

# Self-Reconfiguration Using Graph Grammars for Modular Robotics

Daniel Pickem\* Magnus Egerstedt\*

\* *Georgia Institute of Technology, Atlanta, GA 30332 USA,*  
{*daniel.pickem,magnus*}@gatech.edu

---

**Abstract:** In this paper, we apply graph grammars to self-reconfigurable modular robots and present a method to reconfigure arbitrary initial configurations into prespecified target configurations thus connecting the motions of modules to formal assembly rules. We present an approach for centralized reconfiguration planning and decentralized, rule-based reconfiguration execution for three-dimensional modular structures. The reconfiguration is done in two stages. In the first stage, paths are planned for each module and then rewritten into production rules as defined for graph grammars. In stage two, these rules are applied in a decentralized fashion by each node individually. We show that our approach yields a unique reconfiguration sequence and a graph grammar that results in the target configuration being the only reachable stable configuration.

---

## 1. INTRODUCTION

Modular robotics is the assembly of simple individual modules into a larger, functional robot. The benefit of constructing such modular robots out of smaller building blocks is that they can be rearranged into different configurations that can perform different functions and have different capabilities. In fact, the key advantages of modular robots are their potential for versatility, robustness, and low cost. The ability to reconfigure allows modular robots to adapt to new tasks and environments by changing their morphology. Broken modules can be replaced by functional ones or new modules can be added without changing the general functionality of the structure (see for example Yim et al. [2007]).

The goal of this work is to present a novel approach for the automatic reconfiguration of three-dimensional modular robots from an arbitrary initial configuration into a desired target configuration. This process is completed in two stages; the planning and the execution stage. In stage one, paths are planned for every module from its initial position to its target position. The resulting paths are then rewritten into a ruleset. This paper follows the general route laid down in Jones and Mataric [2003], where rules are automatically generated for two-dimensional structures. Three-dimensional structures were addressed in Brandt and Ostergaard [2004], albeit only for a much smaller class of systems compared to what is done in this paper.

For the algorithm presented in this paper, the initial configuration has to be known (as is also the case for Jones and Mataric [2003] and Brandt and Ostergaard [2004]). The dependence on initial configurations has been treated in Fitch et al. [2003] for manually defined rulesets. Other work on manually defined rulesets includes Butler et al. [2004] who have demonstrated the feasibility and scalability of rule-based self-reconfiguration. The rulesets in these papers do not contain graph grammatical production rules. Klavins [2007], on the other hand, manually synthesizes graph grammars for two-dimensional reconfiguration and

we will follow the approach for three-dimensional structures in this paper.

The main contribution of this paper is the automatic generation of graph grammars for the self-reconfiguration of three-dimensional structures. Any arbitrary initially connected configuration composed of cubic modules can be reconfigured into any prespecified target configuration. The only constraints of our method are that both configurations are not allowed to contain any enclosures and have to feature an overlapping region that contains at least one module. Our approach yields a unique reconfiguration sequence and we prove that the target configuration is the only possible outcome of the reconfiguration sequence.

## 2. RELATED WORK

Much work has been done on the planning aspect of self-reconfiguration. This section specifically presents relevant rule-based reconfiguration approaches. Butler et al. [2004] describe a rule-based system inspired by cellular automata. Their rulesets are designed manually and enable groups of modules to split and merge, climb over or move around obstacles, or move through tunnels. Brandt and Ostergaard [2004] introduce a rule-based control strategy for the ATRON system (see Brandt et al. [2007]). Their rules take connectivity information into account and are automatically generated. They introduce wild card rules to reduce the size of the ruleset. Jones and Mataric [2003] show rule-based control for two-dimensional structures. The rules are automatically generated and only use connectivity information to check for rule applicability. These rules are either manually synthesized, limited to a specific class of structures or do not guarantee a successful reconfiguration to the target structure.

Graph grammars, as a tool for manipulating graphs, have also been applied to modular robotics. Klavins [2007], for example, uses graph grammars to reconfigure programmable parts, a triangle shaped hardware implementation. He manually synthesizes rulesets that are designed to

form specific structures out of the triangular modules. As opposed to our system, Klavins [2007] allows multiple rules to be applicable to the whole system at the same time. Unlike for our system, this approach does not guarantee a uniquely determined reconfiguration sequence or the reaching of the target configuration.

### 3. SYSTEM REPRESENTATION

In this paper we investigate a modular robotic system whose basic building blocks are visually represented by cubes (see Fig. 1). Moreover, no physical constraints beyond collision-avoidance such as gravity, module masses, or forces are taken into account. Additionally, the entire reconfiguration process happens in free space and is not restrained by walls, floors, or any other obstacles. These assumptions are made in order to focus the contribution on the self-reconfiguration process rather than on implementation-specific details.

