Generic Decentralized Control for Lattice-Based Self-Reconfigurable Robots

Zack Butler, Keith Kotay, Daniela Rus and Kohji Tomita

Abstract

Previous work on self-reconfiguring modular robots has concentrated primarily on designing hardware and developing reconfiguration algorithms tied to specific hardware systems. In this paper, we introduce a generic model for lattice-based self-reconfigurable robots and present several generic locomotion algorithms that use this model. The algorithms presented here are inspired by cellular automata, using geometric rules to control module actions. The actuation model used is a general one, assuming only that modules can generally move over the surface of a group of modules. These algorithms can then be instantiated on to a variety of particular systems. Correctness proofs of many of the rule sets are also given for the generic geometry; this analysis can carry over to the instantiated algorithms to provide different systems with correct locomotion algorithms. We also present techniques for automated analysis that can be used for algorithms that are too complex to be easily analyzed by hand.

I. Introduction

Self-reconfiguring robots are systems constructed of large numbers of independent modules that have the potential to be extremely versatile in the tasks they can perform. For example, a self-reconfiguring robot could conform to the shape of the terrain for locomotion by implementing “water-flow” like locomotion gaits which allow the robots to move by conforming to the shape of the terrain. It could assemble into a tall tower to enter a building through a window, or change into a snake-like configuration to enter through a small hole. Once inside, the modules could distribute themselves to form a sensor network for surveillance and monitoring. Since a system will consist of large numbers of modules, distributed control is essential, both for robustness and scalability.

Many different self-reconfiguring robots have been built, with a wide variety of actuation capabilities. Because of this difference between systems, most reconfiguration algorithms have been coupled to particular systems. In order to get general insights into the computation complexity and range of application for self-reconfiguring robots, we are interested in developing algorithms that can apply to a class of systems. In particular, we wish to develop algorithms that can apply to any lattice-based system. A lattice-based self-reconfiguring robots is one in which the modules fill space in a lattice formation. These algorithms can then be specified and analyzed at the higher level, where they are generally much simpler. In addition, one high-level algorithm can be used to generate reconfiguration for different systems.

The shape-changing problem can be divided into two main tasks: static reconfiguration and dynamic reconfiguration. In static reconfiguration, the goal is for the system to achieve a given shape at its current location. This is used for building structures such as scaffolds, sensor arrays, furniture, etc. In dynamic reconfiguration, the goal is for the system to use its shape-changing ability to move through the environment. In general, when undergoing dynamic reconfiguration, the exact shape of the system is not relevant. Specifying a global shape based on purely local information can be challenging and require significant communication within the system.

We observe that in most existing lattice-based self-reconfiguring robot systems (e.g. [10], [11], [16], [26]), an individual module can move in general ways relative to a structure of modules by traveling on the surface of the structure. Specifically, an individual module is capable of: (1) linear motion on a plane of modules; (2) convex transitions into a different plane; and (3) concave transitions into a different plane. The details for how to accomplish these motions are architecture-dependent, and may require low-level feedback control, which we do not consider in this paper. In this paper we introduce a class of distributed control algorithms based on these motion abstractions. These algorithms are generic, in that they can be instantiated on any self-reconfiguring robot that can perform these general motions and maintain the connections inherent in the generic model, as described in Sec. II.

The algorithms presented draw on the concept that dynamic reconfiguration can be naturally specified with simple local motion rules that allow the global shape to remain unspecified. These rules can be based only on the local configuration around a given module. This has led us to develop locomotion algorithms inspired by cellular automata, which also make use of local rules to make
decisions. These algorithms are then inherently distributed in that each module makes its own decisions based solely on local information. The cellular automata model also supports the use of an abstract model of a self-reconfiguring module, as detailed in Sec. II. In this way, the algorithms can be specified in a simple framework but instantiated to particular systems.

In this paper, we present several distributed algorithms based on local rules which together produce versatile “water-flow” style locomotion over the plane and over large obstacles. That is, the robot will move in a way that conforms to the obstacles much like water conforms to a streambed. Examples of such locomotion are shown in Fig. 1. Specifically, we show algorithms for straight-line locomotion on flat ground, over obstacles, through tunnels, and algorithms for turning a group both on flat ground and on obstacles. We also present two correctness analyses for these rule sets. The first analysis approach involves examining the structure and geometry of the rules to argue correctness, while the second approach is an automated technique. Finally we present instantiation of these algorithms on to several different hardware systems. These instantiations include one implementation in hardware (namely the Crystal Robot) and several simulated instantiations. Together these indicate that with proper sensing and control, these algorithms can be applied to a variety of real self-reconfiguring robots.

A. Related Work

The field of self-reconfiguring robotics has seen the development of many interesting hardware systems, both lattice-based and chain-based. Chain-based systems, in which the modules form connected linear structures, are well suited to locomotion over low obstacles and through tunnels, but are less suited to building tall shapes. Examples of chain-based systems include the Polybot [27] and CONRO [7] systems. Lattice-based systems are made of modules that fill space and are best suited to building a wide variety of shapes as well as climbing tall obstacles, and can also perform locomotion over flat ground. Planar lattice-based systems include the planar Fracta [14] system from AIST, in which modules are hexagonal and move over the surface of a hexagonal lattice, the hexagonal metamorphic robots of Chirikjian et al. [16], which have similar overall geometry but actuate through deformation of the modules, and a system by Hosokawa et al. [10] in which modules with square cross-section move in a vertical plane. Among three-dimensional systems, the actuation modes are quite varied. The Proteo system [28] uses modules that are rhombic dodecahedra which pack tightly and roll over each other. The Molecule robot [11] uses modules that incorporate two sub-modules called atoms joined by a fixed bond. Each module has four degrees of freedom and can locally move in three dimensions over the surface of the group. The M-TRAN system from AIST [13] also uses modules that consist of two connecting units joined by a bond. Each M-TRAN module can move only locally in a plane but the overall system can generate arbitrary 3-D shapes, and can also create chain-based structures. The Crystal [3] from Dartmouth and Telecube [21] from PARC are examples of unit-compressible systems, in which the modules form a cubic lattice but actuate internally to the group through expansion and compression rather than over the surface of the group.

