3D RECONFIGURATION USING GRAPH GRAMMARS FOR MODULAR ROBOTICS

A Thesis
Presented to
The Academic Faculty

by

Daniel Pickem

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2012

3D RECONFIGURATION USING GRAPH GRAMMARS FOR MODULAR ROBOTICS

Approved by:

Professor Magnus Egerstedt, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Jeff Shamma
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Patricio Antonio Vela
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 1 July 2010

ACKNOWLEDGEMENTS

I want to thank my adviser, Dr. Magnus Egerstedt, who spurred my interest in graph grammars and modular robotics through his course on networked control. His feedback and guidance throughout my thesis research were invaluable. I also want to thank my family for their continuous support and encouragement in my pursuit of my Master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this thesis is to develop a method for the reconfiguration of three-dimensional modular robots. A modular robot is composed of simple individual building blocks or modules. Each of these modules needs to be controlled and actuated individually in order to make the robot perform useful tasks. The presented method allows us to reconfigure arbitrary initial configurations of modules into any pre-specified target configuration by using graph grammar rules that rely on local information only. Local in a sense that each module needs just information from neighboring modules in order to decide its next reconfiguration step. The advantage of this approach is that the modules do not need global knowledge about the whole configuration. We propose a two stage reconfiguration process composed of a centralized planning stage and a decentralized, rule-based reconfiguration stage. In the first stage, paths are planned for each module and then rewritten into a ruleset, also called a graph grammar. Global knowledge about the configuration is available to the planner. In stage two, these rules are applied in a decentralized fashion by each node individually and with local knowledge only. Each module can check the ruleset for applicable rules in parallel. This approach has been implemented in Matlab and currently, we are able to generate rulesets for arbitrary homogeneous input configurations.

CHAPTER I

INTRODUCTION

Modular or self-reconfigurable robotics describes the assembly of simple individual, independent modules into larger, functional robots that can perform tasks such as locomotion or reconfiguration. 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. A modular robot can, therefore, adapt to changing environments and task specifications. With ever increasing computational power to control such high-dimension-of-freedom-robots and the decreasing cost of producing a large number of modules, modular robots are becoming a viable alternative to fixed morphology robots.

This thesis addresses the problem of how to efficiently control a large number of modules in a distributed, decentralized fashion. Our goal is to develop a rule-based reconfiguration approach, where each module can locally check the applicability of rules and apply them. In this sense the presented system is a massively distributed robot. This chapter introduces self-reconfigurable robotic systems, gives a brief description of the problem, presents previously used approaches to address the problem, and outlines a proposed solution.

1.1 Self-Reconfigurable Robots

Self-reconfigurable systems were first proposed by Toshio Fukuda in the late 1980s (see [20]). His platform, the CEBOT, was able to reconfigure itself to fit the environment and the task it had to fulfill. This robot already featured most of the characteristics of a modern self-reconfigurable robot. It consisted of separable, autonomous units

that were able to communicate, connect, and separate. Broken modules could be replaced by functional ones and the robot was able to maintain system function. Over the course of the last two decades, self-reconfigurable robots have been scaled up to contain thousands of modules (see for example [18]).

Also called metamorphic robots, self-reconfigurable robots can change their aggregate geometric structure without outside assistance or control. They are composed of independently controlled modules that are capable of approximating arbitrary shapes. Each module has the ability to connect and disconnect from and move over a substrate of other modules. Therefore, the robot can dynamically and autonomously reconfigure to adapt to the terrain, environment, and task. A self-reconfigurable robot can support multiple modes of locomotion and manipulation by changing its morphology. For example, a robot can enter a collapsed building for search and rescue in snake-from and explore the building in a configuration featuring legs or wheels.

Self-reconfigurable robots can be categorized according to a number of criteria, two of which are the architecture of the robot and the type of modules it is composed of. The two most often used architectures in the literature are the lattice-type and the chain-type architecture (see [65]). Lattice-type robots contain modules that are arranged and connected in a regular pattern or grid (see for example [19]). Cubic grids are a popular choice due to their high symmetry (see [10], [59], [38], or [18]), but other lattice types such as hexagonal grids are also used in the literature (see [5]). A lattice-type robot moves by relocating individual modules on the surface of the robot thus creating the impression that the robot flows. Control and motion can be executed in parallel. Lattice-type robots usually offer simpler reconfiguration than chain-type ones since modules can only move to a discrete set of neighborhood locations. Therefore, this type of robots can be scaled up to contain a higher number of robots more easily. On the other hand, chain-type robots contain modules that are connected in a tree or string topology and move by articulating their powered

joints (see for example [42], [61], or [1]). Such a chain can fold up to fill the space but the underlying architecture is serial. A chain-type robot can reach any point or orientation in the configuration space (i.e. continuous reconfiguration). It is therefore more versatile than lattice-type robots, but also more difficult to represent and control. The second distinction that can be made about self-reconfigurable robots pertains to homogeneity and heterogeneity, i.e. whether all modules of a robot have the same properties or differ in one or more property. A variety of algorithms have been proposed for homogeneous systems (for example [9], [68], or [18]) and in recent years heterogeneous systems have enjoyed the same attention (see [19]). One last criterion that is relevant in this thesis is the distinction between static and dynamic reconfiguration. According to [12], static refers to reconfiguring a given shape at the current location while dynamic reconfiguration uses the shape-changing ability to locomote through the environment.

