Reconfiguration Planning for Heterogeneous Self-Reconfiguring Robots

Robert Fitch, Zack Butler and Daniela Rus

Department of Computer Science, Dartmouth College
{*rfitch, zackb, rus*}*@cs.dartmouth.edu*

Abstract

Current research in self-reconfiguring robots focuses predominantly on systems of identical modules. However, allowing modules of varying types, with different sensors, for example, is of practical interest. In this paper, we propose the development of an algorithmic basis for heterogeneous self-reconfiguring systems. We demonstrate algorithmic feasibility by presenting $O(n^2)$ time centralized and $O(n^3)$ time decentralized solutions to the reconfiguration problem for n non-identical modules. As our centralized time bound is equal to the best published homogeneous solution, we argue that space, as opposed to time, is the critical resource in the reconfiguration problem. Our results encourage the development both of applications that use heterogeneous self-reconfiguration, and also heterogeneous hardware systems.

1 Introduction

The promise of self-reconfiguring (SR) robotics is to endow systems with the ability to change shape to match structure to task. The *reconfiguration problem*, which asks how to compute these shape changes, is thus a central research question. Reconfiguration has been studied extensively in systems where all modules are identical, known as *homogeneous* systems, but the *heterogeneous* version of the reconfiguration problem, where modules are allowed to be unique, has not received the same attention and appears to be more difficult. However, no published analysis exists. We believe that the heterogeneous reconfiguration problem is an important research question both practically (useful in even primarily homogeneous systems) and theoretically. In this paper we propose solutions to heterogeneous self-reconfiguration with running times matching their homogeneous counterparts.

Although the elegance of a purely homogeneous system is appealing, practical issues suggest some degree of heterogeneity is desirable. Design tradeoffs exist between homogeneous and heterogeneous systems. Castano and Will [3] characterize the issue as a balance between the hardware complexity of increasing the functionality of the base module in a homogeneous system and the software complexity, and loss of redundancy, of a system that uses specialized modules. Certainly it is difficult to argue in favor of using thousands of cameras or expensive sensors to maintain homogeneity when one or two would suffice. Note that the addition of even one special module to a homogeneous system violates the assumptions of current reconfiguration algorithms. The "software complexity" of the resulting system is assumed to be increased, but this has not been proven. We are interested in addressing this question.

The challenge in developing heterogeneous reconfiguration algorithms is that modules can no longer substitute for each other. Consider the case of a 2D system of square modules with unique identifiers to model the presence of cameras, radios, or other capabilities. This problem seems strongly related to the *Warehouse Problem*, which was proved intractable even in the 2D case with rectangular objects [5]. Although the reconfiguration problem imposes connectivity and other constraints not present in the Warehouse Problem, this similarity indicates that heterogeneous self-reconfiguration is also very difficult.

Our approach relies on the availability of free space in the reconfiguration problem. In fact, the Warehouse Problem has been shown to be polynomial-time solvable given sufficient free space [13]. The main contribution of this paper is the surprising result that heterogeneous self-reconfiguration can be solved in lattice-based systems using asymptotically no more time than the homogeneous problem requires. We present both centralized and decentralized 3D solutions that assume arbitrary free space, which we define as *out-of-place* solutions. Developing *in-place* solutions, where free space is limited, as well as determining exactly how much free space is required by in-place solutions, is left to future work.

The paper is organized as follows. In the following section, we review related work. We then describe the model under which we developed our algorithms, called the *Sliding Cube* model. Centralized and decentralized algorithms are presented in Section 4, followed by discussion and future work.