Reconfiguration algorithms have been developed for all of these hardware systems, predominately centralized, but several distributed control algorithms have been proposed. Lattice-based systems have concentrated on static reconfiguration, and distributed algorithms have been developed for several systems, including the Proteo [28], Fracta [14], [22] and unit-compressible systems such as the Crystal and Telecube [6], [23]. Walter, Welch and Amato [25] presented a distributed reconfiguration algorithm (extended by Walter, Tsai and Amato [24]) for planar hexagonal modules. Hosokawa et al. have developed a distributed control algorithm [10] for a particular planar hardware system that has the same actuation primitives as our abstract model, however their algorithm is designed.
to produce only a small collection of shapes. For chain-based robots, distributed algorithms have been proposed for reconfiguration and locomotion. Salemi, Shen and Will developed the idea of digital hormones [19] which are propagated through a modular robot to control shape change and locomotion. A method for distributed control for locomotion of chain-based robots was developed by Stay et al [20]. Our work focuses on dynamic reconfiguration of lattice-based systems, which has not been studied previously.

II. Algorithmic Framework

We represent a generic self-reconfiguring robot as a collection of cells, each cell corresponding to one robot module. We view the resulting structure as a particular type of cellular automata. This enables the construction and analysis of algorithms that can support a variety of hardware instantiations. In particular, any system that can instantiate the generic model as defined below can use the algorithms presented in this paper.

Each algorithm presented here is simply a set of local rules that is run by each cell within the system. Each rule is defined by a set of preconditions on the neighborhood of the cell and a set of actions that are executed if the preconditions are satisfied. We represent the basic module of the robot as a cube, but our proposed abstraction can be replaced by other geometric structures that support the formation of lattices. Each module is assumed to be able to translate across the surface of other modules as well as transition to other planes of motion. It is also assumed to be able to examine or query its neighbors and determine the presence or absence of a neighbor on all sides. Preconditions may include specific geometric configurations of neighbors and empty space, or given values of a module’s internal state. The actions of a rule can include motion of the cell, an update of an internal state, or both.

A. Evaluation Models

Each set of rules is designed in a task-directed way (either manually or automatically using learning). Each module uses the same set of rules and independently evaluates the rules and carries out the specified action resulting from a successful rule. In traditional cellular automata, all cells are evaluated together, such that the next configuration is generated from the current configuration by simultaneously evaluating all the cells. This model is not concerned with physical aspects of the underlying system. Since our robots are physical systems, we modify the evaluation model for traditional cellular automata to a sequential cell evaluation process, in which only one cell is evaluated at a time.\footnote{We can easily ensure no collisions in simulation by simply evaluating the cells one at a time, while in hardware a local locking system is required (note that cells at non-interfering locations in the robot can activate simultaneously).}

We define three evaluation models, based on the relative delay of the activation of the cells. In the simplest model, which we call $D_0$, the cells are evaluated in a set cyclical order, and so there is no variation in the relative delay between any two cells. For the majority of the rule sets presented here, we use an activation model referred to as $D_1$. Under the $D_1$ model, a cell can delay activation at most one cycle relative to any other cell in the system, or equivalently, one cell can activate at most twice before another cell activates once. This model relies on the homogeneity of the modules of the robot, which ensures that they run at approximately the same speed. The $D_1$ can be implemented in simulation using the technique shown in Algorithm 1, although in a real robot the evaluations would be timed implicitly by the system. Our most flexible model is called $D_\infty$, in which a cell may delay for an arbitrary amount between activations (although infinite delay may halt progress of the system). This can be implemented using Algorithm 2. In Sec. IV we present two rule sets that will work correctly under this less restrictive condition. This model requires more complex rule sets and analysis than the $D_1$ model, but allows for more variability of the individual cells, making it more robust for instantiation onto a distributed system. Even if software synchronization is possible, it may represent a significant communication cost, so the asynchronous algorithms may be preferable for systems where the modules have variable speed to avoid this overhead.

III. Planar $D_1$ Locomotion

To perform locomotion with a self-reconfiguring system, we would like to implement a “waterflow” motion, in which the modules conform to the shape of the ground as the group moves forward. Within the group, each module in turn will move from the back of the group, over the top of the group, and settle at the front either on the ground or on another module. This looks globally like
Algorithm 1 Instantiation of $D_1$

1: while (1) do
2:   flag = 0 in all cells
3:   while (Any flag == 0) do
4:     Choose unflagged cell at random
5:     Set flag = 1 for current cell
6:   while (Not all rules tried) do
7:     Evaluate random rule for current cell
8:     If rule applies, use it and break

Algorithm 2 Instantiation of $D_{\infty}$

1: while (1) do
2:   Choose cell at random
3:   while (Not all rules tried) do
4:     Evaluate random rule for current cell
5:     If rule applies, use it and break

a tank tread, in which only the modules on the top of the perimeter of the group move to produce collective motion. Within this framework, there are several different possible implementations. Here we will first discuss and present analysis for two implementations under the $D_1$ activation model. In Sec. IV we present two rule sets for locomotion under the less restrictive $D_{\infty}$ model, and in Sec. V we present extensions climbing and tunneling as well as turning of 3D groups, to allow complete exploration of the plane.

A. Locomotion without obstacles

In order to give a high-level picture of the cellular automata approach for controlling self-reconfiguring robots, we start with the simplest locomotion algorithm: the robot moves on the floor and there are no obstacles. The rule set for this task represents a simple introduction to this type of algorithm. To cause a group of modules (i.e. a connected set of modules) to walk forward over flat ground, a set of only five rules are sufficient. The rules shown in Fig. 2 will move a robot (represented as a rectangular array of cells) eastward, where eastward motion is to the right. The rules are shown as productions, in which the left side represents the preconditions which must exist in the environment of the cell being processed, the “current cell”, represented in this figure by a black square with a white dot. The right side of the rule represents the environment of the current cell after the rule has been applied. The rules in Fig. 2 require three different types of existence tests: whether a cell exists at a specified relative grid location, whether a cell does not exist at a location, and whether a location is empty. For example, Rule 5 in Fig. 2 can be applied only if there is an empty location below the current cell, and cells to the west and southwest. If Rule 5 is applied, then the result is that the current cell moves one unit in the southward direction. We present a simulated implementation of this rule set in Extension 1.