Following the taxonomy in Yim et al. [2007], modular robots can generally be categorized into lattice-type and chain-type architectures. We present a lattice-based system that is embedded in a discrete coordinate system using the sliding cube model (see Fitch et al. [2003]). In the sliding cube model, every module is represented as a cube with dimension  $\delta$  (w.l.o.g. we use unit cubes, i.e.  $\delta = 1$ ), an origin  $x_i \in \mathbb{Z}^3$ , a globally unique integer identifier, and labels. A cube is capable of executing motions, which can be generally described as functions  $f(x_i, m) = x_i + m$  where  $x_i \in \mathbb{Z}^3$  and  $m \in \mathbb{Z}^3$ . Therefore, a motion moves a cube to a new position in  $\mathbb{Z}^3$ . In particular, the cubes in our system are capable of two primitive motions - sliding along a surface made of other cubes as well as transitioning to orthogonal surfaces.

*Definition 1.* A sliding motion  $m_s$  is such that  $f(x_i, m_s) = x_i + m_s$  where  $x_i \in \mathbb{Z}^3$  and  $m_s \in \mathbb{Z}^3 \wedge m_s \in \mathcal{M}_s$ .  $\mathcal{M}_s$  is the set of all possible sliding motions and is defined as  $\mathcal{M}_s = \{m \in \mathbb{Z}^3 | m_x = 1 \vee m_y = 1 \vee m_z = 1 \wedge m_x + m_y + m_z = 1\}$ .

*Definition 2.* A corner motion  $m_c$  is such that  $f(x_i, m_c) = x_i + m_c$  where  $x_i \in \mathbb{Z}^3$  and  $m_c \in \mathbb{Z}^3 \wedge m_c \in \mathcal{M}_c$ .  $\mathcal{M}_c$  is the set of all possible corner motions and is defined as  $\mathcal{M}_c = \{m \in \mathbb{Z}^3 | m_x < 2 \wedge m_y < 2 \wedge m_z < 2 \wedge m_x + m_y + m_z = 2\}$ <sup>1</sup>.

Note, however, that we will not allow the application of sliding or corner motions unless they are feasible, i.e. obey connectivity constraints and do not collide with other cubes. In fact, the movement of individual cubes requires a connected substrate of other cubes, which is referred to as a configuration. The representable space of our system is  $\mathbb{Z}^{3N}$  and any configuration  $\mathcal{C}$  is a subset of the representable space,  $\mathcal{C} \subset \mathbb{Z}^{3N}$ .

One way in which a configuration can be described is through three adjacency matrices and a labelset. Every adjacency matrix describes the adjacency of cubes along one dimension. Its entries are given by  $g(\mathcal{C}) = (A_k, l)$ , where

$$A_k = [a_{i,j,k}] = \begin{cases} 1 & \text{if } (x_i - x_j)^T \cdot b_k = 1 \\ -1 & \text{if } (x_i - x_j)^T \cdot b_k = -1 \\ 0 & \text{otherwise} \end{cases}$$

$$l(i) = l(c_i), c_i \in \mathcal{C}, i, j \in \{1, \dots, N\}$$

Here,  $k$  represents one dimension of the configuration space  $\mathbb{Z}^{3N}$  spanned by the three orthogonal base vectors  $b_k$ ,  $c_i$  stands for a cube of the configuration  $\mathcal{C}$ , and the labels  $l(i)$  of node  $i$  are the same as the labels of cube  $c_i$ . One adjacency matrix for every dimension of the configuration space is required to encode the three-dimensional geometry of the configuration. Alternatively, a configuration can be represented as a labeled graph.

*Definition 3.* The represented graph  $G = (V, E, l_G)$  is composed of the vertex set  $V$ , the edge set  $E$ , and edge and vertex label set  $l_G$  and is derived from  $(A_k, l)$  via the adjacency-to-graph mapping  $(V, E, l_G) = h(A_k, l)$ .  $V$  is a finite set of integers corresponding to the cube IDs, i.e.  $V = \{1, \dots, N\}$ , where  $N$  is the total number of cubes.  $E$  is derived from the three adjacency matrices  $A_k$  as  $E \subseteq V \times V$ , with  $e_{i,j} \in E$  if  $A_{i,j,k} \neq 0$  for some  $k \in \{1, 2, 3\}$ .  $l_G$  contains edge labels and vertex labels and is derived from  $A_k$  and  $l$  as follows:

$$l_G(v_i) = l(c_i) \quad \text{with } v_i \in V, c_i \in \mathcal{C}$$

$$l_G(e_{i,j}) = \text{sign}(A_{i,j,k})b_k \quad \text{with } e_{i,j} \in E \text{ and } A_{i,j,k} \neq 0$$

Here,  $i, j \in \{1, \dots, N\}$ ,  $l(c_i) \in l$ , and  $b_k$  is a base vector.

The vertex labels of  $v_i \in G$  are the same as the labels for the cubes  $c_i \in \mathcal{C}$ .  $G$  is a directed, labeled graph that preserves the three-dimensional structural information of the configuration.

The graph notation is required to define graph grammar rules and apply them to our system in Section 5.2. An advantage of the graph notation is that we can apply graph theoretical concepts such as the notion of connectivity.

In this paper, we assume that the initial configuration  $\mathcal{C}^I$  and the target configuration  $\mathcal{C}^T$  are known and contain the same number of modules. We then employ a two-stage planning process, in which we plan paths for every module that has to be moved in stage one (see Section 4) and generate the ruleset based on these paths in stage two (see Section 5.2). The ruleset is then executed as shown in Section 5.3.