Modular robots offer a variety of advantages over fixed-morphology robots. In fact, the key advantages of modular robots is their versatility, but also their potential for robustness and low cost (see [65]). Broken modules can be replaced by functional ones or new modules can be added without changing the general functionality of the structure. The low cost of producing modular robots stems from the fact that they are built of identical, relatively simple modules that can be mass-produced cheaply. This only seems to be true for homogeneous robots, but even for heterogeneous robots the number of specialized modules is small. So the majority of a heterogeneous robot is made of identical modules as well. One major drawback of modular robots is their lack of specialization and as a result their inferior performance compared to robots tailored to a specific task or mode of locomotion. Additionally, the versatility of modular robots and their high number of degrees of freedom come at the cost of increased mechanical and computational complexity of controlling them.

The possible tasks for self-reconfigurable robots are manifold. To name just a few

examples, they can be employed for exploration ([65]), self-assembly and self-repair of tools ([31]), self-reconfigurable furniture ([18]), or larger structures ([7]). Their shape-shifting capabilities can be used for locomotion in unstructured environments (as in [18]), flexible manipulation ([7]), or three-dimensional visualization. In general, self-reconfigurable robots are well-suited for working in environments and on tasks with incomplete a-priori knowledge.

1.2 Problem Description

Self-reconfiguration can be done in a centralized or decentralized way. While centralized self-reconfiguration suffers from a lack of scalability due to limited parallelism, decentralized self-reconfiguration lacks in efficiency during the planning stage mainly because of the communication overhead due to message passing. Additionally self-reconfiguration has to address the question of how to create a desired global behavior and shapes based on local information and interactions. While this is straightforward for centralized approaches that have global knowledge during the planning, decentralized approaches run the risk of getting stuck in local minima (and therefore a suboptimal reconfiguration) during planning due to limited local knowledge. A desirable solution to the reconfiguration problem would e to use a hybrid approach merging both centralized and decentralized concepts. Rule-based self-reconfiguration with manually (for example [31], [8], [10], [12], or [66]) and automatically generated rules (for example [25] or [7])have been used in the literature to achieve just that. None of the existing approaches fulfills all of the following criteria though: arbitrary three-dimensional input and output configurations, automatic generation of rulesets, unambiguity of the reconfiguration, guaranteed reaching of the target configuration,

straightforward extensibility to heterogeneous systems. To address all of these requirements, we propose a centralized planning approach that encodes all the necessary information in locally applicable rules and a decentralized, distributed execution stage.

1.3 Goal of this Thesis

The goal of this thesis is to present a novel approach for the automatic self-reconfiguration of three-dimensional modular robots from an initial configuration $\mathcal{C}^{\mathcal{I}}$ into a desired target configuration $\mathcal{C}^{\mathcal{T}}$. In other words, we want our system to automatically plan and execute $\mathcal{C}^{\mathcal{I}} \xrightarrow{\Phi} \mathcal{C}^{\mathcal{T}}$, where Φ is a graph grammar or ruleset that is automatically generated. 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. This planning is done in a centralized fashion with global knowledge. The resulting paths are then rewritten into a graph grammar Φ composed of production rules. After generating the rules in stage one, these rules can be checked for applicability and applied in stage two. During this rule execution stage, modules can only access local information about neighboring modules and apply the rules in a decentralized fashion.

The main contribution of this thesis 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, connected 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.

CHAPTER II

BACKGROUND AND PREVIOUS WORK

Modular robots with their large number of individual components require novel, scalable control strategies and tools for describing the structure and dynamics of the system. In recent years, decentralized, distributed approaches to self-reconfiguration have been shown to scale better than centralized ones. One way of dealing with these distributed robotic systems is to employ graph theory and graph grammar theory. In networked and distributed robotics, graph theory has been established as a reliable tool for describing the topology of large teams of individual agents. It also allows us to formally describe how local interactions affect the global system behavior. Graph grammars, on the other hand, provide the means to formally represent these local interactions with rules. Combining both graph theory and graph grammars allows us to create a distributed, scalable system whose actions are based on local information and decisions only. In this chapter we present an introduction to graph theory, graph grammars, as well as previously introduced approaches to self-reconfiguration. Note, however, that this chapter is not meant to be a thorough review on graph theory and graph grammars. Only concepts that are required to represent the structure of our system and interactions between individual agents are covered. Additionally, we present an in-depth treatment of previous work in the area of self-reconfiguration and various approaches research teams have tried to solve this problem.

2.1 Graph Theory

Graph theory is the study of graphs, which can be thought of mathematical abstractions of networks. We present our system as a labeled graph so that we can apply graph grammatical concepts such as the notion of connectivity to it. This section

explains the concepts of graph theory required for the representation of our system.

2.1.1 Graphs

A graph is an abstraction of a networked system that contains no information about the details of interactions between agents. All a graph specifies is the structure and topology of a system. Its two main components are vertices and edges. Vertices represent agents or modules of a system and store data about them and their internal states. Edges describe communication links or physical connections between modules. In a graph, an edge represents a connection between two vertices that can communicate with each other. Edges can either be directed or undirected, meaning that the flow of information is unidirectional or bidirectional. In our system, we will utilize bidirectional edges. Therefore, if agent i can communicate with agent j, j can communicate with i as well. Another important characterization of a graph is whether it is static or dynamic. In a static graph, the edge set will not change over time and the topology is fixed. In a dynamic system like our representation of a self-reconfigurable robot, however, edges are disappearing and reappearing because modules are moving around and the topology of the corresponding graph is constantly changing. So, while the vertex set of our system is time-invariant (unless modules fail), the edge set and therefore the topology of our network of agents changes over time. A graph can be formally defined as follows.

