pair (y, z) (i.e. (w, x) precedes (y, z) in the DS tree). In this case split off pair (y, z) and insert it into the 'blue' portion of the spanning tree as shown in picture 6.2a.

In the opposite scenario, pair (y, z) is the parent of pair (w, x). In this case, the blue kink case must have been formed from right to left. Note that if the kink case was formed from left to right, modules p and q would have been added as part of a column. Yet, since pair (y, z) is the parent of pair (w, x), modules y and z would have been added as part of a column. By Spanning Tree Property 1, this means that y, z, p, and q would have to be part of the same column. This is not the case as shown in picture 6.2. Therefore, the kink case must have been formed right to left and pair (r, s) must be the parent of pair (p, q). Thus, we can split off pair (p, q) and add it to the red portion of the spanning tree as shown in 6.2b.

In both 6.2a and 6.2b, the DS module pair chosen to move to a new location was the child pair rather than the parent. Therefore, when the parent and child are split, the parent can stay at its old location. Meanwhile, the child pair may be successfully added to its new location or may be stolen by a third spanning tree location not shown. In this way, no conflicts will arise between multiple applications of Rule 6.

Rule 7: Picture 7.0 in Figure 6 depicts a case where two different kink cases will attempt to expand into the same open space. Picture 7.1 depicts the resulting collision. To resolve this conflict, consider the case where module pair (p, q) is the parent of pair (y, z). In this case we can, once again, apply the "split and steal" tactic and join module pair (y, z) to the blue portion of the spanning tree. Figures 7.2 and 7.3 depict how this can happen depending on whether module y has a neighbor to its left in the blue question mark area shown in picture 7.3. Alternatively, if module pair (a, b) was the parent of pair (w, x), then the same reconfiguration could be applied to add module pair (w, x) to the red portion of the spanning tree instead. If neither case occurs, then pair (y, z) is the parent of pair (p, q), and pair (w, x) is the parent of pair (a, b). In this case, we can add modules y, z, p, and q to the blue portion of the spanning tree as shown in picture 7.4.

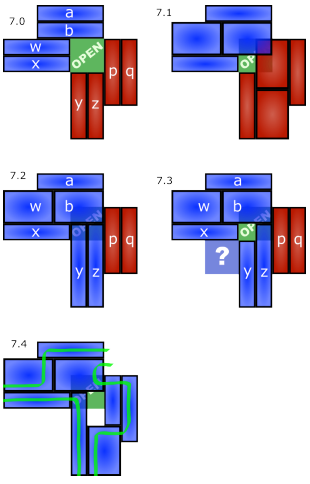

Fig. 6. **Rule 7:** Correcting a conflict between two rule 5 cases.

Rule 8: In Figure 7 rules are given to resolve a conflict between two simultaneous rule 4 applications. The initial state

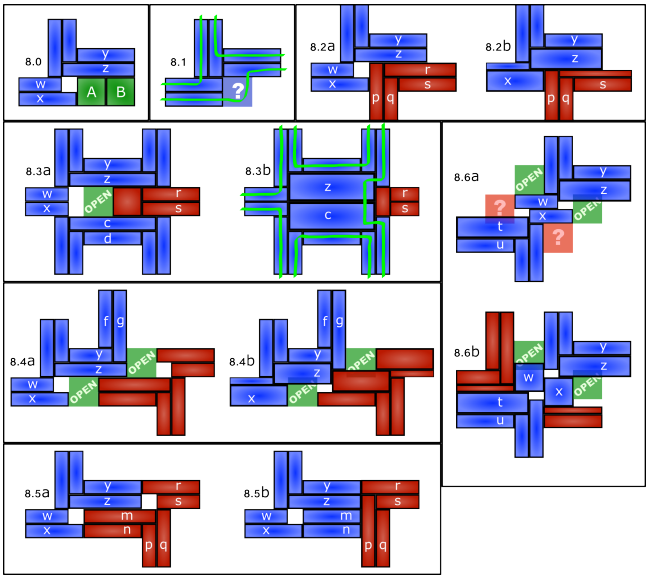

Fig. 7. **Rule 8:** Correcting a conflict between two rule 4 cases.

of this module arrangement is given with the blue modules in picture 8.0. The green squares labeled A and B are given simply to label the two important spaces beneath module z. For example, if space A is occupied by any blue module — one directly adjacent either to pair (w, x) or pair (y, z) on the spanning tree — then no reconfiguration is necessary and the cycles may be merged as shown in picture 8.1. This is true regardless of what type of blue module occupies space A.