### 4. PATH PLANNING

The reconfiguration process requires us to move cubes from their initial positions to their target positions. Therefore, we have to calculate paths for cubes  $c_i \in \mathcal{C}^I$  to their desired positions in  $c_j \in \mathcal{C}^T$ . Cubes  $c_i$  in the initially overlapping region  $\mathcal{O}_{init} = \mathcal{C}^I \cap \mathcal{C}^T$  (see Fig. 1(a)), do not have to be moved and are excluded from the planning process. The planning stage is composed of multiple steps, which are the calculation of the currently overlapping region  $\mathcal{O} = \mathcal{C} \cap \mathcal{C}^T$ , the movable set  $\mathcal{M}$ , the immediate target successor set  $\mathcal{R}$ , the path calculation (for every cube  $c_i \in \mathcal{C}^I \setminus \mathcal{O}_{init}$ ), and the ruleset generation (see Section 5.2). In this section, we will formally define  $\mathcal{M}$  and  $\mathcal{R}$ , the notion of articulation points and paths, and describe the planning approach.

*Definition 4.* An articulation point  $v$  in a graph  $G$  is a node whose removal would increase the number of

<sup>1</sup> In Rus and Vona [2001], a corner motion is referred to as *convex motion*

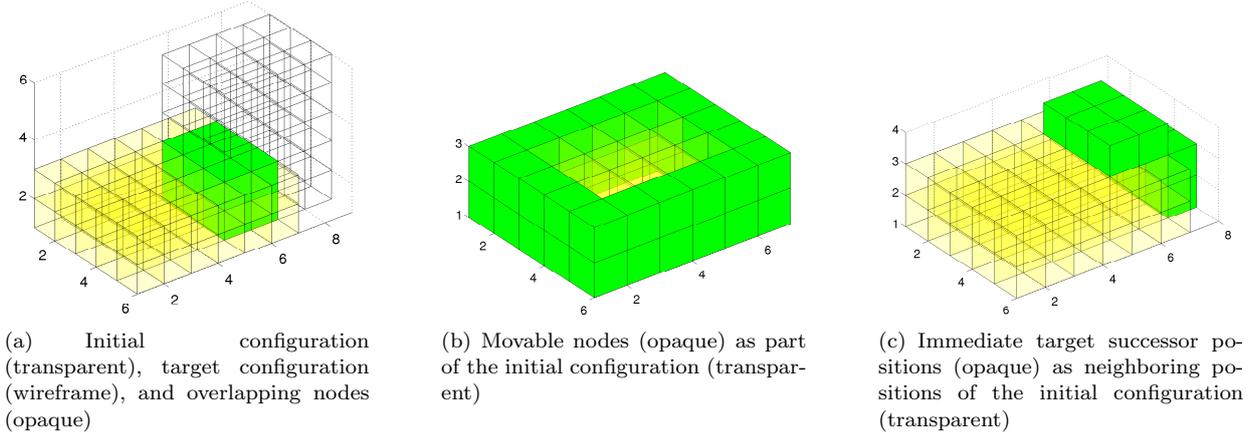


Fig. 1. Graphical representation of the overlapping, movable, and immediate target successor set in the simulator

connected components  $c(G)$ , i.e.  $c(G - v) > c(G)$ <sup>2</sup>. In other words, the removal of  $v$  would disconnect the graph.

A connected graph  $G$  has only one connected component,  $c(G) = 1$ . Our self-reconfigurable system has to remain connected at all times to guarantee a successful reconfiguration. In order to enforce this requirement, we have to ensure that a node that is an articulation point is never moved. The movable set is therefore defined as follows:

*Definition 5.* The *movable set*  $\mathcal{M}$  is a set of cubes that can be moved without disconnecting the configuration and is defined as  $\mathcal{M} = \{c_i \in \mathcal{C} \mid c_i \in \mathcal{C}^I \setminus \mathcal{O} \wedge c_i \notin \mathcal{A}(\mathcal{C}) \wedge |\mathcal{N}_1(c_i, \mathcal{C})| \leq 5\}$ .  $\mathcal{N}_1(c_i, \mathcal{C})$  is the one-hop neighborhood and defined as  $\mathcal{N}_1(c_i, \mathcal{C}) = \{c_j \in \mathcal{C} \mid \text{dist}(c_i, c_j) = 1\}$ .  $\mathcal{A}(\mathcal{C})$  is the set of articulation points of the graph representing the current configuration  $\mathcal{C}$ .

This definition is based on the sliding cube model (see Fitch et al. [2003]) and only allows modules on the surface of the configuration to be relocated. This is because Def. 5 excludes immobile cubes within the configuration (i.e. cubes that have six neighbors) from the movable set. While  $\mathcal{M}$  is a set of movable cubes, we need to find potential target positions for cubes  $c_i \in \mathcal{M}$  - the immediate target successor set  $\mathcal{R}$ .