Definition 1. *A labeled graph G is defined as a set of labeled vertices and edges of the form $G = (V, E, l)$, where V is a finite or infinite set of vertices, E is a set of unordered pairs connecting two elements of V and l is a labeling function $l : V \to \Sigma$ that assigns labels to vertices. Σ is the alphabet that labels can consist of. More formally, $V = \{v_1, \ldots, v_N\}$, where N is the total number of nodes in the graph and E is the edge set defined as $E \subseteq V \times V$, with $e_{i,j} \in E$ if there exists a connection between v_i and v_j. For a static graph, E and l are time-invariant, for a dynamic graph, $E(t)$*

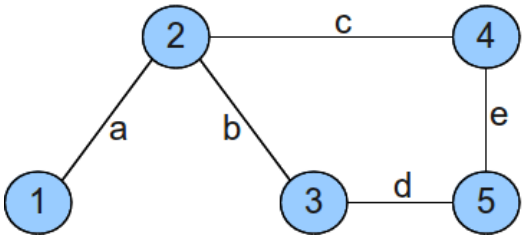


Figure 1: Example of a labeled graph. Vertex labels are represented by integers, edge labels by letters.

and $l(t)$ are time-varying. An example of a labeled graph is shown in Fig. 1.

In this work, a graph is a model of the network topology of an interconnected set of cubic modules. Each cube is represented by a labeled vertex in the graph and an edge exists in the graph where a cube is adjacent to a neighboring cube. A detailed description of this adjacency notion is given in Section 3.1.

2.1.2 Representation of a Graph

The previous section showed the representation of a graph as a vertex set V and an edge set E. Another useful representation employs matrices, specifically adjacency matrices $A(G)$, degree matrices $\Delta(G)$, and the graph Laplacian $\mathcal{L}(G)$.

For an undirected graph, the adjacency matrix $A(G)$ is symmetric and encodes the adjacency information of the network. $A(G)$ is defined as follows:

$$[A(\mathcal{G})]_{ij} = \begin{cases} 1 & \text{if } e_{i,j} \in E \\ 0 & \text{otherwise} \end{cases}$$

The degree matrix $\Delta(G)$ is a diagonal matrix that contains the vertex-degrees of G along its diagonal. The vertex-degree is the cardinality of the neighborhood $\mathcal{N}(v_i)$ of vertex v_i, i.e. the number of neighbors adjacent to vertex v_i.

The graph Laplacion $\mathcal{L}(G)$ is another representation of a graph and is defined as $\mathcal{L}(G) = \Delta(G) - A(G)$. This matrix is the basis of algebraic and spectral graph theory. It allows us to calculate the connectivity of a graph as well as the number of

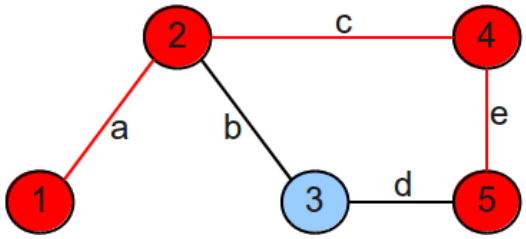


Figure 2: Example of a path from node 1 to 5 (shown in red)

connected components, two concepts that we use extensively in this work and that will be explained next.

2.1.3 Connectivity of a Graph

A graph is said to be connected, if a path exists between any two vertices in the graph. A path in this context refers to a set of vertices $\{v_1, v_2, \ldots, v_m\}$, such that for $k = 0, 1, \ldots, m-1$ the vertices v_k and v_{k+1} are adjacent. The vertices v_0 and v_m are said to be the endpoints of the path (see Fig. 2). To describe connectivity in mathematical terms, the connectivity $c(G)$ of a graph $G = (V, E, l)$ has been defined as the number of connected components of the graph G. A connected component is a subgraph G_S of G that is connected, but not connected to other subgraphs of G. A connected graph features a connectivity of $c(G) = 1$. In other words there is only one connected component in a connected graph.

This property can be shown by analyzing the eigenvalues of $\mathcal{L}(G)$, specifically the number of zero eigenvalues of $\mathcal{L}(G)$. Every zero eigenvalue corresponds to one connected component of the graph. In other words, a connected graph has only one zero eigenvalue, i.e. $\lambda_i > 0$ for $i > 1$ and $c(G) = 1$. For our system, we need to maintain connectivity at all times, which is why G is only allowed to have one connected component, $c(G) = 1$, and $\lambda_i > 0$ for $i > 1$.

2.2 Graph Grammars

Graph grammars were first introduced in the late 1970s and are a generalization of the standard linear grammars used in automata theory and linguistics (see [16] or [58]). They offer a tool for manipulating multidimensional data in graph form and can be seen as a graph-rewriting tool since they enable the parsing, generation, and manipulation of graphs. Graph grammars can be used to describe and control changes in the network topology through rules. Each rule r in a graph grammar or ruleset Φ is composed of two labeled graphs g_l and g_r and is of the form $g_l \xrightarrow{r} g_r$. If the local topology of a module, i.e. its neighborhood structure, in the system matches g_l, then the module rewrite its states and updates its local topology to match g_r. Because graph grammar rules require local information only and can be applied by multiple modules concurrently, they allow the manipulation and coordination of a large number of objects (see [32] and [31]). Depending on the ruleset design, the dynamics resulting from the application of a graph grammar can be nondeterministic or deterministic and concurrent or sequential. The graph grammars we generate as outlined in Section 3.4 are designed in such a way that interactions happen sequentially and deterministically. Graph grammars have been applied to a wide range of problems such as term graph rewriting, DNA computing, distributed algorithms and scheduling problems, software architectures and evolution, or visual modeling of behavior and programming. Most recently, graph grammars have also been applied to self-reconfigurable robotics for example in [32] or [31].