Alternatively, if space A is not occupied by a blue module then this conflict case can typically be resolved by the reconfigurations shown in the transition from picture 8.2a to picture 8.2b. Here module z expands downward to make a connection with module x, but module x also expands upward to meet z. To allow this, w must contract its width (shortest dimension in x/y-plane) to $1/4$ and potentially red modules occupying spaces A or B must also contract. Red modules p, q, r and s represent such a case. Note that r contracts but s does not, leaving the boundary between the two modules at the same location. Thus, module r maintains any connections to other modules it previously had (though with smaller area).

The reconfiguration from 8.2a to 8.2b allows red modules in spaces A or B may contract and maintain prior connections. However, conflicts can occur between two simultaneous applications of rule 8. In pictures 8.3a and 8.3b a case is shown where expanding module z must connect not to module x but to another expanding module c instead. Since modules w and x have unlabeled blue modules above and below in these pictures they must have been added by a row. Therefore by Spanning Tree Property 1 there must be an open space between them and any red modules filling the space between z and c. Because of this, any such red modules are surrounded on 3 sides by a blue module or open space and these red modules can be contracted out of the way. An example is given with r, s, and the red single module shown. Now modules z and c can expand until they make contact (a width of 1 for each).

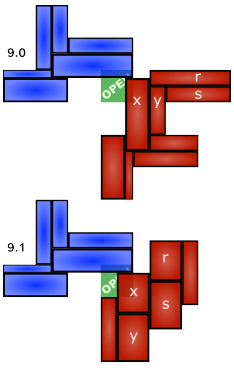

Fig. 8. **Rule 9:** Correcting a conflict between two rule 8 cases or a rule 5 and a rule 8 case.

Two more possible conflicts between rule 8 applications are handled in the transition from picture 8.4a to 8.4b and in going from 8.5a to 8.5b. Here spaces A and B are partially or completely filled by another module facing a rule 8 application. In the case where only space B is filled (8.4a/8.4b) then module z can have its length contracted before expanding its width. A potential connecting module g may then expand its length to maintain the connection. The corresponding red modules do the same. In these pictures it can be proven that the green spaced marked "open" are in fact unoccupied, but the reconfigurations shown work even if these spaces were occupied. If spaces A and B are both occupied by red modules in a rule 8 case, then the conflicted modules are stolen to become blue modules just as was done for rule 7. In 8.5a and 8.5b these modules are m and n and by changing them from the red to the blue portion of the spanning tree this conflict is easily handled.

With this fix, all possibilities have been checked to ensure that blue module z can expand downward if needed. The last remaining problem is that module z must connect to horizontal DS pair module x as in picture 8.2b. This requires module w to contract its width so x can expand. A second rule 8 application could require the opposite thing. This case is depicted in picture 8.6a and the resulting reconfiguration fix is given in 8.6b. Note that here modules w and x achieve the required connections (w to t and x to z) while exceeding their original bounding box by a width of 1/4 unit. Thus, any module previously occupying that space can be pushed out of the way without further conflict (as with the red module given below x in picture 8.6b). If the blue portion of the spanning tree in these pictures began in the lower-left corner (either with pair (t, u) or the vertical blue DS pair next to it) then it can be shown that the green squares marked "open" must be unoccupied. Again, the reconfigurations shown work without creating further conflicts even if these spaces are filled.

Rule 9: When the modules in rule 8 expand outside their original bounding boxes any modules in those locations are pushed out of the way without conflict. Rule 9, depicted in Figure 8, shows that this is true even if those "previous" occupants were modules that underwent a rule 5 or another rule 8 application. With all potential conflicts between rules now resolved, we have a complete rule set for transforming the DS spanning tree into a Hamiltonian Cycle.

Lemma 5: *Reconfiguration Rules 1-9 successfully transform a DS spanning tree into a known Hamiltonian Cycle through the modules in O(1) movement time.*

Proof Sketch: A cycle is formed by merging adjacent local cycles. Five types of merges are needed. Rules 1-5 handle these types. Rules 6-9 resolve any conflicts. Rules 1-4 are used in unison, then rules 5-9, in $O(1)$ total move time. □

V. General 2D Reconfiguration

In the previous section we showed that any given 2D configuration of modules along the x/y plane could be reconfigured in O(1) movement time so that a known Hamiltonian Cycle through those modules was formed. Now, given such a 2D Hamiltonian Cycle, in this section we will show how to transform that into a single column of modules in the z-axis direction in O($\sqrt{n}$) movement time. By using this z-axis column as an intermediate step, we will have the ability to perform general reconfiguration between any 2 configurations of modules along the x/y plane in O($\sqrt{n}$) movement time.