*Definition 6.* The *immediate target successor set*  $\mathcal{R}$  is defined as positions  $c_j \in \mathcal{C}^T$  that are adjacent to the current configuration, i.e. lie in the one-hop neighborhood  $\mathcal{N}_1(\mathcal{C})$  of the current configuration  $\mathcal{C}$  as well as in the target configuration  $\mathcal{C}^T$ .  $\mathcal{R} = (\mathcal{C}^T \cap \mathcal{N}_1(\mathcal{C})) \setminus \mathcal{C}$ , where  $\mathcal{N}_1(\mathcal{C}) = \{c_j \mid c_j \notin \mathcal{C} \wedge c_i \in \mathcal{C} \wedge \text{dist}(c_i, c_j) = 1\}$  (see Fig. 1(c)). Therefore,  $\mathcal{R}$  is a subset of  $\mathcal{N}_1(\mathcal{C})$ .

Note that  $\mathcal{N}_1(\mathcal{C})$  is the one-hop hull of the current configuration  $\mathcal{C}$  and at the same time the planning space for the path planner. In that planning space, we plan a path  $p_i$  for a single cube at a time, i.e. from  $c_i \in \mathcal{M}$  to  $c_j \in \mathcal{R}$ . We know by assumption that  $\mathcal{R}$  is nonempty unless the target configuration has already been assembled. The path  $p_i$  is only allowed to contain positions  $c_k \in \mathcal{N}_1(\mathcal{C})$  and use primitive motions to move the current cube  $c_i$ .

*Definition 7.* A path is a concatenation of motions  $m \in \{m_c, m_s\}$  (see Def. 1 and Def. 2) that move cube  $c_i \in (\mathcal{C}^I \setminus$

<sup>2</sup> Connected components of a graph  $G$  are its maximal connected subgraphs.

$\mathcal{O}) \cap \mathcal{M}$  to position  $c_j \in \mathcal{C}^T \cap \mathcal{R}$ . The length of the path  $p_i$ , or the total number of motions, is denoted as  $|p_i|$ .

Both  $\mathcal{M}$  and  $\mathcal{R}$  define a set of cubes. Before we can plan a path for a cube  $c_i$  to a position  $c_j$ , we need to define an assignment. Therefore, we calculate the pairwise costs between any two cubes  $c_i \in \mathcal{M}$  and  $c_j \in \mathcal{R}$  and pick the pair of cubes with the lowest cost. The assignment resulting from this greedy approach is then used as input for the path planner. This process is repeated for all cubes  $c_i \in \mathcal{C}^I \setminus \mathcal{O}$  and paths are computed to their respective target positions  $c_j \in \mathcal{C}^T$ . Note that this approach ensures that progress is never blocked and the target configuration will indeed be assembled given that the initial configuration is connected and does not contain any enclosures (see Theorem 1)

In this paper, we use A\* for path planning together with the Manhattan distance as cost metric as it most accurately represents the discrete lattice structure of our system and the possible movements of the nodes. The result of the path planning stage is a set of paths that describe the complete reconfiguration from  $\mathcal{C}^I$  to  $\mathcal{C}^T$ . This set of paths is then rewritten into a ruleset as discussed in the next section.

## 5. RULE GENERATION

In this paper, we employ graph grammatical concepts to bridge the gap between global information that is available during planning and local information that is available to the cubes during reconfiguration. The centralized path planning results are rewritten into rules that can be checked locally for applicability. Contrary to rules only based on connectivity information, graph grammars offer fine-grained control over the applicability of rules and allow the encoding of additional information into the labels of the rules. Before we introduce our rule generation and execution approach, we define the graph grammatical terms used in this paper, taken from Klavins et al. [2006].

*Definition 8.* A *production rule* or simply a *rule* consists of two labeled graphs; a left-hand side  $g_l$  and a right-hand side  $g_r$ . It describes a transformation of a graph  $G_S$ , that is isomorphic to  $g_l$ , from  $G_S$  to  $g_r$ .

A graph grammar is a set of production rules that operate on a graph  $G_0$ . Therefore, we call the pair  $(G_0, \Phi)$  a system, where  $G_0$  is an initial labeled graph and  $\Phi$  is a graph grammar. A rule  $r \in \Phi$  can be applied to  $G_0$  only when it is applicable:

*Definition 9.* A rule is *applicable* to  $G$  if there exists a subgraph  $G_S$  of  $G$  that is isomorphic to  $g_l$ . This is also denoted as  $G_S \cong g_l$ .

Once the applicability of a rule  $r$  to a subgraph  $G_S$  of  $G$  has been determined,  $r$  can be applied to  $G$ . The application of a rule  $r$  yields a new graph  $G_i \xrightarrow{r} G_{i+1}$ , where  $G_{i+1}$  results from  $G_i$  by replacing the subgraph  $G_S$  with  $g_r$ . Each step in the reconfiguration process yields a graph  $G_i$  that is part of a trajectory, i.e. a finite or infinite sequence  $\sigma = \{G_i\}_{i=0}^k$  s.t. there exists a sequence of applicable rules  $\{r_i\}_{i=0}^{k-1}$  where  $r_i \in \Phi$  and  $G_i \xrightarrow{r_i} G_{i+1}$ . Each graph  $G_i$  is called *reachable* by the system  $(G_0, \Phi)$ . A reachable graph can be temporary, such that some rule  $r$  in  $\Phi$  is applicable to it, or *stable* (see Klavins et al. [2006]).

*Definition 10.* A graph  $G$  is *stable*, if no rule in  $\Phi$  is applicable to it.