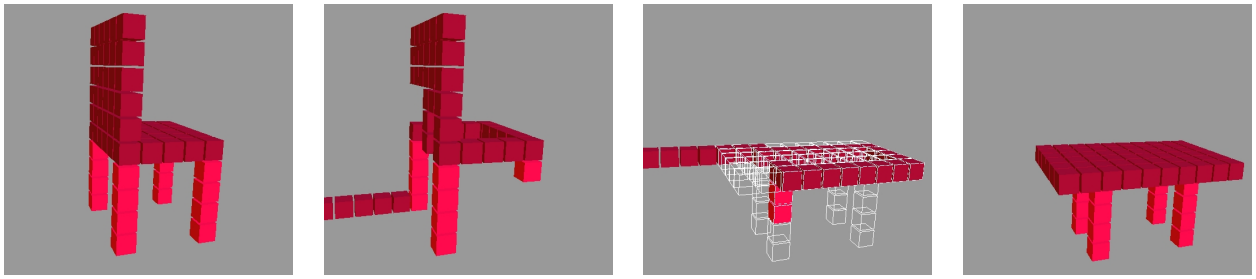

Figure 1: *Simulation of chair-to-table reconfiguration with two module types. Light modules forming legs of the chair map to legs of the table. Dark modules in the chair form the top of the table. This reconfiguration was planned with the out-of-place MeltSortGrow algorithm presented in this paper.*

2 Related Work

The algorithms presented in this paper focus on lattice-based systems. Several such systems have been designed and constructed in hardware [10, 6, 18, 7, 9, 15, 11].

Planning and control for self-reconfiguration in homogeneous systems has been studied by numerous groups. One approach is to develop centralized algorithms [17, 7, 11, 19]. Alternatively, work in decentralized planning begins with the Fracta system, using precompiled sets of rules [8, 14]. A similar algorithm is given by Hosokawa et al. [6]. Yim, Duff and Roufas [18] present a distributed controller for Proteo modules that achieves arbitrary shapes. A later approach in distributed planners is the use of message-passing. Salemi, Shen and Will [12] propose a control system for CONRO using a message-passing scheme called *Digital Hormones*. The problem of distributed reconfiguration for unit-compressible modules was solved by a combination of the *Pacman* algorithm developed by the Dartmouth group [1] and later modifications and analysis by Vassilvitskii, Yim and Suh [16]. Decentralized algorithms that use reconfiguration as part of locomotion are presented in [2].

All previous work in reconfiguration planning assumes primarily homogeneous or bipartite systems. The algorithms we propose, in contrast, are the first to support module uniqueness.

3 Sliding Cube Model

We would like to design algorithms not for one specific robot, but for a class of module hardware. We will extend a model we introduced in our previous work [2] that was shown to be easily instantiated to several module types [2, 4, 6] and define the *Sliding Cube*. The module is a cube with connectors on all faces. Modules connect face-to-face to any other module, and have two motion primitives: sliding across another modules, and making a convex transition. These primitives allow a single module to move across the surface of a robot shape, or through the robot's volume using tunnelling techniques. All modules are of the same size, but to support simple heterogeneity we assign each module a type, or *class ID*.

Systems of Sliding Cubes have properties similar to Crystal [11] or Telecube Robots [16] using *metamodules*, a group of modules treated as a single unit with additional motion capabilities. In particular, any Sliding Cube robot has at least one *mobile* module, where a module is mobile if it can move without disconnecting the structure of the robot. This is shown by the following lemma:

Lemma 1. *Any Sliding Cube robot contains at least one mobile module on its surface.*

Proof. We prove this based on induction on the number of modules in the robot, similar to that given by [11]. Details of the proof are omitted for space considerations. □

Although the level of heterogeneity in the Sliding Cube only consists of unique labels, it is possible to extend the model to encompass a greater degree of heterogeneity. Each module can be given connectivity constraints to model different types of connectors, more complicated motion primitives can be defined, or modules can vary in size. In the future, we would like to develop reconfiguration algorithms for models with these extensions.

4 Heterogeneous Reconfiguration Planning

The *reconfiguration problem* is central to research in SR robots. The problem is to compute a feasible plan that when executed from an initial configuration C results in a specified goal configuration C'. A plan is *feasible* if it maintains the property of connectivity and consists of valid primitive motions for the module actuation model specified. Currently we do not consider dynamics of the intermediate steps. An example reconfiguration is shown in Figure 1.