This section introduces graph grammar theory and presents the concepts and definitions used in this thesis. An in-depth treatment of graph grammars can be found in [16], [64], and [17]. Previous work on graph grammars is presented in Section 2.3.2. The following definitions are based on [32].

Definition 2. *A production rule or simply a rule consists of two labeled graphs (as*

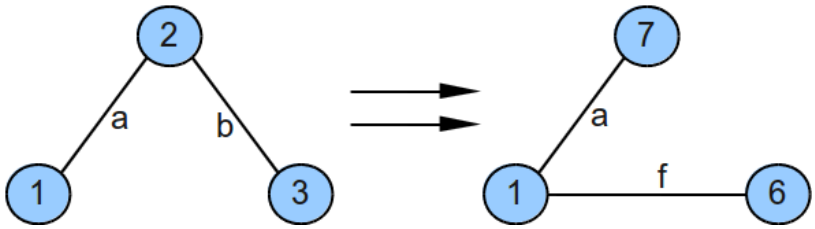


Figure 3: Example of a rule. Shown are the left-hand side and the right-hand side. Note how the rule changes vertex labels, edge labels, and the edge structure.

defined in Def. 1), 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. An example of a rule is shown in Fig. 3.

Basically, a rule describes how the states and local topology of a subset of vertices $S \subset V$ changes. The size of the rule is a measure of "how local" the rule is. A graph grammar is a set of production rules that operate on a graph G_0. Therefore, we call the pair (G_0, Φ) a system, where G_0 is an initial labeled graph and Φ is a graph grammar. A rule $r \in \Phi$ can be applied to G_0 only when it is applicable:

Definition 3. *A rule is applicable to G if there exists a subgraph G_S of G that is isomorphic to g_l, which is also denoted as $G_S \cong g_l$. A graph G_S is said to be isomorphic to g_l if both the labels and the edge structure are equivalent.*

Note that the applicability of rules only depends on the labels of the vertices $v_i \in V$ and not on the underlying vertex IDs in the graph.

Definition 4. *The application of a rule r yields a new graph G_{k+1} that results from G_k by replacing a subgraph of $G_k \cong g_l$ with g_r. g_l and g_r are the left-hand side and the right-hand side of the rule r, respectively. Given a graph $G_k(V, E, l)$, the application of a rule r to G_k yields a new graph $G_{k+1}(V, E', l')$, i.e. $G_k \xrightarrow{r} G_{k+1}$.*

Fig. 4 shows a graphical representation of a rule application on a host graph. The identified subgraph matching the left-hand side of the rule shown in Fig. 3 is highlighted. In other words, the rule shown in Fig. 3 is applicable to the highlighted

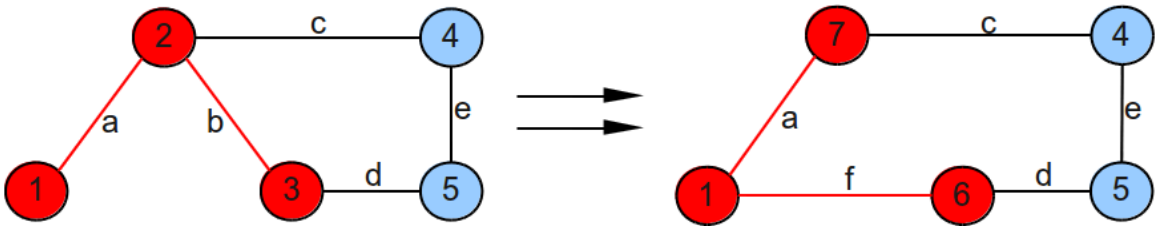


Figure 4: Example of a rule application. The subgraph matching the left-hand side of the rule shown in Fig. 3 is highlighted.

subgraph according to Def. 3. In the above definition, the vertex set V remains the same upon the application of a rule. Only the edge structure and the labels of the vertices change. For the purpose of self-reconfiguration, 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 $G_i \xrightarrow{r_i} G_{i+1}$. The set of all trajectories is denoted as $\mathcal{T}(G_0, \Phi)$ and the i^{th} graph of $\sigma \in \mathcal{T}$ as G_i. An example of a valid trajectory can be seen in Chapter 4, e.g. in Fig. 15. Each graph G_i as part of a trajectory is called reachable by the system (G_0, Φ).

Definition 5. *A graph G is reachable by the system (G_0, Φ) if there exists a finite trajectory $\sigma \in \mathcal{T}(G_0, \Phi)$ such that $G \cong \sigma_k$ for some k. A reachable graph can be temporary, i.e. some rule in Φ operates on part of it, or stable.*

Definition 6. *A graph G is stable, if no rule in Φ can alter it. Note, however, that this doesn't mean that no more rules are applicable to G, merely that it is left unchanged by the application of any further rules.*

The goal of this thesis is to generate a graph grammar Φ in such a way that, starting with a system (G_0, Φ), the reconfiguration leads to a stable graph via a valid trajectory obeying given constraints (see Section 3.3). We designed our algorithms such that the stable set only contains the graph derived from the target configuration $\mathcal{C}^{\mathcal{T}}$. The rule generation and a proof that this goal is indeed achieved by our generated

graph grammars is given in Section 3.4.2.