Stable graphs play an important role in this paper. In this section, we prove that the graph representing  $\mathcal{C}^T$  is the only reachable stable graph given a system  $(G_0, \Phi)$ . Here,  $G_0$  is derived from  $\mathcal{C}^T$  and  $\Phi$  is an automatically generated graph grammar. Furthermore, this section describes the structure of our rules and the rule generation framework.

### 5.1 Rule Structure

For the purpose of self-reconfiguration, a production rule for our system uses the rule structure shown in Def. 8 and its two labeled graphs  $g_l$  and  $g_r$  are defined as follows:

$$\begin{aligned} g_l &= f(\mathcal{N}_2(c_i, \mathcal{C})) \\ g_r &= f(\mathcal{N}_2(c_i + m, \mathcal{C})), \end{aligned}$$

$f$  is given below but essentially maps from cubesets to graphs.  $\mathcal{N}_2(c_i, \mathcal{C})$  is the immediate motion successor set of the current cube  $c_i$  in the configuration  $\mathcal{C}$  and is given by  $\mathcal{N}_2(c_i, \mathcal{C}) = \{c_j \in \mathcal{C} \mid c_i \in \mathcal{C} \wedge \text{dist}(c_i, c_j) \leq \sqrt{2}\}$ .  $\mathcal{N}_2(c_i, \mathcal{C})$  contains all cubes at a distance of one primitive motion from cube  $c_i$ . Both graphs,  $g_l$  and  $g_r$ , are derived from the sets  $\mathcal{N}_2(c_i, \mathcal{C})$  and  $\mathcal{N}_2(c_i + m, \mathcal{C})$  of  $c_i$  and its current motion  $m$  (given by the path planner) via the cubeset-to-graph mapping  $f$ .

*Definition 11.* The relationship between a cubeset  $\mathcal{C}$  and a graph  $G = (V, E, l_G)$  is given by the *cubeset-to-graph mapping*  $f$ , which is defined as  $f = h \circ g$  such that  $(V, E, l_G) = f(\mathcal{C})$  with mappings  $g$  and  $h$  defined in Section 3. The inverse *graph-to-cubeset mapping*  $f^{-1}$  is given by  $f^{-1} = g^{-1} \circ h^{-1}$  such that  $f^{-1}(V, E, l_G) \ni \mathcal{C}$ .

The application of a rule  $r$  to a subgraph  $G_S$ , i.e.  $r(G_S \cong g_l) = g_r$ , yields a new graph  $G'$ . The changes in the edge and label set described by  $G = (V, E, l_G) \xrightarrow{r} (V, E', l'_G) = G'$  represent the motion in the configuration space, i.e.  $\mathcal{C} = f^{-1}(G)$  and  $\mathcal{C}' = f^{-1}(G')$ , where  $c_i$  has been moved from  $c_i \in \mathcal{C}$  to  $c_i + m \in \mathcal{C}'$ . Fig. 2 shows a graphical representation of  $\mathcal{N}_2(c_i, \mathcal{C}) = f^{-1}(g_l)$  and  $\mathcal{N}_2(c_i + m, \mathcal{C}) =$

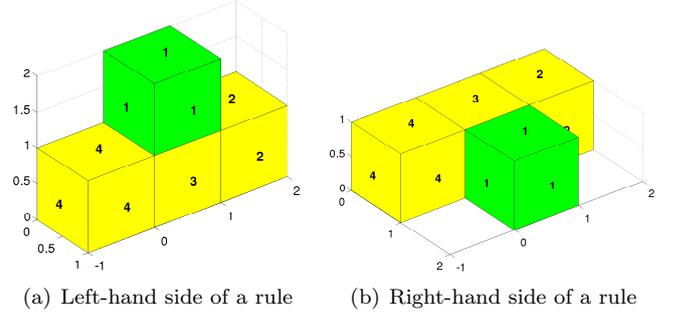


Fig. 2. Visual representation of a rule showing a corner motion of cube 1.

$f^{-1}(g_r)$  of some rule. The highlighted cube is the currently active cube  $c_i$ .

As part of  $g_l$  and  $g_r$ , each rule contains information about how the labels of the current node change through the application of the rule as well as optional label updates for the neighbors. Each label is composed of multiple data fields, which include the *node ID*, the *rule ID*, a *flooding flag* indicating the start and the end of the flooding process, and a field storing the *latest finished path*. The *node ID* and the *rule ID* are globally unique integers and ensure the uniqueness of each rule and the unambiguity of the whole reconfiguration. The *flooding flag* controls the start and end of the propagation process to update every node's knowledge about the latest finished path. The field *last path* concludes a label and stores the most recently finished path locally at every node. This field also controls the execution sequence of all individual paths, since the execution of path  $p_i$  depends on the conclusion of path  $p_{i-1}$ . The initial labeling of all nodes of the graph  $G_0 = (V, E, l_G) = f(\mathcal{C}^T)$  and the label update mechanism through rules are designed so that only one rule is applicable to any cube  $v_i \in V$  at any given time. Therefore, the reconfiguration is unambiguous and deterministic.