A.1 Analysis

The rule set of Fig. 2 will provably generate eastward locomotion when evaluated under the $D_1$ model on a rectangular collection of cells. There are two ways in which we can prove this. To prove

![Fig. 2. Five rules for eastward locomotion without obstacles, with the direction of motion given for each.](image-url)
This, it is sufficient to show that 1) some rule from the rule set can always be applied to the cell array, 2) eastward motion must result from all possible sequences of rule activations, and 3) the array remains connected, i.e., the cell array always consists of one simply connected component based only on face-to-face connections. We present these statements as lemmas and give the outline of the proofs here, and refer the reader interested in the proof details to [4]. Lemmas 1, 2, and 3 present the proofs we have developed as a human-centered analysis. In Sec. VII we describe an automated proof that takes advantage of the regular structure of these systems.

**Lemma 1:** Given an initially rectangular cell array, in the absence of disconnections some rule from the rule set shown in Fig. 2 can always be applied.

To show that there is always a rule that can be applied, we look at the set of actions that can be taken from the initial configuration of a rectangle. In the western (rearmost) column of the group, the topmost cell will move northeast by rule 1, making way for the other cells in that column to move north. No cell in any other column can initiate motion, so the first cell to move will be able to make its way to the front of the group without interference. Subsequent motions will always be available at the back of the group, as each cell will be able to move over the front of the group. A cell moving over the top of the group may be temporarily delayed by a cell in front of it, but eventually will be able to move forward toward the front of the group and allow the cell behind it to move as well.

**Lemma 2:** Given an initially rectangular cell array, eastward motion must result from all possible sequences of continuous rule activations from the rule set of Fig. 2.

Since by Lemma 1 some rule can always be applied, the cell array will move eastward as long as there are no oscillations between complementary rules. Since Rules 2-4 have an eastward component of motion and there is no rule that produces westward motion, only Rules 1 and 5 are complementary, i.e., have offsetting motions. But Rule 1 can only activate on the western edge of the cell array and Rule 5 can only activate on the eastern edge of the cell array. Therefore no oscillation can occur, and the cell array must move eastward.

**Lemma 3:** Given an initially rectangular cell array, no possible sequence of rule activations from the rule set shown in Fig. 2 can disconnect the cell array.

By examination of the rule set it can be seen that no rule can cause the cell executing the rule to become disconnected from the cell array, because every rule contains as a prerequisite the existence of a post-motion connecting cell. Thus, a disconnection can only occur when a cell is connected to only one other and the other cell moves away. This requires the interaction between two cells on the surface of the group, and the only possible location for this configuration is at the northeast corner of the cell array. We can show that under the \( D_1 \) activation model, no cell will move into a configuration in which it is only adjacent to a cell that will move away.

**Theorem 4:** The rule set in Fig. 2 produces eastward locomotion on any initially rectangular array of cells.

**Proof:** Lemmas 1, 2, and 3 prove that some rule from the rule set can always be applied to the cell array, eastward motion must result from all possible sequences of rule activations, and the array remains connected. Therefore, the rule set produces eastward locomotion on a rectangular array of cells.

**B. Locomotion over low obstacles**

The ability to perform locomotion over perfectly flat ground has limited application. However, we can easily augment the rule set in Fig. 2 to allow the cell array to crawl over obstacles. The new rule set, Fig. 4, has eight rules. The three new rules allow for southwest and northwest cell movement.
which is needed for the cells to comply to the obstacle field (two rules are used to implement the southwest movement). Preconditions that involve the presence of obstacles are also added as shown. The basic idea is still the same, in that a cell moves upward along the west side of the group, across the top, and down the east side, however in the presence of obstacles the exact path is slightly modified. In our simulation, obstacles are represented as cubes, but cells do not connect to them, so algorithmically these rules can be used over irregular obstacles given proper sensing — partially filled cell locations would be considered obstacles in the rules.

Figure 3 shows four snapshots from a simulation run of the rule set of Fig. 4. For this rule set, the maximum obstacle height is one less than the height of the cell array. In this case, the robot is composed of five separate layers, each running the planar rule set. The observed motion of the robot is very compliant to the terrain and appears to flow over the obstacles. We also note that since the rules are planar in nature, each layer can execute the rule set as if in isolation, but the layers may separate if one moves significantly faster than another due to the presence of varying obstacles. To mitigate this, we add an additional condition to the NE motion rule (Rule 2 of Fig. 4) that requires neighboring layers to be caught up with the current layer. This ensures that the group will remain connected in the third dimension, and this condition is used in all of our planar rule sets to extend them to 3-D groups.

B.1 Analysis

To prove the correctness of this rule set, we begin with the correctness of the five rule set as shown in Sec. III-A.1. First, note that in the absence of obstacles, the current rule set reduces to the five rule set: the NW rule (8) can never apply without obstacles present, and the SW rules (6, 7) will not, since in the absence of obstacles columns are filled one at a time, so there could not be an empty space to the west of a cell on the east face. We now explore the situations in which obstacles are encountered, and show that for each, progress will continue without deadlock or disconnection.

Theorem 5: The rule set of Fig. 4 will produce eastward locomotion for any (initially rectangular) group of cells over any obstacle field that is (a) shorter everywhere than the initial height of the group, and (b) everywhere supported from directly below.

Proof: The group of cells will locomote as described by the five-rule set until the first obstacle is reached. At this point, the cells moving to the east edge of the group will settle on this obstacle using Rule 5 just as they would settle on other cells in the no-obstacle case. We now show that the cells that will make up the next column (assuming the obstacle is shorter in the next column) will arrive there without disconnecting the group.

Once the column on top of the obstacle is completed, the next cell will execute Rule 4 followed by Rule 5 until it reaches the same height as the lowest cell in the previous column. If the obstacle is of the maximum height, this cell will simply extend the top row, with empty space above and below it, as shown as cell \( F \) in Fig. 5a. We must now show that the following cells will continue to their correct destinations and not disconnect. We show this using the notation of Fig. 5a to describe positions in the shape.

Because of the \( D_1 \) evaluation model, the cell at location \( C \) will go to location \( G \) before \( A \) can move. The cell at \( A \) can therefore get only as far as \( C \) before \( G \) moves to \( H \) (using Rule 7). Note that during this process, \( F \) cannot move. This leads to the configuration of Fig. 5b. The cell that
was at A and is now at location C will go to G and then I before another cell can get on top of it (as shown in Fig. 5c), before using Rule 6 to move to J. Additional cells will follow using a similar set of motions to complete the new column.

Now we must show that the cells beginning motion at the back of the group will be able to actuate in the correct order to produce motion without disconnections (and while maintaining the general arrangement of cells across the top to ensure no deadlocks).