2.3 Self-Reconfiguration Approaches

Self-reconfiguration of modular robots is not a new field. Researchers have been working in this area for over three decades and came up with a variety of approaches to solve the self-reconfiguration problem. A lot of work has been done on the planning aspect of self-reconfiguration. Available planning strategies include hierarchical or layered planning, rule-based planning, Markov decision process-based planning, and graph signatures-based planning. This section provides a brief overview of the state of the art in self-reconfigurable robotics.

2.3.1 Hierarchical Planning and Reconfiguration

Hierarchical planning for self-reconfiguration decomposes the problem into multiple layers. [36] introduced a planner that operates on three levels: trajectory planning for individual modules, configuration planning to reconfigure the whole structure, and task-level planning. Similar to our work, their algorithms compute the movable and reachable set. Additionally, they employ a scaffold planning approach to match module positions in the initial to those in the target configuration. [68] presents a two-layered planning approach. The upper layer decomposes the planning problem into smaller subproblems. The lower layer then solves these subproblems using a rule database whose rules are based on connectivity information and do not use labels. [57] uses a similar layered approach, in which the top layer calculates paths for blocks of cubes, another layer calculates paths for individual cubes, and the lowest layer determines link actions. [69] and [68] also introduced hierarchical reconfiguration planning and use rules for their lowest level of motion selection.

2.3.2 Rule-based Reconfiguration

As opposed to hierarchical planners, which almost exclusively plan module paths in a centralized way with global knowledge, rule-based reconfiguration is a decentralized and distributed approach to self-reconfiguration. Previous work on rule-based self-reconfiguration can be broadly grouped into two categories: manually defined rulesets and automatically generated rulesets.

[19] accomplished self-reconfiguration with a manually defined ruleset and shows how an intermediate configuration can be used in the reconfiguration process. [10] and [12] 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. Their approach allows the instantiation of their algorithms to a wide range of systems, for example the M-TRAN system (see [42]). Other work on manually defined rulesets includes [10], [12], and [68], which have demonstrated the feasibility and scalability of rule-based self-reconfiguration. Contrary to our automatically generated rulesets, the rulesets in these papers do not contain graph grammar rules, since no labels are used.

The second category of automatically generated rulesets is represented by [25] and [8]. [25] applies the rules to a simulated two-dimensional structure The rules are automatically generated and only use connectivity information to check for rule applicability. Since no additional labels are used to control the reconfiguration, multiple rules can potentially be applicable at the same time. [8] introduces a rule-based control strategy for the ATRON system (see [7]). The rules are automatically generated and take connectivity information into account. They introduce wild card rules to reduce the size of the ruleset. This paper tries to solve two main problems with rule-based reconfiguration. On one hand, the complexity of defining rules increases faster than the complexity of the desired behavior. And on the other hand, with an increasing number of rules, the probability of conflicting rules increases. Our approach solves both

of these problems by automatically generating graph grammar-based rules instead of rules based on connectivity information only.

Graph grammars, as a tool for manipulating multidimensional data, have also been applied to modular robotics. [31] and [6] show the feasibility of graph grammars by reconfiguring programmable parts, a triangle shaped hardware implementation. The authors use a manually defined ruleset that is designed to form specific structures out of the triangular modules. As opposed to our system, [31] allows multiple rules to be applicable to the whole system at the same time. This means that unlike for our system, the approach shown in this paper does not guarantee a unikely determined reconfiguration sequence or the reaching of the target configuration
A compact overview of graph grammatical concepts and definitions is given in [17]. Graph grammars provide a tool for achieving fine-grained control of the reconfiguration process by using labeled graphs to describe local states. Additionally, graph grammars allow to avoid problems such as multiple applicable rules. To the best of our knowledge automatically generated graph grammars for self-reconfiguring modular robots is a novel approach to self-reconfiguration.

2.3.3 Various Approaches

Many other approaches have been presented in the literature, some of which include the following: [66] shows reconfiguration algorithms for the Proteo system. This paper describes a distributed control system, in which each module acts as an independent agent and determines whether it can move toward the target position or not based on local criteria and information about the initial and the target configuration. Paths are not pre-computed but planned online. The approach presented in this paper assumes that the order, in which the modules are moved, is known a-priori and that agents have incomplete and delayed information about the global configuration. Additionally, this approach does not guarantee the reaching of the desired target configuration.

[18] formulates the reconfiguration problem as Markov decision process to reconfigure a lattice-based robot. According to the paper a solution is obtained in sublinear time. Their algorithms can handle three-dimensional homogeneous configurations of modules up to size 75^3 and map actions to lattice positions instead of modules. Because an optimal action is associated with each lattice position, the same action is applied to every module at a certain position. Therefore, this approach can only handle homogeneous systems. Furthermore, this approach is specifically designed for the locomotion of self-reconfigurable systems. The approach presented in this thesis, on the other hand, can compute arbitrary reconfiguration sequences, which includes locomotion as a special case.

2.3.4 Implementations

Several hardware platforms have been designed for algorithm verification and feasibility studies, for example the SuperBot [61], the Ubot [63], the YAMOR [52], the ATRON [7], or the M-TRAN platform [42]. On the other hand, various simulators have been designed to facilitate modular robotics research. Examples include the USSR (Unified Simulator for Self-Reconfigurable Robots) [15], Gazebo [34], Player/Stage [21], [22], and URBI [3].