### 5.2 Rule Generation

The main contribution of this paper is the automatic generation of a graph grammar  $\Phi$  that describes the unambiguous reconfiguration  $\mathcal{C}^T \xrightarrow{\Phi} \mathcal{C}^T$ . The path planning and the rule generation are interleaved, which means that once a path  $p_i$  ( $i \in \{1..|P|\}$  where  $|P| = |\mathcal{C}^T \setminus \mathcal{O}_{init}| = |\mathcal{C}^T \setminus \mathcal{O}_{init}|$ ) has been computed, the ruleset  $R_{p_i}$  that represents  $p_i$  is generated.  $R_{p_i}$  consists of  $|p_i|$  motion rules, one flooding activation rule, and one propagation rule. More formally  $R_{p_i}$  is defined as  $R_{p_i} = \{\{r_{m_i}\}_{i=1}^{|p_i|}, r_p, r_f\}$ . The entire ruleset  $\Phi$  is composed of all sub-rulesets  $R_{p_i}$ , i.e.  $\Phi = \{\{R_{p_i}\}_{i=1}^{|P|}\}$ . The three types of generated rules are defined as follows:

*Definition 12.* A *motion rule*  $r_m$  changes the edge set and the label set of the graph  $G = (V, E, l_G)$ , specifically those edges whose end point is the current node  $v_i$ . Therefore, the application of a motion rule results in the motion of a cube  $c_i$  (represented by node  $v_i \in G$ ) in the configuration space  $\mathcal{C}$ . More formally,  $r_m$  rewrites the graph  $G = (V, E, l_G)$  as follows:

$$(V, E, l_G(v_i)) \xrightarrow{r_m} (V, E', l'_G(v_i))$$

*Definition 13.* A *flooding activation rule*  $r_f$  updates the *last path* field of the current node  $v_i$  and sets the *flooding* flag from 1 to 0, which activates the corresponding propagation rule. A flooding rule only affects the labels of the current node  $v_i$  and does change the edge set. More formally,  $r_f$  rewrites the graph  $G = (V, E, l_G)$  as follows:

$$(V, E, l_G(v_i)) \xrightarrow{r_f} (V, E, l'_G(v_i))$$

*Definition 14.* A *propagation rule*  $r_p$  updates the current node  $v_i$ 's labels by setting the *flooding* flag from 0 to 1 and incrementing the *last path* field of all its neighbors  $v_j \in f(\mathcal{N}_2(c_i, \mathcal{C}))$ . It also sets the *flooding* flag of its neighbors  $v_j$  to 0 so that the same rule  $r_p$  is applicable to them. This type of rule is a wildcard rule w.r.t. the node ID. A propagation rule does not change the edge set. More formally,  $r_p$  rewrites the graph  $G = (V, E, l_G)$  as follows:

$$(V, E, l_G(v_i, v_j)) \xrightarrow{r_p} (V, E, l'_G(v_i, v_j))$$

For each motion  $m_j$  in the path  $p_i$ , the neighborhood structure of two consecutive positions of the active cube is calculated and stored in a rule. Additionally, each rule stores the labels before and after the application of the rule. More formally, for each motion  $m_j$  ( $j \in \{1..|p_i|\}$  as defined in Def. 7) of path  $p_i$ , our algorithm generates a motion rule  $r_{i,j}$  composed of  $g_l$  and  $g_r$ :

$$g_l = f(\mathcal{N}_2(c_i + (\sum_{k=1}^j m_k) - m_j, \mathcal{C}))$$

$$g_r = f(\mathcal{N}_2(c_i + \sum_{k=1}^j m_k, \mathcal{C}))$$

Here,  $c_i$  is the currently moved cube and the starting point of path  $p_i$  and  $\mathcal{N}_2(c_i, \mathcal{C})$  is the immediate motion successor set as defined in Section 5.1. The labels of  $g_l$  are defined the following way:

$$l_G(g_{l_{i,j}}) = \begin{cases} l_G(g_{r_{i,j-1}}) & \text{if } j > 1 \\ l_G(g_{r_{i-1,|p_{i-1}|}}) & \text{if } j = 1, i > 1 \\ l_{G,init} & \text{otherwise} \end{cases}$$

The labels of  $g_r$  are derived from the labels of  $g_l$  via the label update mechanism defined for motion rules, flooding rules, and propagation rules and can be summarized as follows.

$$l_G(g_{r_{i,j}}) = \begin{cases} l_G(g_{l_{i,j}}) \xrightarrow{r_m} l_G(g_{r_{i,j}}) & \text{for motion rules} \\ l_G(g_{l_{i,j}}) \xrightarrow{r_f} l_G(g_{r_{i,j}}) & \text{for flooding rules} \\ l_G(g_{l_{i,j}}) \xrightarrow{r_p} l_G(g_{r_{i,j}}) & \text{for propagation rules} \end{cases}$$

The labels are created with a strictly monotonically increasing global rule ID ensuring that each rule is globally unique and describes exactly one step in the complete reconfiguration sequence.