Existing reconfiguration algorithms rely heavily on module interchangeability. Instead of physically moving a

certain module from one point to another, it can be relocated virtually by shifting the original module out of position and shifting *any* other module into the goal position. This technique is efficient since no one module needs to move very far. Virtual module relocation becomes restricted in heterogeneous systems, however, since not all modules are identical. This seems to indicate that heterogeneous reconfiguration is much more difficult than homogeneous reconfiguration, but the complexity of heterogeneous reconfiguration has not been studied in depth.

It turns out that in both the Warehouse and heterogeneous reconfiguration problems, the main issue is not running time, but free space [13]. Like the Warehouse problem, we show in this paper that heterogeneous reconfiguration is solvable in polynomial time given sufficient space. We define a solution as *out-of-place* if the quantity of space available in which to move modules is assumed to be unbounded. Conversely, in an *in-place* solution the space available in which to move modules is constrained to the union of the start and goal configurations plus a constant-sized crust of space around this union. Both in-place and out-of-place algorithms exist for homogeneous reconfiguration.

In this section we describe an out-of-place solution, *MeltSortGrow*. We present both centralized and decentralized versions of the algorithm, analyze correctness and running time, and describe its implementation.

4.1 MeltSortGrow Algorithm

We developed the algorithm *MeltSortGrow* as an out-of-place solution to the heterogenous reconfiguration planning problem with the Sliding Cube actuation model. The planner takes two robot configurations as inputs, and outputs a sequence of module motion primitives that transform the start configuration into the goal. Centralized MeltSortGrow runs in $O(n^2)$ time and generates a plan of length $O(n^2)$, where n is the number of modules in the robot.

Our approach is based on an early solution to homogeneous reconfiguration, *MeltGrow* [11]. As in MeltGrow, rather than transforming from the initial configuration directly to the goal configuration we generate an intermediate configuration that is easy to reach from any initial condition, and easy to transform into the goal configuration. In this case we choose a single line, or chain, as a simple intermediate configuration. We first plan a reconfiguration from the start configuration to the intermediate structure. Then we sort the modules in the intermediate structure to prepare for the final step, where we plan a reconfiguration from the sorted intermediate structure to the goal shape. This algorithm outline is listed in Algorithm 1. We now formally present the algorithm, prove correctness, completeness and running time, and describe results in simulation.

Algorithm 1 Generic centralized out-of-place algorithm for heterogeneous self-reconfiguration.

```
1: "Melt" configuration into 1D linear chain
2: Compute feasible assembly order for goal shape
3: Sort chain by assembly order
4: while Reconfiguration is not complete do
5:     Move next module into final position
```

Melt. The objective of the Melt step is to compute a plan that reconfigures the initial configuration into the intermediate configuration, a line. As in MeltGrow, we repeatedly choose a module in the initial configuration and move it to a free position in the intermediate structure. To choose a module in the initial configuration, we identify all mobile modules using standard graph search techniques. Then we find a path from any mobile module to the end of the chain, called the *tail*.

Algorithm 2 Melt algorithm builds intermediate structure (1D chain).

```
1: Intermediate configuration I
2: while Modules remain in initial configuration do
3:     Find articulation points
4:     Find path p from root of I to non-articulation point
       module m using breadth-first search
5:     Move m to end of I using p
```

The melt step is specified in Algorithm 2. To begin, we choose one leftmost module from the initial structure and label this the *root*. We will grow the intermediate structure to the left of the root (since by our choice of root we know there are no other modules to its left). A module is mobile if it is not an articulation point in the module connectivity graph, and if it is on the surface of the structure. In line 3 we compute articulation points, and line 4 chooses a surface module from the set of non-articulation points. If we begin at the tail, and search all possible module paths, then any module we reach must be on the surface. Observe that in the Sliding Cube model, the only reachable positions are adjacent to existing modules. We call these *path* positions. In line 4 of the algorithm, we search through path positions using breadth-first-search (BFS), starting from the tail position and terminating when we reach a non-articulation point module. The resulting path is transformed into a motion plan and executed in line 5 by reconstructing the motion primitives used in the BFS path. We repeat this procedure until all modules have been moved, which is once per module. See Figure 2 for an illustration of this step.