CHAPTER III

SELF-RECONFIGURATION

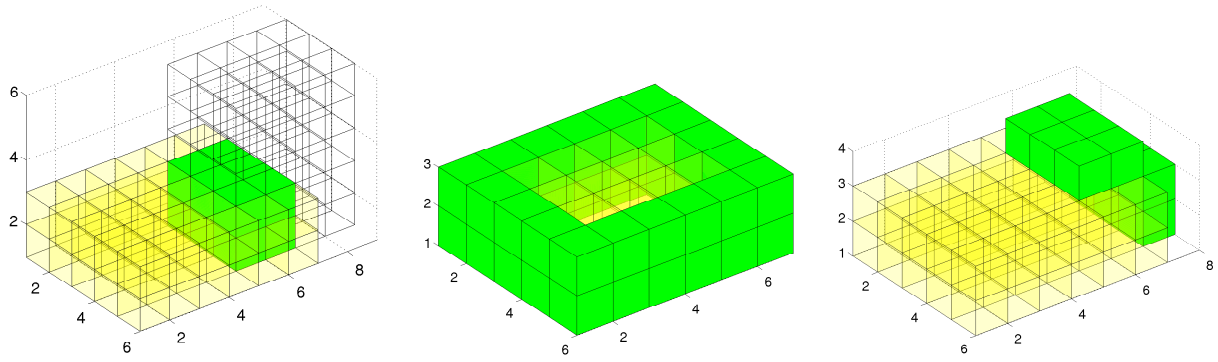
This chapter describes the representation of our system and the reconfiguration planning approach in detail. We outline the sliding cube model as the basis for our system representation and present the mathematical description of our system. Next, we give an overview of the reconfiguration process and the individual components of our algorithm. That is followed by a discussion of the assignment and path planning stage, as well as a description of constraints that apply to the reconfiguration. Lastly, the automatic rule generation as the main contribution of this thesis is presented in depth.

3.1 System Representation

In this thesis we investigate a modular robotic system whose basic building blocks are visually represented by cubes (see Fig. 5). Moreover, no physical constraints 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. In chapter 4 we present one reconfiguration in obstacle-constrained space to show the straightforward extensibility of our approach.

3.1.1 Sliding Cube Model

Following the taxonomy in [65], modular robots can generally be categorized into lattice-type and chain-type architectures. We present a lattice-based system that is



(a) Initial (transparent) and target (transparent) configuration and overlapping nodes (opaque)

(b) Movable nodes (opaque) as part of the initial configuration (transparent)

(c) Immediate target successor positions (opaque) as neighboring positions of the initial configuration (transparent)

Figure 5: Graphical representation of the overlapping, the movable, and the immediate target successor set in the simulator

embedded in a discrete coordinate system using the sliding cube model (see [19]). The sliding cube model is an abstraction that greatly simplifies development of algorithms and can be instantiated to various platforms. Every module (also referred to as node) is represented as a cube with dimension δ (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 features connectors on each surface and is capable of executing motions. A motion can be generally described as a function $f(x_i, m) = x_i + m$ 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 convex transitions to orthogonal surfaces (see [38]).

Definition 7. *A sliding motion is defined as a function* $f(x_i, m_s) = x_i + m_s$ *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* $\mathcal{M}_s = \{m \in \mathbb{Z}^3 | m_x = \delta \vee m_y = \delta \vee m_z = \delta \wedge m_x + m_y + m_z = \delta\}$.

An example of a sliding motion would be $f(x_i, m_s) : (x_{i,x}, x_{i,y}, x_{i,z}) \xrightarrow{m_s} (x_{i,x} + \delta, x_{i,y}, x_{i,z})$.

Definition 8. *A convex motion is defined as a function* $f(x_i, m_c) = x_i + m_c$ *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 convex motions and is defined as* $\mathcal{M}_c = \{m \in \mathbb{Z}^3 | m_x < 2\delta \wedge m_y < 2\delta \wedge m_z < 2\delta \wedge m_x + m_y + m_z = 2\delta\}$.

An example of a convex motion would be $f(x_i, m_c) : (x_{i,x}, x_{i,y}, x_{i,z}) \xrightarrow{m_c} (x_{i,x} + \delta, x_{i,y} + \delta, x_{i,z})$. Note, however, that we will not allow the application of sliding or convex motions unless they are feasible. In fact, the movement of individual cubes requires a connected substrate of other cubes. Such a connected arrangement is referred to as a configuration, i.e. a configuration describes a geometric arrangement of cubes. The representable space of our system is $\mathbb{Z}^{3N}$ and any configuration $\mathcal{C}$ is a subset of the representable space, $\mathcal{C} \subset \mathbb{Z}^{3N}$.