This process is repeated for every motion  $m_j$  of path  $p_i$ . After the end of the current path  $p_i$  is reached a flooding activation rule and a propagation rule are generated, resulting in a ruleset  $R_{p_i} = \{\{r_{m_i}\}_{i=1}^{|p_i|}, r_p, r_f\}$ . The rule generation process is repeated for every path  $p_i$  ( $i \in \{1..|P|\}$ ) until the reconfiguration is completed, i.e. until the target configuration  $\mathcal{C}^T$  has been assembled. This means that the only reachable, stable graph as defined

in Def. 10 is the graph representing the desired target configuration  $\mathcal{C}^T$ .

*Theorem 1.* The graph  $G = (V, E, l_G) = f(\mathcal{C}^T)$  representing the target configuration  $\mathcal{C}^T$  is the only reachable, stable graph to the ruleset  $\Phi$ .

*Proof 1.* This proof is based on Theorem 1 in Rus and Vona [2001] and the definition and properties of a unit-modular self-reconfiguring system. Our system is composed of unit cubes, which can be assembled into arbitrarily shaped configurations. Thus our system satisfies property one. Property two states that in a configuration composed of unit modules, there always exists a module that can be relocated to any position on the surface  $S$ . In our system,  $S$  is defined as  $S = \mathcal{N}_1(\mathcal{C}) = \{c_i | c_i \notin \mathcal{C} \wedge c_j \in \mathcal{C} \wedge \text{dist}(c_i, c_j) = 1\}$  and  $\mathcal{R}$  is a subset of  $S$ ,  $\mathcal{R} \subset S$ . The movable set  $\mathcal{M}$ , on the other hand, is a subset of the boundary  $\partial\mathcal{C}$  of  $\mathcal{C}$ , where  $\partial\mathcal{C} = \{c_i | c_i \in \mathcal{C} \wedge |\mathcal{N}(c_i, \mathcal{C})| \leq 5\}$ .  $|\mathcal{M}| \geq 2$  according to Lemma 6 in Rus and Vona [2001] and only contains cubes  $c_i \in \partial\mathcal{C}$ . Therefore, our system fulfills property two of Theorem 1 as well. As a result, our system is self-reconfigurable and  $\mathcal{C}^T$  can be assembled incrementally from  $\mathcal{C}^I$ . Therefore, every cube  $c_i \in \mathcal{C}^I \setminus \mathcal{O}_{init}$  will be moved to its target position  $c_j \in \mathcal{C}^T$ . Since the current configuration  $\mathcal{C}$  always remains connected (by construction of  $\mathcal{M}$  and  $\mathcal{R}$ ), i.e.  $c(G) = c(f(\mathcal{C})) = 1$ , a path always exists between  $c_i$  and  $c_j$ . The individual module paths are planned sequentially, which means that path  $p_{i+1}$  is planned after path  $p_i$  was planned and executed. This approach implicitly determines a unique reconfiguration sequence. The outcome of the planning stage, i.e. the execution of paths  $p_i$  for  $i \in \{1, \dots, N\}$  with  $N = |\mathcal{C}^I \setminus \mathcal{O}_{init}|$ , therefore, unambiguously yields  $\mathcal{C}^T$ .

By construction of the ruleset and the label update mechanism, the same sequence of reconfiguration steps is achieved during the execution stage of the ruleset. Therefore, we can conclude that the only reachable, stable graph is  $(V, E, l_G) = f(\mathcal{C}^T)$ .  $\square$

### 5.3 Ruleset Execution

The goal of the ruleset execution is the reconfiguration of  $\mathcal{C}^I$  into  $\mathcal{C}^T$ . In other words, given a system  $(G_0, \Phi)$  we want to execute the assembly sequence  $G_0 \xrightarrow{r_1} G_1 \xrightarrow{r_2} G_2 \xrightarrow{r_3} \dots \xrightarrow{r_n} G_{stable}$ , where  $G_0 = f(\mathcal{C}^I)$ ,  $G_{stable} = f(\mathcal{C}^T)$ , and  $n$  is the total number of rules. To accomplish this reconfiguration, every node  $v_i \in \mathcal{G}$  periodically checks the ruleset for applicable rules  $r \in \Phi$ . If the graph represented by the current neighborhood  $\mathcal{N}_2(c_i, \mathcal{C})$ , i.e.  $G_S = f(\mathcal{N}_2(c_i, \mathcal{C}))$  is isomorphic to the left-hand side  $g_l$  of some rule  $r \in \Phi$ , an applicable rule has been found and is applied to the current node  $v_i$ . The application of a rule  $r$  rewrites the subgraph  $G_S$  into  $g_r$ , i.e.  $G_S \xrightarrow{r} g_r$ . If the application of a rule changes the edge structure of  $G_S$ , which is the case only for motion rules, the cube  $c_i$  is moved in the configuration space. The execution of the last motion rule  $r_{m_i, |p_i|}$  of a path  $p_i$  triggers a flooding activation rule  $r_{f_i}$ . This rule in turn triggers a propagation rule  $r_{p_i}$ . Through the repeated application of  $r_{p_i}$  to every node  $v_i \in V$  every node's local state is updated about the completion of the latest path through directed flooding. This process is repeated until every path  $p_i$  is completed and no more rules in  $\Phi$  are applicable to any node  $v_i \in V$ ,