Sort. The next phase of the algorithm is to modify the intermediate structure such that it is possible to easily grow the new configuration in the final phase. Assembly order is important since specific modules must be moved into their assigned positions. We approach this problem

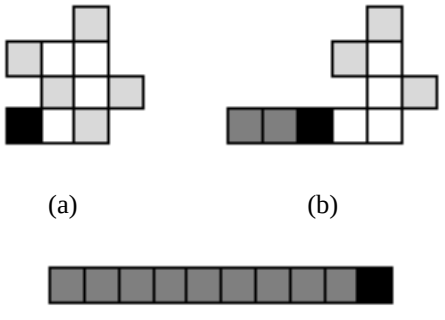

Figure 2: *Illustration of Melt step. In (a), the root module (lower-left) is darkened and mobile modules are shown in light grey. The state after two modules have been moved is shown in (b), with modules in the intermediate structure shown in medium grey. The complete intermediate structure is drawn in (c).*

by sorting the modules in the intermediate chain, where the sorting corresponds to a feasible assembly order of the goal configuration. Therefore we must compute a feasible assembly order and then physically sort the modules according to this sequence.

Algorithm 3 Compute feasible assembly order and sort chain.

1: Intermediate configuration I
2: **for** counter c = 1 to n **do**
3: Find articulation points in goal configuration C'
4: Find path p from root to non-articulation point module m in C' using BFS
5: Label m with c and mark as deleted
6: Move left half of I on top of right half, forming rows a and b
7: **while** a is not empty **do**
8: Find module m in a with minimum label
9: Move m to leftmost unoccupied position in row below b
10: Repeat for row b
11: Merge rows a and b into one sorted row above a, in the style of MergeSort.

The sort step is specified in Algorithm 3. First, in lines 2-5, we compute the (dis)assembly order of the goal configuration using the melt technique described earlier, although instead of computing a path for each module we simply label it with its assembly order and mark it deleted. This can be thought of as a virtual melt. Reversing the resulting disassembly order yields a total ordering on the modules in the intermediate chain, assuming all modules have unique types. The case where modules share types can be handled by artificially labelling modules as convenient.

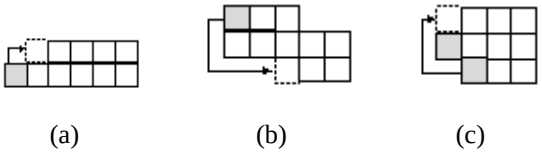

Figure 3: *Steps of the sort phase. Modules next to move are greyed. Construction of the double row is shown in (a). In (b), SelectionSort is in progress. The final sorted order is assembled in (c) by merging the two sorted rows.*

In order to physically sort the chain, we use the simple quadratic-time sorting algorithm SelectionSort, and also one step of MergeSort. Clearly, removing a module from the middle of the line violates connectivity constraints, but the same operation on a double line is permissible. Therefore in line 6 we create a double line by moving half of the modules from the end of the line to form a second row alongside the original row. Now we are free to use SelectionSort to move modules in the top line into sorted position in a new row below the bottom line. Continuing in this way, we obtain two sorted rows. Now we simply merge these two as in MergeSort into a final sorted line adjacent to the double row. By carefully choosing where the new rows are created, the final sorted line is assembled at the same position as the original intermediate structure. See Figure 3 for an illustration of this phase of the algorithm. The following Lemma shows that is always possible to perform sorting:

Lemma 2. *An intermediate configuration with arbitrary module ordering can be reconfigured into a configuration with the same shape and a designated module ordering.*

Proof. Consider the specification of Algorithm 3. The double row can be assembled without disconnecting the structure since modules always move from the end of the top row, which can not be an articulation point since it has only one neighbor. The selection sort step maintains connectivity since all modules at all times are connected to a module in the lower row, and correctly assembles the sorted row since modules are chosen in order. The merge step is correct since we merge two sorted chains, and the three rows are always connected by the rightmost modules. Thus the sorting algorithm is correct. □

Grow. In the final phase of the algorithm we build the goal configuration from the sorted intermediate configuration. Due to the sorting step, it is guaranteed that at any time, there exists a path from the tail module to its unique position in the goal configuration. Therefore we repeatedly find such a path and execute it for each module in the intermediate configuration.