Suppose that we have an initial configuration of n modules with unit-length dimensions in a single row along the x-axis. Let the modules be labeled $i = 0, \ldots, n$ along this row so that module i has its leftmost edge at x-axis coordinate $x_i = i$. If z_i is used to denote the z-axis coordinate of module i's bottom face, then each module initially has $z_i = 0$. Recall that the coordinate of module i along some axis direction is the lowest global coordinate of any point on i.

Our goal is to reconfigure these n modules, starting at rest, into an x/z-axis diagonal with length n in each direction and final velocity 0 for each module in all directions. Note that this may be achieved by moving each module i a distance of 1 unit-length upward in the z-axis direction relative to its adjacent neighbor $i-1$. Module $i = 0$ is stationary. This reconfiguration will be completed in movement time $T = 2\sqrt{n-1}$ while meeting the requirements of the SRK model.

All modules must begin and end with 0 velocity in all directions and remain connected to the overall system throughout reconfiguration. The effects of friction and gravity are ignored to simplify calculations, though the bottom face of module $i = 0$ is assumed to be attached to an immovable base to give a foundation for reconfiguration movements.

Lemma 6: *An x-axis row of n modules with unit length dimensions in the z-axis, may be reconfigured into an x/z-axis diagonal with length n in the z-axis direction in total movement time $T = 2\sqrt{n-1}$.*

Proof Sketch: Reconfiguration may be completed by each module traveling distance $x_i = 1$ relative to its neighbor on one side while lifting $\leq n-1$ modules on its other side. Each module has mass $m = 1$ and exerts force $F = 1 = ma$. So each module may accelerate by $\alpha = 1/(n-1)$. Since we need $x_i(T) = \frac{1}{2}\alpha_i T^2 = 1$, then $T = \sqrt{2/\alpha_i} \leq 2\sqrt{n-1}$. □

Corollary 7: *Any 2D configuration of modules in the x/y-axis plane with a Hamiltonian Path through it may be reconfigured in movement time $T = 2\sqrt{n-1}$ such that each module $i = 0, \ldots, n-1$ along that path finishes with a unique z-axis coordinate $z_i(T) = i$.*

Proof Sketch: Same as in Lemma 6. A module's neighbors are those preceding/following it along a Hamiltonian Path (from breaking a single connection in the Hamiltonian Cycle). □

In Section IV we showed that any connected 2D configuration of unit-length dimension modules along the x/y-axis plane could be reconfigured into a Hamiltonian Path of those modules in O(1) movement time. From corollary 7 we have

shown that such a Hamiltonian Path may be reconfigured so that each module $i = 0, \ldots, n-1$ along that path has its own unique z-axis coordinate $z_i = i$. This required movement time $T = 2\sqrt{n-1}$ and created a winding stair step or "spiral" of modules. To reconfigure this new spiral configuration into a z-axis column, all that remains is to move the modules so that they all have the same x/y-plane coordinates.

Performing this contraction motion for each axis direction is similar to the reconfiguration proven in Lemma 6. The difference is that here all modules finish with the same coordinate value rather than begin with the same value. Also, the modules may have different lengths in the x-axis and y-axis directions as a result of reconfigurations used to create the initial Hamiltonian Path. However, those lengths are still no less than 1/2 unit and no more that 3 unit lengths and so we may use the same relative position and velocity analysis as was used in the proof for Lemma 6.

We will first show that all n modules may be reconfigured so that they have the same x-axis coordinate in total movement time $T \leq 4\sqrt{(n-1)}$. In particular, since the bottom module $i = 0$ is assumed to have its bottom face attached to an immovable base, all modules will finish with the same x-axis coordinate as module $i = 0$. Let the relative x-axis location and velocity of module i at time t be denoted as $x_i(t)$ and $v_i(t)$, respectively. These values for module $i = 1, \ldots, n-1$ are relative to the module $i-1$ directly below module i.