Table 1. Reconfiguration planning results for overlapping box configurations

Size	Overlap [N]/[%]	Steps	Rules	Runtime [min]
100	30 / 30%	837	907	3.70
200	60 / 30%	1543	1683	16.65
300	90 / 30%	2426	2636	63.26
400	120 / 30%	3279	3559	135.64
500	150 / 30%	4275	4625	246.93

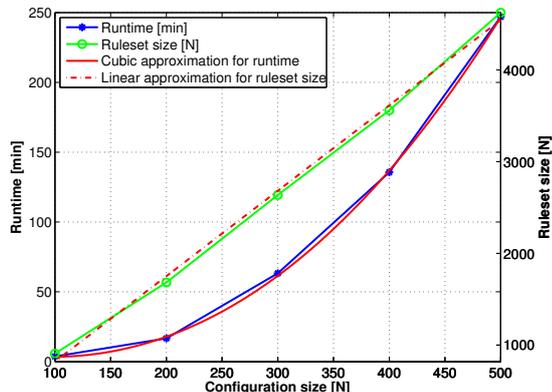


Fig. 3. Number of generated rules and required runtime for ruleset generation of box configurations

i.e. until a stable graph is reached. An example of a reconfiguration sequence is shown in Fig. 4.

## 6. RESULTS

This section presents simulation results obtained by reconfiguring overlapping box configurations (see Table 1), which ranged in size from 100 to 500 modules. Our test system was equipped with an Intel Core i5-540M dual core processor and 4GB of DDR3 memory.

In Table 1, the field *Size* refers to the number of modules in the configuration, *Overlap* is the number of initially overlapping modules, *Steps* is the total number of motions of all modules to achieve the desired reconfiguration, *Rules* is the total number of generated rules in the ruleset, and *Runtime* is the time it took to generate the ruleset. As shown in Fig. 3, the size of the ruleset increases approximately linearly with the number of nodes, while the runtime of our algorithm increases approximately cubically for the box configurations. The runtime is primarily determined by the planning approach, which necessitates planning a path for every individual node. Our algorithm features a time complexity of  $O(N^2)$  for the relocation of an individual cube and a total time complexity of  $O(N^3)$  for the entire reconfiguration process.

## REFERENCES

D. Brandt, D. J. Christensen, and H. H. Lund. Atron robots: Versatility from self-reconfigurable modules. In *In Proceedings of the IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 2254–2260, Harbin, China, Aug. 2007.

David Brandt and Esben H. Ostergaard. Behaviour subdivision and generalization of rules in rule-based control of the atron self-reconfigurable robot, 2004.

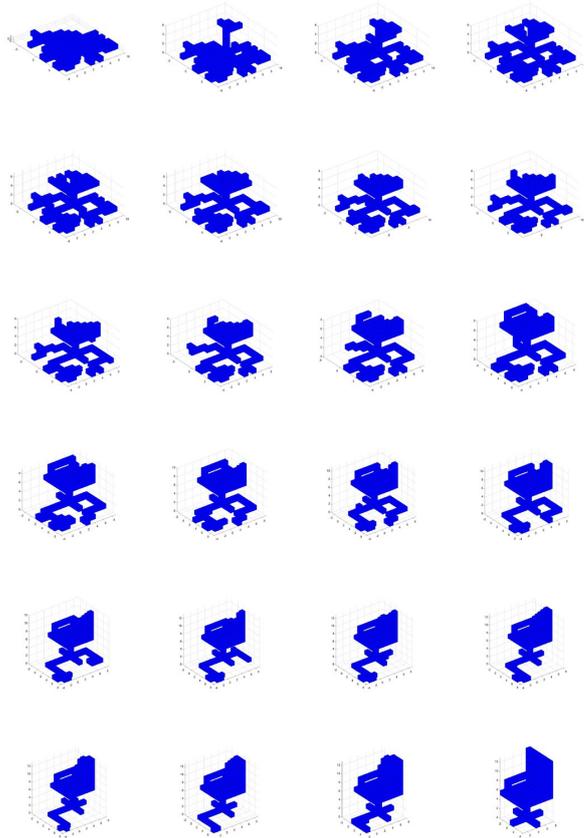


Fig. 4. Example of a reconfiguration sequence from a random two-dimensional configuration to a chair.

Zack Butler, Keith Kotay, Daniela Rus, and Kohji Tomita. Generic decentralized control for lattice-based self-reconfigurable robots. *The International Journal of Robotics Research*, 23(9):919–937, 2004.

R. Fitch, Z. Butler, and D. Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2460 – 2467, Oct. 2003.

Chris Jones and Maja J. Mataric. From local to global behavior in intelligent self-assembly. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA 2003, September 14-19, 2003, Taipei, Taiwan*, pages 721–726. IEEE, 2003.

E. Klavins. Programmable self-assembly. *Control Systems, IEEE*, 27(4):43–56, Aug. 2007.

E. Klavins, R. Ghrist, and D. Lipsky. A grammatical approach to self-organizing robotic systems. *Automatic Control, IEEE Transactions on*, 51 Issue: 6:949 – 962, 2006.

Daniela Rus and Masette Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10(1):107–124, Jan. 2001.

Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular self-reconfigurable robot systems – challenges and opportunities for the future. *IEEE Robotics and Automation Magazine*, March:43–53, 2007.