First we note that due to the nature of Rules 5-7, there will never be a cell with obstacles both on its immediate left and right, so there will always be at least two cells in any row between two obstacles. We examine two cases based on the relative heights of the westernmost two columns: first, where the obstacle in the westernmost column is taller than (or the same height as) that of the second column, and second, where the obstacle in the westernmost column is shorter. In this latter case, there are some cells at the bottom of the west column that have an obstacle to their east (e.g. H, J, L in Fig. 6a).

In the first case, none of the cells in the western column will have obstacles to their east, and so Rule 8 will never apply. This is the same as in the five-rule case, and the argument there follows. In the second case, the cell(s) with obstacles to their east will move NW using Rule 8 once the opportunity presents itself.

The second column before the obstacle will actuate from the top down as in the no-obstacle case, eventually reaching a state similar to that of Fig. 6a. Note that as the cells in this column move upward, the cell at L cannot move NW. Only once the last cell in the column (cell K) moves to D (as in Fig. 6b), cell L can move to I. K may be as far as B by this point, but not further. If K is at location B, there are two spaces between the cells (see Fig. 6c), which can be transiently increased to at most three as they move over the surface of the group. But since Rule 2 requires four consecutive empty spaces around the perimeter, E cannot move until all cells below that have an adjacent obstacle have moved past it. In addition, the only rules that could apply to cells H and J (which have a cell to their north and an obstacle to the east) are Rules 5-6, both of which require two adjacent cells at the bottom of the column to the east. This cannot happen since by the time one cell moves NW, the spots above and below it are guaranteed to be empty.

\[\text{III. Locomotion with Asynchrony}\]

In this section, we discuss two locomotion algorithms that operate under the $D_1$ activation model. The first is a simple algorithm that walks over flat ground, corresponding to the $D_1$ algorithm of Sec. III-A. The latter is a larger rule set that was designed for easiest instantiation to real systems, in that it not only allows for asynchronous execution but also only uses preconditions that fall within the current cell’s immediate neighborhood.

A. Asynchronous locomotion without obstacles

The analysis of the previous rule sets indicates that the $D_1$ activation model is required for their correct behavior, as it enforces proper separation between moving cells around the surface of the group. In the absence of this constraint, we need to find another way to ensure correct behavior. We have chosen to add a binary state to each cell indicating whether it is “active” or not. The idea
Fig. 7. Eleven rules for eastward locomotion without obstacles for the $D_{\infty}$ activation model, with the direction of motion given for each. Cell variable “A” is the moving state variable which can assume the values \{O, X\}, where O is moving and X is not moving. If a cell has no indicated value for a variable it means that the value of that variable was not considered as a precondition for the rule.

Fig. 8. Snapshots of two different simulations based on the $D_{\infty}$ climbing rule set.

is that a cell will become active when it first moves away from the back of the group, and remains active until it reaches its resting position at the front of the group. Then, as long as active cells do not interfere with each other, they can maintain the correct structure. We have developed a simple rule set for the $D_{\infty}$ case corresponding to the rules for walking over flat ground under the $D_1$ model as shown in Fig. 2. This extended rule set contains 11 rules, including some which only change the state of the cell (i.e. do not produce motion). In addition, some preconditions involve checking the state of neighboring cells to see if they are active. This rule set is shown in Fig. 7.

B. Climbing

The second $D_{\infty}$ rule set uses a larger number of simpler rules, and can allow the group climb over obstacles significantly higher than the initial height of the group. This makes it more capable than any of the rule sets presented so far. It is composed of 22 rules, each of which uses only the Moore neighborhood (eight adjacent cells). The concept of active and inactive cells is also used here, and the rules are explicitly classified into three types: five activation rules, 12 moving rules for activated units, and five inactivation rules. Two examples of simulated sequences using these rules are shown in Fig. 8. Gray, dark gray, and black squares represent an obstacle, an inactivated unit, and an activated unit, respectively. Correctness of this rule set can be shown through the use of a loop invariant that represents all valid configurations. We can prove that all cell motions that start in a valid configuration lead to another valid configuration.
V. LOCOMOTION EXTENSIONS

A. Climbing and Tunneling

The algorithm presented in the previous section can climb effectively over tall obstacles. In this section, we present a rule set based on the rules of Fig. 4 which extends climbing to obstacles taller than the group and can also be extended to tunneling. The key difference is that in a tunnel, the robot is constrained both from above and from below. For situations such as search and rescue operations, navigation in enclosed spaces is a critical function. For this reason, we have also investigated rules that allow a group to automatically shrink their height in order to tunnel through a small hole.

As in the previous algorithms, this rule set uses the notion of active and inactive cells, however it operates under the $D_1$ activation model. The new rule set is essentially a superset of the eight rule set of Fig. 4, so it is presented as additions and modifications to those rules. We first present rules that extend the $D_1$ algorithm to allow climbing. These rules are shown graphically in Fig. 9. The rules L1-L3 are replacements for the first three rules of Fig. 4 which allow a cell to move northward more than once in order to build a taller group. The cell becomes active upon the use of any locomotion rule, and rules D1-D4 will deactivate the cell when it reaches the front of the group.

The global property of the group under this rule set is that the group height will remain constant as under the original rule set until a tall obstacle is encountered. At that point, the group will build additional complete rows one at a time until it is one unit taller than the obstacle, as outlined in Fig. 10. Cells can then move over the top of the obstacle and build down the opposite side. The rules presented here then maintain the taller group size after the obstacle, although there are methods (including the use of the tunneling rules described below) to reduce the height of the group for better stability. Like the locomotion rules, the climbing rule set also extends to 3-D groups with the same preconditions on the NE rule. These rules are included in a simulation presented as Extension 1.

To enable tunneling, we extend the concept of active and inactive units so that cells in front of a small hole do not become inactive. This allows the group to decrease its overall height and enter the tunnel, while allowing these rules to coexist with the climbing rules in a single rule set. This coexistence allows the group to climb within a tunnel, which is important since the tunnel will rarely be flat. The height of the group generally remains constant, but will decrease to pass under overhanging obstacles and increase when necessary for climbing. Using these rules, the system can traverse any flat tunnel that is at least two modules high and any tunnel that is at least three modules high everywhere.