Pseudocode for this phase is listed in Algorithm 4. Line 3 finds a path from the tail to its position in the goal con-

(a) (b) (c)

Figure 4: *Steps of the grow phase. Shading indicates module type. In (a), a path is found from the module at the left end of the intermediate structure to its final position in the goal configuration, shown in (b). This repeats until the complete goal shape is assembled in (c).*

Algorithm 4 Grow goal configuration.

1: Intermediate configuration I
2: **while** Reconfiguration is not complete **do**
3: Find path p from tail of I to goal position using BFS
4: Execute path p

figuration using the same BFS technique described in the melt phase. A motion plan based on the returned path is computed in line 4, and this process repeats for each module. See Figure 4 for an example of this step.

Analysis. Using results from previous sections, we now show that MeltSortGrow is correct and complete. We also show the running time as claimed.

Theorem 1. *The algorithm MeltSortGrow computes a feasible reconfiguration plan of length $O(n^2)$ for all start and goal configurations in $O(n^2)$ time, where n is the number of modules in the system.*

Proof. By the specification of Algorithm 4, each position in the goal configuration is filled only by a module of appropriate type. Therefore, the goal configuration is assembled correctly. It remains to prove completeness. We will show that any start configuration can be reconfigured into the intermediate configuration, and that the intermediate configuration can be reconfigured into any goal configuration.

Recall Lemma 1, which showed that in any configuration at least one surface module is mobile. Using the primitive motions, this mobile module can reach any other position on the surface. By repeatedly relocating mobile modules, we can therefore form the intermediate configuration from any start configuration. This argument also applies to computing the (dis)assembly sequence, since the same melt procedure is used. The intermediate configuration can be sorted by the assembly sequence due to Lemma 2. Now consider the module at the left end of the intermediate configuration, the tail. The tail is clearly mobile, and can move to any position on the surface of the structure without disconnection. Due to the assembly order, the destination position of the tail in the goal is unfilled and is reachable, and therefore the tail can successfully be relocated. Continuing in this way, we can relocate all modules in the intermediate structure and the goal configuration is assembled. Therefore MeltSortGrow is correct and complete for all start and goal configurations.

Running time is easy to see as each of the algorithm's three phases requires $O(n^2)$ time. First, one step of the melt phase requires $O(n)$ time for articulation point finding and $O(n)$ time for BFS, or $O(n^2)$ in total for n modules. Sorting requires $O(n^2)$ time each for selection sort and merge sort. Finally the grow phase performs BFS $O(n)$ times or $O(n^2)$ total. Overall, the algorithm requires $O(n^2)+O(n^2)+O(n^2)=O(n^2)$ time. No more than one primitive move is generated during each time step, so the length of the resulting path is also $O(n^2)$. □

Simulation. We designed and implemented a simulation environment called *SRSim* (self-reconfiguring robot simulator) in which to implement and animate algorithms for self-reconfiguring robots. SRSim is written in the Java programming language using the Java3D API for 3D graphics. It is designed as a set of base classes that are extended by the implementation of a specific algorithm. We implemented MeltSortGrow using SRSim, with screenshots shown in Figures 1 and 5. The simulation reads the start and goal configurations from a file specification, although we are currently designing a graphical tool to make configuration specification easier. SRSim animates all module motions and also search paths as desired.

4.2 Decentralized MeltSortGrow

A centralized solution to heterogeneous reconfiguration planning is a good start in understanding the problem, but it is not sufficient for general use since self-reconfiguring systems are designed to include large numbers of modules. The goals of scalability and redundancy lead us to seek decentralized planners for SR robots. In this section, we extend centralized MeltSortGrow to run in a decentralized manner.