3.1.2 Adjacency Matrix Notation

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

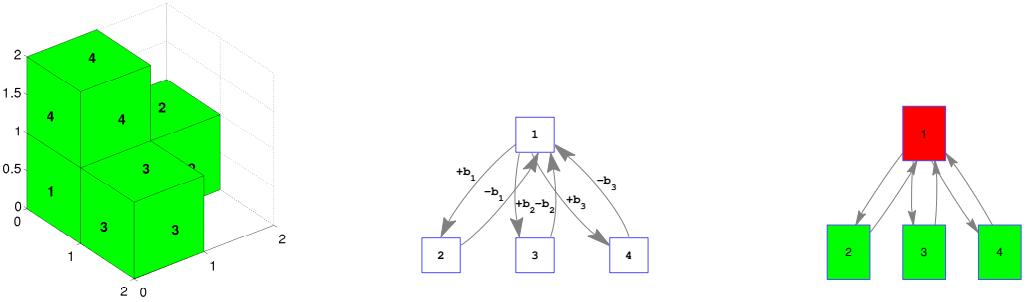
$$A_k = [a_{i,j,k}] = \begin{cases} 1 & \text{if } (x_i - x_j)^T \cdot b_k = 1 \\ -1 & \text{if } (x_i - x_j)^T \cdot b_k = -1 \\ 0 & \text{otherwise} \end{cases}$$

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

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. The advantage of our adjacency representation is that we can encode the vertex set as well as the edge set of the represented graph in one data structure. Just the labels of each vertex have to be stored separately. An example of the adjacency representation is shown in Fig. 7(a) and Fig. 6.

$$A_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} A_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} A_3 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 6: Adjacency matrices for the configuration in Fig. 7(a)



(a) Representation of the adjacency matrices shown in Fig. 6

(b) Graph $G = (V, E, l)$ represented by the adjacency matrices in Fig. 6

(c) Connectivity graph of the configuration in Fig. 7(a) with articulation points (highlighted)

Figure 7: Graphical representation of the adjacency matrices in Fig. 6 and the corresponding graph.

3.1.3 Graph Notation

Alternatively, a configuration can be represented as a labeled graph $G = (V, E, l_G)$ where G is composed of a vertex set V, an edge set E, and a label set l_G. This representation is required for the rule generation presented in Section 3.4.

Definition 9. *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 $h(A_k, l) = (V, E, l_G)$. V is a finite set of integers corresponding to the cube IDs, i.e. $V = \{1, \ldots, N\}$, 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:*

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

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

Here, $i, j \in \{1, \ldots, N\}$, $l(c_i) \in l$, and b_k is a base vector. The vertex labels of $v_i \in G$ are the same as the labels for the cubes $c_i \in \mathcal{C}$.

A graphical representation of a graph G is shown in Fig. 8. G is a directed, labeled graph that preserves the three-dimensional structural information of the configuration. Whereas this representation is required for rule generation and application, most other functions do not need three-dimensional information. Therefore, we use a reduced version of G that only store the connectivity information.

Definition 10. *The connectivity graph $G_c = (V, E, l_{G_c})$ contains a vertex set V and an edge set E, which are the same as before and the labels l_{G_c}, which are defined as follows: $l(v_i) = l(c_i)$ for $v_i \in V$ and $c_i \in \mathcal{C}$, i.e. the labels of nodes $v_i \in G_c$ are the same as the labels of cubes $c_i \in \mathcal{C}$. The connectivity graph does not contain any edge labels and stores connectivity information in only one adjacency matrix A, which is defined as*

$$A_{i,j} = |sign(\sum_{k=1}^{3} |A_{i,j,k}|)| \quad i, j \in \{1, \ldots, N\}$$

An example of a connectivity graph is shown in Fig. 9(b).

We assume that the initial configuration $\mathcal{C}^{\mathcal{I}}$ and the target configuration $\mathcal{C}^{\mathcal{T}}$ are known and contain the same number of modules. We employ a two-stage planning process. In stage one, our algorithm finds the initially overlapping region $\mathcal{O}_{\rangle\backslash\rangle\sqcup}$ of both $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ and then calculates a path for every node $c_i \in \mathcal{C}^{\mathcal{I}} \setminus \mathcal{O}_{\rangle\backslash\rangle\sqcup}$ to a position

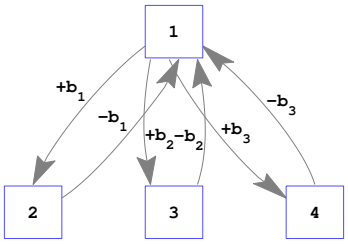
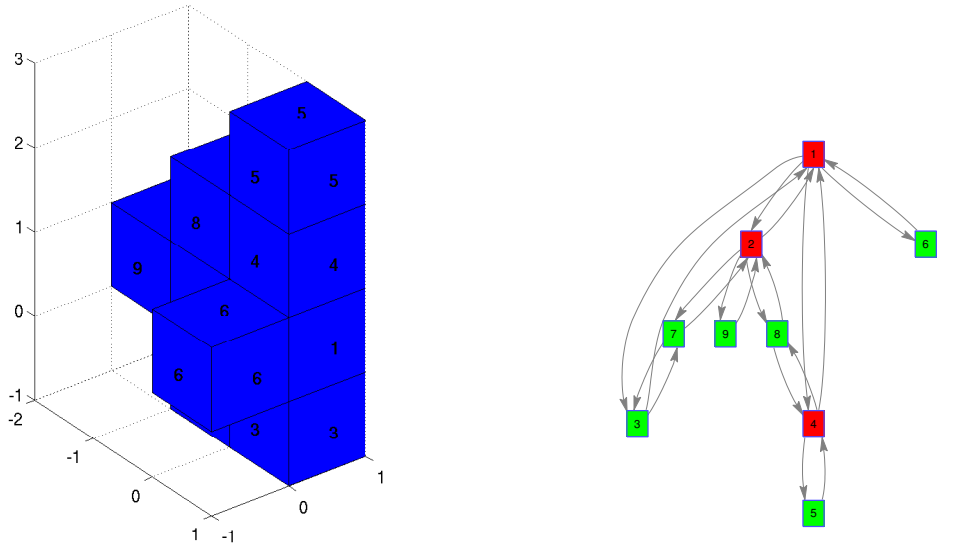


Figure 8: Graph $G = (V, E, l)$ represented by the adjacency matrices in Fig. 6



(a) Randomly generated configuration

(b) Connectivity graph of the configuration in Fig. 9(a) with articulation points (highlighted)

Figure 9: Configuration representation in the simulator

$c_j \in \mathcal{C}^{\mathcal{T}}$. Furthermore, a ruleset or graph grammar is generated from these paths. In stage two, each node then executes rules that can be checked locally for applicability. Local in this context means that each rule describes a neighborhood of the current cube and can only manipulate cubes in that neighborhood. The rule execution is done in a decentralized way during which each cube can just access neighborhood information and the ruleset.