Only three new rules are required to enable tunneling, as shown in Fig. 11. The first two, T1 and T2, specify how an active cell moves under the front of an obstacle, lowering the height of the group if necessary. These rules allow the behavior seen in Fig. 12a-b. The third new rule enables climbing...
within a tunnel, as performed in Fig. 12d-e. We also have to make sure that cells do not always deactivate in front of obstacles, since there may be holes under the obstacle, and so we replace one of the deactivation rules (D3 of Fig. 9) with a pair of more specific rules, also shown in Fig. 11. The overall behavior under this rule set is that when a cell is moving down the front face of the group and notices that the wall in front of it has a hole at the bottom, it will not stop in front of the wall, but will move forward and start a new short column under the obstacle.

B. Turning

In order for a 3-D group of modules to achieve complete locomotion over the plane, it is necessary for the group of modules to change direction. Since the modules are in a square lattice, they will only turn locally in multiples of 90°, but this is sufficient for full coverage of the plane. We have developed rule sets that enable turning both on flat ground as well as over most obstacles.

The basic set of turning rules, first described in [5], allows turning on any surface that is flat under the footprint of the group. Under these rules, upon recognition that a turn is required, either due to sensory data or in response to a command from a higher level controller, the group will stop and reform into a rectangular prism. The modules will then select the new direction of travel and restart locomotion. Since the group is in a prismatic shape, it is equally capable of moving in any direction from that point, and the turn is easily accomplished. The locomotion rules are specified as being relative to the direction of travel for the group, so after the direction change, the same rules are used but are evaluated in a different orientation. These rules are also included together with the climbing rules in the simulation presented in Extension 1.

These rules make use of a new binary state, halt, which allows the group to recognize that it is time to stop locomotion. This state is first set by a module moving to the front of the group and is propagated through the system — any module that sees a halted module in front of it will also halt. The halt prevents the start of the back-to-front locomotion process by inhibiting inactive cells from becoming active. Active modules continue their motion, so that a rectangular prism is once again formed. Once a module is halted, it can discern from its neighborhood whether locomotion has completed. Modules on top simply wait for there to be no active modules in their neighborhood, while modules further inside the group wait for the module above them to clear its halt state. When a module turns off its halt state, it also selects a new travel direction. Since the group is prismatic, only the modules at the back can restart locomotion, just as when locomotion first begins, and the group will move in the new direction.
The turning task can be implemented with a total of just four new rules. Two of these rules describe how the halt signal is propagated back through the group. The other two detect the completion of locomotion by noticing a lack of moving modules in the neighborhood or by noticing that adjacent modules have already turned. The latter two rules also set the new direction, currently hard-coded as a left turn (a hard-coded right turn is also simple), but a higher-level instruction to turn one way or the other could be implemented. In addition, there must be some way to trigger the initial halting of the group. Currently, this is done through either the presence of a tall obstacle in front of the group or an external scripted directive.

B.1 Turning over obstacles

The previous algorithm can also produce turning on non-flat areas, but will almost always give unsuccessful results. That is, the group will always change direction, but since it will not form a prism first, it will leave some modules behind, or some planes in the new direction may not have sufficient modules for correct locomotion. However, we have developed a second set of locomotion rules which maintains a sheared prism at all times, and therefore allows turning on nearly any surface. As with any increase in capability, this new rule set requires additional state (in this case, knowledge of the initial height of the group), but it is still a purely local algorithm and operates with any group size. As with the previous rule sets, it is only guaranteed to work for groups that initially form a rectangular prism, although it may work for a variety of other initial configurations.

The concept behind this rule set is that rather than the group remaining the same absolute height as it goes over obstacles, each column of the group will keep the same relative height. That is, each stationary column of the group will contain the same number of modules. This property can be seen in 2-D simulation snapshots in Fig. 13. The group is therefore able to stop locomotion on top of an obstacle field and be shaped as a sheared prism. When the group is in this shape, it can change direction as in the flat ground case described above, since each plane will consist of a sheared rectangle of cells.

To achieve this, we use a rule set similar to the climbing rules presented in Sec. V-A, with the addition that each cell maintains a height state along with its active state. When a cell reaches the front of the group and deactivates, it will determine its height based on the cell below it, simply by adding one to its neighbor's height. If there is no cell below it, it will set its height to 1 or 2, depending on the local neighborhood. The rules that result in motion are also generalized from the climbing rules, since an active module may need to move up or down while it moves over the top of the group. The rules that activate a cell at the back of the group are the same, but once a cell is active, there are less restrictive rules that force it to move forward over the uneven group. The cell deactivates when it reaches the front of the group, either as in the climbing rules, or when it moves onto a column that is not yet of full height. This requires a total of 18 rules. This rule set can walk over any terrain that the climbing rule set can traverse, although as described below, it may not be able to produce correct turning over all terrain.

To turn under this rule set, the group will behave similarly to the above case on flat ground. The group will still halt and turn when a module at the front of the group detects a tall cliff or receives a halt instruction. The halt signal is propagated back here as well, although due to the shape of the group, it must be propagated up and down as well as straight back. As before, active modules will continue moving so that the frontmost columns of the group are completed. Whenever a group halts, each column will be the same height as the original group. Since each column is the same height, and the number of modules is the same as the initial group, there will be no extra modules, and a sheared rectangle will result. The only exception to this is if the group is traveling over a steep downhill. In this case the modules cannot produce a column of the correct height without looking arbitrarily far down, and so this situation may not produce correct turning. Otherwise, the group can be conceptually turned 90° and still have an equal number of modules in each plane. The
turning rules that detect the absence of active modules in the neighborhood must be generalized to handle the non-flat surface of the group, but are conceptually the same as above. The action of the turning rules on non-flat ground is shown in Fig. 14.

Once the modules have selected their new direction, they can resume locomotion. However, the sheared rectangle that they are in is not necessarily a valid starting point for the locomotion rules (although it is a valid intermediate point). To correct for this, an additional boolean state is set by each module indicating that the group has recently turned. A module with this state set can only move if it is in the back-most column, and once locomotion resumes, the state is cleared.

In order to allow successful turning on the widest variety of terrain, we make one further observation. When a group of modules is moving forward over obstacles, one layer may travel through a narrow crevasse. Consider the case where the group decides to turn while over this crevasse. Since the crevasse is only one module wide, the new planes of locomotion (perpendicular to the previous planes) will contain a single column of modules isolated in the crevasse. These isolated modules cannot come out without disconnecting the group. Therefore, we want to have the modules avoid entering such a crevasse in the first place. We do this by treating any area that is bounded immediately by obstacles on both left and right sides as itself an obstacle. This allows the locomotion to continue correctly and allow correct continuing locomotion after a turn.