As in our previous work in decentralized algorithms, we adopt the message-passing model of communication;

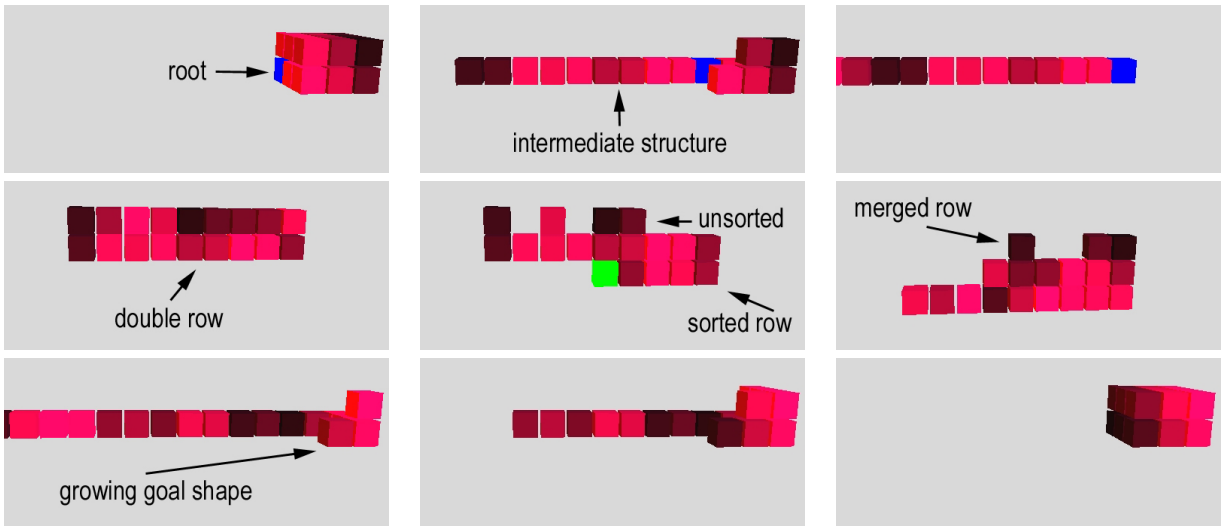

Figure 5: *Screenshots from MeltSortGrow implementation in SRSim simulation. Module shading indicates unique labels. Cube-shape reconfigures from light-to-dark ordering in upper-left to dark-to-light ordering in lower-right.*

modules can communicate only with their immediate neighbors. The main approach is to retain the overall structure of the algorithm, but replace centralized computation with message-passing as necessary. We assume that at the beginning, a single module receives a message to start running the algorithm. Further, we assume that all modules have a copy of the goal configuration in their local memories. If only one module knows the goal shape, then it can propagate this data to the rest of the system.

Melt. Previously, to perform the melt step we identified a root by choosing a leftmost module, and then repeatedly found mobile modules and moved them to the end of the chain formed from the root. Now, since only one module, m_{start}, receives the signal to begin the algorithm, we can find the root by propagating a message depth-first style from m_{start} that computes a relative lattice position for each module. We use depth-first search (DFS) since it is easier to distribute than BFS. Starting at m_{start} $(0, 0, 0)$, each module computes positions for its children and passes this information in a message. In this way, the leftmost coordinate can be returned by comparing the return values of a module's children. Eventually m_{start} will receive the answer and propagate this information to the rest of the system. To melt, think of the root as the initial tail. At any time, there exists exactly one tail. To find a mobile module, the tail initiates a distributed articulation-point labelling algorithm. This is the same as the centralized version except recursive calls are replaced by message-passing to children. The tail then initiates distributed DFS to find the first non-articulation point, which then follows the path back to the tail and becomes the new tail. This ends when DFS fails. Pseudocode is shown in Algorithm 5.

Algorithm 5 Distributed Melt. Pseudocode for single module.