Note that if module i has relative location $x_i(t) = 0$ then at time t module i has the same x-axis coordinate as module $i-1$ below it. Module $i = 0$ does not move throughout reconfiguration. Thus, having $x_i(T) = 0$ for each module i would give all modules the same global x-axis coordinate. Finally, let a z-axis "spiral" configuration of n modules be defined as one where for modules $i = 0, \ldots, n-1$ module i has unit length in the z-axis direction, a z-axis coordinate of $z_i = i$ and has either an edge-to-edge or a face-to-face connection with module $i-1$, if $i > 0$, and with module $i+1$, if $i < n-1$. All modules are assumed to begin at rest and we will show that the desired reconfiguration may be completed in movement time $T = 4\sqrt{(n-1)}$ while meeting the requirements of the SRK model.

Lemma 8: *A z-axis spiral configuration of n modules may be reconfigured in movement time $T = 4\sqrt{(n-1)}$ such that all modules have final relative x-axis coordinate $x_i(T) = 0$.*
Proof Sketch: Same approach as in Corollary 7, but modules are condensed rather than expanded. Slightly more time is required as modules may have x-axis length up to 3. □

From the result just proven, we may conclude that the same reconfiguration may be taken in the y-axis direction.

Corollary 9: *A z-axis spiral configuration of n modules may be reconfigured in movement time $T = 4\sqrt{(n-1)}$ so that all modules have final relative y-axis coordinate $y_i(T) = 0$.*
Proof Sketch: Same as for Lemma 8. □

Theorem: *Let A and B be two connected configurations of n modules, with each module having a z-axis coordinate of 0 and unit-length dimensions. Let modules δ in A and β in B have the same coordinate location. Reconfiguration from A to B may be completed in O($\sqrt{n}$) movement time while satisfying the requirements of the SRK model.*
Proof Sketch: From configuration A, Lemma's 1-5 create a Hamiltonian Cycle in O(1) time. Lemma 6 – Corollary 9 gives a z-axis column in $O(\sqrt{n})$ time. We break the cycle so that δ is the base of the column. Reversing the process gives B. □

VI. Conclusion

In this paper we presented a novel algorithm for general reconfiguration between 2D configurations of expanding cube-style self-reconfigurable robots. This algorithm requires O($\sqrt{n}$) movement time, met the requirements of the SRK model given in [4], and is asymptotically optimal as it matches the lower bound on general reconfiguration given in [4]. In addition, it was also shown that a known Hamiltonian Cycle could be formed in any 2D configuration of expanding cube-style modules in O(1) movement time.

There are a number of open problems remaining. For simplicity, in this paper we assumed that all reconfigurations of modules were executed by a centralized controller to permit synchronous movements. In the future, asynchronous control would be preferable. Local, distributed control of reconfiguration movements is also a topic of future interest. In this paper we have ignored the effects of friction and gravity in the SRK model, but frictional and gravitational models should be included in future work. Note that this makes the Principle of Time Reversal invalid. Furthermore, the general problem of reconfiguration between two arbitrarily connected 3D configurations remains open, although we have developed an algorithm for certain types of configurations. Our current work focuses on extending this, and developing simulation software to implement these algorithms with the SRK model.

VII. Acknowledgement

This work has been supported by grants from NSF CCF-0432038 and CCF-0523555.

References

[1] K. Kotay and D. Rus. Generic distributed assembly and repair algorithms for self-reconfiguring robots. In *Proc. of IEEE Intl. Conf. on Intelligent Robots and Systems*, 2004.

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

[3] A. Pamecha, I. Ebert-Uphoff, and G. Chirikjian. Useful metrics for modular robot motion planning. In *IEEE Trans. Robot. Automat.*, pages 531–545, 1997.

[4] J. H. Reif and S. Slee. Asymptotically optimal kinodynamic motion planning for self-reconfigurable robots. In *Seventh International Workshop on the Algorithmic Foundations of Robotics (WAFR2006)*, July 16-18 2006.

[5] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with unit-compressible modules. *Autonomus Robots*, 10(1):107–124, 2001.

[6] S. Vassilvitskii, J. Suh, and M. Yim. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2002.

[7] 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, 2002.

[8] Jennifer E. Walter, Jennifer L. Welch, and Nancy M. Amato. Distributed reconfiguration of metamorphic robot chains. In *PODC '00*, pages 171–180, 2000.