VI. Instantiations

A key feature of our abstract locomotion algorithms is that they can be instantiated and realized on a variety of self-reconfiguring robot systems. This primarily requires developing actuation primitives for the system that can realize the types of motion called for by the cellular automata. In this section we discuss several different instantiations. Most of these instantiations are in simulation, however we have also implemented the control on one hardware system. It should also be noted that there exist 2-D self-reconfiguring systems, those of Hosokawa et al. [10] and Chiang and Chirikjian [8] that have the exact geometry and motion capability of our abstract model.

A. Meta-modules

The simplest way in which these automata may be instantiated onto several different systems is through the use of meta-modules. For many systems, it is possible to construct a group of several non-isotropic modules in such a way that the group has isotropic motion capability of the type required by the algorithms presented here (motion in a unit lattice over the surface, convex and concave transitions). For the Molecule, a structure called a tile has been developed with this property [12], while the grain meta-module was developed for the Crystalline Atom [17]. These meta-modules can then immediately use the rules of the automata, although communication between the individual modules would be necessary to determine the meta-module’s neighbors and coordinate its actuation. The use of meta-modules requires many more modules to achieve the same amount of reconfigurability. Therefore, instantiations of the rules on to individual modules may be preferable.

B. Molecule meta-modules

The Molecule robot is a 3-D module developed at Dartmouth [11], in which each module consists of two atoms connected by a fixed bond, with a total of four degrees of freedom per module not including the mechanical connectors. Individual Molecule modules are shown in Fig.15. It is unclear whether cellular automata algorithms can be instantiated to single Molecules, since the Molecule
shape is very different than the abstract cellular automata module. However, it is possible to instantiate cellular automata algorithms onto Molecule meta-modules, which can be constructed as convex shapes. In particular, we have developed instantiations in which just two Molecules can be used in place of one abstract module.

This two-module group can be configured to occupy a rectangular shape of size $1 \times 2 \times 3$ atom diameters, and therefore is capable of emulating the cubical abstract module. Individual Molecules are capable of performing the isotropic motion primitives, and so the pair can also perform them by utilizing sequential Molecule relocations. However, free space is an issue for pair movement, as Molecule pairs require extra free space for movement in some directions. For the 5-rule set in Fig. 2, extra free space is required for the north and south moves, since the Molecules must climb on each other to move. It is possible to amend the 5-rule set to add additional free space for the north and south rules. A simulation of this instantiation is shown in Fig. 16. In the initial configuration (upper left), fifteen pairs of Molecules are arranged in a $3 \times 5$ rectangle. The modules that make up each pair are side-by-side in the dimension into the page. In the remaining snapshots, these pairs move over the top as directed by the 5-rule set.

C. Hexagonal lattice systems

The 5 rule set can be instantiated to systems with hexagonal modules with minor modification. Systems of this nature include the Fracta system [14] and the metamorphic robot of Chirikjian et al [16]. The adapted set of rules are shown in Fig. 17. We assume the $D_1$ model and no obstacles as for the original rule set. Because the lattice configuration is different, we assume that the initial configuration is a trapezoid instead of a rectangle, with a top row of at least four units.
Fig. 19. One module of the system of [15].

Fig. 20. Eight snapshots of a simulation for the instantiation of the rule set of Fig. 2 to the M-TRAN system.

Directions of tilt are shown in the figure around each cell as a parallelogram. Rules applied to the left part of the group (north and northeast rules) are tilted right, and the rules for the right part (southeast and south rules) are tilted left. The east rule combines them, resulting in an additional cell condition. A simulated sequence is shown in Fig. 18. Units are shown in black and the floor in gray. The figure shows eastward locomotion of the group.

The correctness proof is similar to the proof in Sec. III-A. One difference is that the northeast rule makes a unit move one step further from the point of view of the left tilt. When another moving unit is still on the same height of the top row, the unit might move eastward over this unit, which does not happen using the original rule. This case happens when the top row contains only three units. By restricting robots to have at least four modules on the top row, we can ensure eastward progress.

**D. Modular Transformer (M-TRAN)**

Another system that we have investigated the M-TRAN robot [15]. This system is quite powerful in that it can operate either as a lattice-based system or a chain-based system. The individual modules are somewhat unusual, as can be seen in Fig. 19. Each module consists of two semi-cylindrical components connected by a rigid bond, and has two degrees of freedom. A module in this system cannot connect to neighbors on all sides, as is assumed in our abstract geometric model. Therefore, at least two 2-D layers must be present to maintain connections between all modules, and local connections do not necessarily imply that the group as a whole is connected. In addition, since each module consists of two articulated bodies, it inhabits a configuration space larger than the two dimensions of the simple automata described here.

Despite these complications, it is possible to instantiate locomotion algorithms to this system. We have done this with the five rule set of Sec. III-A. Neighbor relationships translate in a straightforward way, and movement of a module along a straight line is also easy. However, the motion of each module does not translate directly from the abstract model to the M-TRAN. First of all, a single module cannot perform the convex transition motion primitive. Rather than employing meta-modules, we have added the concept of a helper, so that a module that needs to perform a northeast or southeast move can ask its neighbor for assistance. The request for assistance is done with an explicit message rather than through the rules. Also, while each rule produces a specific direction of motion in the abstract geometry (as in Fig. 2), that motion may be implemented with different physical motions depending on the module’s configuration. The instantiated rules take this into account to produce the correct low-level actuation. Finally, as mentioned above, we must also ensure that the group remains connected despite the limited connection ability of the individual modules. This is also done with message passing, whereby a module with two connections that wishes to disconnect one queries its neighbors to determine if the disconnection is acceptable.

It should be noted that this algorithm uses only the lattice-based mode of M-TRAN, which is a subset of its overall capabilities. However, it is a simple way to produce forward locomotion for an arbitrary number of units, and may be useful in this regard. A simulation has been implemented as the instantiation of these rules in which each module is simulated by a different process. This ensures that each module knows only about its immediate neighborhood, and can query its neighbors to determine their configuration. This implementation has produced successful forward locomotion. The results of this simulator can be displayed in the GUI developed for these modules at AIST, as shown in the screenshot in Fig. 20.
E. Crystal

All of the previously mentioned systems consist of modules that reconfigure by moving over the surface of the group. The Crystal robot is one in which the modules are instead unit-compressible, meaning that they actuate by expansion and contraction. Two modules are shown in Fig. 21. The modules then reconfigure by moving through the volume of the group. The locomotion algorithms therefore require adaptation to this different actuation technique. Each rule still evaluates the presence and absence of local neighbors, as well as the state of the current cell and its neighbors, and causes actuation and/or an internal state change. In this case, the state may be a physical one, i.e. whether the module is expanded or contracted. Each module passes messages to its neighbors to inform them of its current state.