State:
articulationPoint, am I an articulation point

Messages:
start, sent to exactly one module to begin algorithm
  Action: determine root, root executes meltOneModule()
labelArticulationPoints, labels articulation point modules
  Action: DFS-send(*labelArticulationPoints*), setting *articulationPoint* as result
findMobileModule, search to find non-articulation point
  Action: if I am mobile, follow path to tail and execute meltOneModule(), else DFS-send(*findMobileModule*)

Procedures:
meltOneModule()
  DFS-send *labelArticulationPoints*
  DFS-send *findMobileModule*
  if result is false, signal start of sort phase
DFS-send(*message*)
  send *message* to first child, wait for response
  repeat for all children and compute result
  send result in return message to parent

Sort. The next step is to sort the intermediate structure. We first need to determine the sort order by virtually disassembling the goal configuration and then physically sorting the modules. Disassembly in a decentralized manner can be handled in different ways, but the simplest solution is for each module to perform the computation itself to discover its own order. This solution is acceptable due to our assumption that all modules know the goal configuration. Next, the single chain must fold in half to form a double row. The tail module initiates this by mov-

ing around to its final position below the root. Other modules follow until all top row modules have lower neighbors. The last module in the top row signals its row to begin sorting, using distributed BubbleSort. When one pair has finished their comparison, they signal the next pair. This happens back and forth down the row until no more swaps are made. Then the top row signals the bottom row to performs the same sort operation. The two then merge to complete the intermediate configuration. See pseudocode in Algorithm 6.

Algorithm 6 Distributed Sort. Pseudocode for single module.

State:
goal, goal configuration
sortLabel, my order in assembly sequence
swapCount, counter to detect bubble sort termination

Messages:
sort, sent to start sort phase
  Action: propagate *sort*, execute computeSortOrder(), execute formDoubleRow()
bubbleSort, sent to do swap test
  Action: execute handleBubbleSort()
bubbleSortDone, sent to signal bubble sort termination
  Action: if top row, send *bubbleSort* to bottom row. if bottom row, execute merge()

Procedures:
computeSortOrder()
  disassemble *goal* to determine *sortLabel*
formDoubleRow()
  move around structure to form double row
  if I am last module, execute bubbleSort()
bubbleSort()
  compare sortLabel with neighbor and swap if necessary
  send *bubbleSort*(*swapCount*) to neighbor
handleBubbleSort()
  if *swapCount*=0 then propogate *bubbleSortDone*
  otherwise if I am at the end of the line, send *bubbleSort* in opposite direction
merge()
  tails of sorted rows propagate *sortLabel*
  tail with minimum moves to final row and signals next tails when complete, grow phase begins

This specification handles the case when all modules are uniquely identified. The modification to remove this assumption is simple, but requires extra memory. Instead of computing a unique sort position, the modules keep an array of all positions for their class ID. Then when the single chain is assembled, the unique position is resolved by passing a counter down the chain.

Grow. The grow phase reconfigures the intermediate configuration into the final shape. The basic step is that the tail module finds a path, executes it, and signals the new tail. Path planning is done using distributed DFS. When the root module becomes the tail, the algorithm terminates. See Algorithm 7 for pseudocode.

Algorithm 7 Distributed Grow. Pseudocode for single module.

State:
goalPosition, my position in goal configuration
root, am I the root

Messages:
grow, sent to signal start of grow phase
  Action: if I am the current tail, execute moveToGoal()
nextModule, sent to grow next module
  Action: if I am the current tail, and I am root, signal completion. else execute moveToGoal()
findPath(goalPosition), sent to find path
  Action: if I am adjacent to *goalPosition*, return true. else DFS-send(*findPath*(*goalPosition*))

Procedures:
moveToGoal()
  DFS-send *findPath*(*goalPosition*)
  follow path
  propagate *nextModule*
DFS-send()
  specified in Algorithm 5

Analysis. Theorem 1 proved correctness and completeness for the centralized algorithm. Here we note that the decentralized version operates equivalently. In the melt phase, a path is found by searching free positions adjacent to modules. Because the system is connected, we visit all free spaces eventually and a path will be found. Continuing in this way, the intermediate structure is assembled. Sorting is performed in the style of bubble-sort until both halves of the double row are sorted. Because each half is sorted sequentially, connectivity is maintained. Finally, the grow phase assembles the goal shape using search techniques as in the melt phase.