In the Crystal system, locomotion can be achieved by expanding and contracting modules to create an inchworm-like motion which causes the group to move forward. However, these motions must be done in the proper order to make progress. We therefore developed an algorithm in which local rules determine whether the module is at the front, middle, or back of the group. Then, based on the physical states of its neighbors and its own position, it will contract and expand at the correct times to produce locomotion. Full details of this algorithm can be found in [3]. This algorithm will work correctly under the $D_{12}$ model, so we did not have to develop any synchronization capabilities for this system. The state of each module is sent to its neighbors after each state change, and the receiving module then checks all rules against the new neighborhood configuration to decide whether to expand or contract. The results of one experiment with this algorithm are shown in Fig. 22.

VII. Automated analysis

The analysis in the previous sections for the 5-rule and 8-rule sets shows that the correctness of cellular automata algorithms can be determined. These proofs have been manually constructed and are quite complex. This proof methodology may be hard to extend to more complex tasks. Because of this, we have developed a methodology for automating these correctness proofs using the uniformity of the robot modules, similar to the proof of the Four-Color Theorem [1], [2].

A. Graph analysis

The key concept in automated proofs of our algorithms is that any rule set applied to an initial structural configuration defines an automaton, and the properties of the graph corresponding to the automaton are the salient features of the proof of the algorithms’s correctness. The automaton consists of states, which are the possible structural configurations of the group of modules, and edges, which represent transitions from one state to another by application of some rule. If the graph of these states and the edges connecting them can be created, then it can be examined to determine whether it satisfies the properties required.

The original proof for the correctness of the 5-rule set as presented in Sec. III-A is based on three properties: (1) some rule from the rule set can always be applied to the cell array, (2) eastward motion must result from all possible sequences of rule activations, and (3) the array remains one simply connected component. If we consider the graph created from the application of this rule set on some initial configuration of modules, we wish to verify that this graph has a structure such that these three properties can be inferred.

Property (1) mandates that the graph has no leaves, since a leaf state has no edges leaving it which means that no rule applies for that state. Property (3) is trivial to verify, as it suffices to check each structural configuration in the graph for connectedness. Property (2) seems difficult to satisfy, since it would seem that this automaton is not finite. In fact, if the absolute location of the structure is considered to be a part of the state then the graph is infinite. However, if the
### Table I

Experimental results for demonstrating the correctness of the $D_{\infty}$ 11-rule set using the graph analysis method. The experiments were performed on a 2.4GHz Intel Pentium 4 computer running Linux.

<table>
<thead>
<tr>
<th>Initial Size</th>
<th>Perimeter Modules</th>
<th>Nodes</th>
<th>Edges</th>
<th>Back Edges</th>
<th>Elapsed Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3</td>
<td>7</td>
<td>295</td>
<td>839</td>
<td>13</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>4x3</td>
<td>8</td>
<td>519</td>
<td>1560</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>3x4</td>
<td>9</td>
<td>1268</td>
<td>4459</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>4x4</td>
<td>10</td>
<td>2314</td>
<td>8658</td>
<td>43</td>
<td>7</td>
</tr>
<tr>
<td>5x4</td>
<td>11</td>
<td>4143</td>
<td>16196</td>
<td>82</td>
<td>24</td>
</tr>
<tr>
<td>4x5</td>
<td>12</td>
<td>9720</td>
<td>43201</td>
<td>80</td>
<td>160</td>
</tr>
<tr>
<td>5x5</td>
<td>13</td>
<td>17934</td>
<td>83745</td>
<td>143</td>
<td>627</td>
</tr>
<tr>
<td>6x5</td>
<td>14</td>
<td>32406</td>
<td>156884</td>
<td>280</td>
<td>2173</td>
</tr>
<tr>
<td>5x6</td>
<td>15</td>
<td>74594</td>
<td>402363</td>
<td>259</td>
<td>14211</td>
</tr>
<tr>
<td>6x6</td>
<td>16</td>
<td>138547</td>
<td>778564</td>
<td>478</td>
<td>54563</td>
</tr>
</tbody>
</table>

state consists of only the structural configuration with the absolute location as an annotation, then as states with equivalent structural configurations occur during the generation of the graph it is possible to compare the annotated absolute location for the configuration already in the graph with the absolute location of the newly generated configuration to determine if property (2) holds, i.e. there is eastward displacement. In either case, a labeled back edge is created from the parent state of the newly generated configuration to the equivalent configuration already in the graph, with the edge label being the veracity of the property (2) test. The result is that the graph is actually finite, since all possible structural configurations will occur in the graph only once, and number of structural configurations is finite, assuming that the structure remains connected. Thus for the rule set to satisfy property (2) there must be no back edges in the graph which are labelled “FALSE.”

Property (2) also implicitly assumes that the structure does remain connected—otherwise the graph may not be finite. For this reason, the connectedness test is actually performed as new configurations are generated to guarantee that only connected structures are in the graph. Thus, property (3) is already satisfied once the graph is completed, since the presence of an unconnected structure results in a termination of the graph building process. Therefore, if no error occurs during graph generation, the graph has no leaves, and the graph has no back edges labelled “FALSE,” then the three properties have been satisfied and the rule set has been proven correct for that specific initial configuration.

The graph analysis method has been implemented as a variation of our cellular automata simulator. Instead of producing a graphical output, the proof mode constructs the complete graph based on the rule set and the initial module configuration, and then checks to see whether the required properties hold. Currently, the rule set must be written to use the $D_{\infty}$ activation model. This model is the easiest to use because a breadth-first configuration generation strategy can be employed in which, for each configuration, every module is evaluated for all rules and all resulting configurations are added to the graph. Although the other activation models actually reduce the number of configurations in the graph (the configuration sets are subsets of the $D_{\infty}$ configuration set), it is more difficult to generate the restricted graphs since the activation histories of all other modules must be considered. However this is not a great concern since the $D_{\infty}$ model is, in some sense, the most versatile model because arbitrary module delay is supported. Furthermore, the increased rule set complexity of the $D_{\infty}$ model makes traditional proofs more difficult, and thus the ability to prove correctness of a rule set, even if only for a specific initial configuration, is useful. Because of the $D_{\infty}$ model restriction, we have only attempted to prove rule sets designed for this model.

The 11-rule set seen in Figure 7 was tested for a variety of robot sizes between $3 \times 3$ and $6 \times 6$ modules. It was proven to function correctly for all initial configurations. For larger robot sizes, larger graphs are generated, due to the increased number of modules on the perimeter of the group which may be able to move. The number of nodes in the graph is approximately $2^{p+1}$, where $p$ is the number of perimeter modules. Details of the graph generation are presented in Table I.

One concern regarding this method of proof is the tractability of building the automata graphs. As stated above, the proof is not a general one, but is based on the initial configuration used to
generate the graph. It might be expected that similar configurations that differed only in size, such as all square initial configurations, would behave similarly and therefore the rule set would be proven correct on any of these configurations if any specific configuration passed the test. This has not been shown to be the case, but it is possible that an argument could be made for extending a specific result to larger configurations. The fundamental problem with configuration size is that the number of possible structural configurations grows exponentially in the number of modules. Thus, it is difficult to build graphs for even moderate initial configurations. Graphs for initial configurations with more than 25 modules have not been tractable using a single computer, however the use of a parallelized algorithm on multiple processors might enable larger configurations to be tested. In any case, there will always be some point beyond which proving is intractable, which emphasizes the need for a way to extend these results beyond what can be exhaustively proven by machine.

B. PAC analysis

The graph analysis method can prove absolute correctness for a rule set and a given initial module configuration, but the size of the structure is a limit to its usefulness. However, it is possible in some cases to use a different approach to overcome the size restriction at the expense of accepting a probabilistic estimate of the correctness of the algorithm. We can then state that the rule set produces correct behavior with a very high probability. This is done using a Probably Approximately Correct (PAC) learning algorithm [18]. The PAC approach can be used to bound the size of the error region, \( \epsilon \), with a confidence of \( \delta \) for a given number of correct, random simulations as follows. Assume that the size of the error region for a given rule set is \( \epsilon \). Then, the probability of running \( n \) random correct simulations is \((1 - \epsilon)^n\). We want to bound this probability by \( \delta \), and determine the resulting number of correct simulations required. This constraint on \( n \) can then be calculated as

\[
n > \frac{1}{\epsilon} \ln\left(\frac{1}{\delta}\right),
\]

This provides an estimate on the number of random correct simulations necessary to bound the error region to size of \( \epsilon \) with a confidence of \( \delta \). Thus, if a value of 0.0001 is chosen for both \( \delta \) and \( \epsilon \), the number of simulations required to be 99.99% confident that the size of the error region is no more than 0.01% of the total number of possible rule set evaluations is, \( n > \frac{2}{0.0001} \ln\left(\frac{1}{0.0001}\right) \approx 92,104 \).

While close to 100,000 simulations may seem like a large number, simulations run fairly quickly and require far fewer resources than the graph analysis method. In fact, simulations run in linear time with respect to the number of modules and the number of rules in the rule set, a much more tractable scenario than the exponential growth seen using the graph method. And, if a greater degree of confidence is required, more simulations can be run to boost the values of \( \epsilon \) and \( \delta \). The result is that the PAC approach is very effective at establishing an arbitrarily high degree of confidence that the rule set is correct, although it can never establish it absolutely.

A consequence of the PAC approach is the issue of what kinds of rule sets are most appropriate for its use. Since the locomotion algorithms are open-ended, in that their graph is completely cyclic with no specified end state, it is unclear what constitutes a “successful” simulation. It could be argued that a “sufficiently” long simulation (if “sufficiently” could be defined), or one which resulted in a return to the original configuration (ignoring displacement), would be adequate to demonstrate success. Unfortunately, it would most likely be a matter of conjecture as to whether this was true, making the PAC probability claims somewhat dubious for locomotion algorithms. For the PAC method, an algorithm with a definite final state is preferred. One instance of this type of algorithm was developed and tested in this regard. In this rule set, modules construct a cube from an initially flat sheet, so that the simulation is considered successful only if a perfect cube shape is built. PAC analysis has been performed on this rule set while building cubes between 3 modules and 8 modules on a side. 100,000 complete simulations were performed for each robot size, requiring between 1 hour (for 100,000 simulations of the 3x3x3 cube) to 52 hours (for the 8x8x8 cube). Since all simulations were successful — otherwise the algorithm would be known to be faulty — the algorithm is correct with better than 99.99% probability.

\( ^2 \)Note that various interpretations of 92,104 correct simulations are possible, such as a 99% confidence that size of the error region is no more than 0.005% of the total number of possible simulations, and a 99.999999% confidence that size of the error region is no more than 0.02%.

\( ^3 \)It is interesting the note that Equation 1 does not depend on the number of modules in the simulation. This is because it is only establishing an estimate of the percentage of erroneous simulations. Thus, simulations with larger module counts may have much larger error regions in an absolute sense, even though the percentage of erroneous simulations may be the same as for smaller simulations.
VIII. Conclusions

Self-reconfigurable robots have great potential for parallel exploration and surveillance over very rough terrain. In this paper we have presented a new general model for self-reconfiguring robots, and presented several dynamic reconfiguration algorithms within this model. These algorithms allow a group of modules to roam over terrain, build tall structures to overcome obstacles, and enter enclosed spaces through small tunnels. We have also analyzed the correctness of several of these algorithms and presented two automated techniques which have proven correctness of some of the algorithms.

We have also instantiated these algorithms to different real self-reconfiguring systems, both in simulation and hardware. This shows the utility of the generic model in that systems with very different actuation models can use the same underlying algorithm. The hardware implementation was performed on the Crystal robot, a completely distributed system. We plan in the future to extend this work both algorithmically and on real hardware. We have also begun investigation into using learning techniques for development of complex rule sets, as it is difficult to develop rules for complex tasks. The goal of all of this work is to help better realize the great potential of self-reconfiguring robots.

Acknowledgments

Robert Fitch developed the underlying engine for the Java3D simulations (more details can be found in [9]). Support for this work was provided through NSF awards IRI-9714332, EIA-9901589, IIS-9818299, IIS-9912193 and EIA-0202789 and ONR award N00014-01-1-0675.

References