The running time bound is shown by observing that each atomic step in the centralized version is replaced by a step in the decentralized version bounded by $O(n)$, the time to propagate a message. Equating message-passing steps with other atomic steps, the overall running time is thus $O(n^2 n) = O(n^3)$, with $O(n^2)$ moves generated.

Simulation. We have implemented this algorithm in the SRSim simulator described earlier. Each module runs in a separate thread to approximate asynchronous communication.

5 Discussion and Future Work

While previous work in reconfiguration planning in SR systems has assumed homogeneity, this paper promotes the development of planners for heterogeneous systems. We describe a system model that incorporates heterogeneity and models a class of existing SR modules. We

developed algorithms that solve the reconfiguration problem in this model, and present both centralized and decentralized out-of-place solutions. We show that in the out-of-place case, the running time of the heterogeneous planner is equivalent to the fastest published homogeneous solution.

The significance of this paper is that it challenges the presumption that heterogeneous reconfiguration is inherently more difficult than the homogeneous version. Based on the characteristics of the related Warehouse Problem, we conjecture that free space is the resource in contention, instead of the usual metric of running time as a measure of complexity. It is surprising that the heterogeneous problem is not intractable, as previously thought, but is no more difficult than homogeneous reconfiguration given enough free space. This contradicts the intuition that module interchangeability is the most important factor in efficient reconfiguration.

Given these results, a few interesting theoretical questions arise. If heterogeneous reconfiguration is solvable in $O(n^2)$ time, can homogeneous reconfiguration be solved any faster? What is the lower bound? If space is indeed the critical issue, then is an in-place heterogeneous solution possible in polynomial time? Finally, if this is so, what is the minimum amount of free space required by the in-place solution? We are currently investigating these questions.

Practically speaking, it is our hope that these algorithmic results provide the basis for more advanced applications of SR robots. Immediately, it should be possible to add different sensors to existing systems without loss of reconfigurability. It is also interesting to consider multiple modes of actuation, such as unactuated battery modules, or modules such as wheels to increase locomotion capabilities. We would also like to develop algorithms that support more types of heterogeneity, such as varying module geometry or connection constraints. We are now developing these types of applications, and also continuing our algorithmic work as outlined.

Acknowledgments

Support for this work was provided through NSF awards IRI-9714332, EIA-9901589, IIS-9818299, IIS-9912193, EIA-0202789 and 0225446, and ONR award N00014-01-1-0675.

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*, 2001.

[2] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for a class of self-reconfigurable robots. In *Proc of IEEE ICRA*, 2002.

[3] A. Castano and P. Will. A polymorphic robot team. In T. Balch and L. Parker, editors, *Robot Teams: From Diversity to Polymorphism*. A K Peters, Ltd., 2002.

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

[5] J. Hopcroft, J. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; $pspace-$hardness of the "warehouseman's problem". *The International Journal of Robotics Research*, 3(4):76–88, 1984.

[6] K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H. Asama, Y. Koruda, and I. Endo. Self-organizing collective robots with morphogenesis in a vertical plane. In *Proc. of IEEE ICRA*, pages 2858–63, 1998.

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

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

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

[10] 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.*, 1996.

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

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

[13] R. Sharma and Y. Aloimonos. Coordinated motion planning: The warehousman's problem with constraints on free space. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(1):130–141, 1992.

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

[15] Cem Ünsal and Pradeep Khosla. Mechatronic design of a modular self-reconfiguring robotic system. In *Proc. of IEEE ICRA*, pages 1742–7, 2000.

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

[17] M. Yim. New locomotion gaits. In *Proc. of IEEE ICRA*, pages 2508–2514, 1994.

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

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