3.2 Assignment

The reconfiguration process requires us to move cubes from their initial positions to their target positions. Therefore, we have to calculate paths for cubes $c_i \in \mathcal{C}^{\mathcal{I}}$ to their desired positions in $c_j \in \mathcal{C}^{\mathcal{T}}$. This section describes the assignment of a $c_i \in \mathcal{C}^{\mathcal{I}}$ to a position $c_j \in \mathcal{C}^{\mathcal{T}}$, which includes the computation of the overlapping set, the movable set, the immediate target successor set, as well as the actual assignment.

3.2.1 The Overlapping Set

To reduce planning time, we first determine cubes $c_i \in \mathcal{C}^{\mathcal{I}}$ that already occupy positions $c_j \in \mathcal{C}^{\mathcal{T}}$. These cubes are said to be in the overlapping region of $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ and do not have to be moved to the target configurations. Cubes c_i in the initially overlapping region $\mathcal{O}_{init} = \mathcal{C}^{\mathcal{I}} \cap \mathcal{C}^{\mathcal{T}}$ (see Fig. 5(a)) are therefore excluded from the planning process.

Definition 11. *The overlapping set of cubes $\mathcal{O}$ is defined as $O = \mathcal{C} \cap \mathcal{C}^{\mathcal{T}}$. In other words, every cube c_i that is both part of the current and the target configuration is in the set of overlapping cubes (see Fig. 5(a)).*

Note that O is the overlap of the current configuration $\mathcal{C}$ and the target configuration $\mathcal{C}^{\mathcal{T}}$. Therefore, it has to be recalculated after every reconfiguration step. The initial overlapping region is furthermore denoted as O_{init}.

3.2.2 The Movable Set

The movable set $\mathcal{M}$ contains all cubes c_i of the current configuration $\mathcal{C}$ that can be relocated without disconnecting the configuration. Before we can define the movable set, we need to introduce the notion of articulation points.

Definition 12. *An articulation point in a graph is a node $v \in V$ whose removal would disconnect the graph and reduce the rank of the associated graph Laplacian (see Fig. 9(b)). In other words, the removal of an articulation point from the graph would increase the number of connected components $c(G)$, i.e. $c(G - v) > c(G)$.*

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 of the connectivity graph is never moved. The movable set is therefore defined as follows:

Definition 13. *The movable set $\mathcal{M}$ is a set of cubes that can be moved without disconnecting the configuration and is defined as $\mathcal{M} = \{c_i \in \mathcal{C} | c_i \in \mathcal{C}^{\mathcal{I}} \setminus \mathcal{O} \wedge c_i \notin \mathcal{A}(\mathcal{C}) \wedge |\mathcal{N}_1(c_i, \mathcal{C})| \leq 5\}$. $\mathcal{N}_1(c_i, \mathcal{C})$ is the one-hop neighborhood and defined as $\mathcal{N}_1(c_i, \mathcal{C}) = \{c_j \in \mathcal{C} | dist(c_i, c_j) = 1\}$. $\mathcal{A}(\mathcal{C})$ is the set of articulation points of the graph representing the current configuration $\mathcal{C}$ (see Fig. 7(c)).*

This definition is based on the sliding cube model (see [19]) and only allows modules on the surface of the configuration to be relocated. This is because Def. 13 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 $\mathcal{R}$.

3.2.3 The Immediate Target Successor Set

The immediate target successor set $\mathcal{R}$ contains all positions c_j in the neighborhood of the current configuration $\mathcal{C}$ that cubes can be relocated to without disconnecting the configuration or violating other constraints (see Section 3.3.2). According to the sliding cube model (see [19]) and to the following definition, $\mathcal{R}$ contains only positions c_j on the surface of $\mathcal{C}$.

Definition 14. *The immediate target successor set $\mathcal{R}$ is defined as positions $c_j \in \mathcal{C}^{\mathcal{T}}$ 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 $\mathcal{C}^{\mathcal{T}}$. $\mathcal{R} = (\mathcal{C}^{\mathcal{T}} \cap \mathcal{N}_1(\mathcal{C})) \setminus \mathcal{C}$, where $\mathcal{N}_1(\mathcal{C}) = \{c_j | c_j \notin \mathcal{C} \wedge c_i \in \mathcal{C} \wedge dist(c_i, c_j) = 1\}$ (see Fig. 5(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. We know by assumption that $\mathcal{R}$ is nonempty unless the target configuration has already been assembled. The immediate target successor set $\mathcal{R}$ consists of reachable positions c_j, which refer to

individual cube positions and not to a reachable, stable graph as defined for graph grammars in Def. 6.

3.2.4 Assignment

Both $\mathcal{M}$ and $\mathcal{R}$ define a set of cubes. Before we can plan a path for a cube $c_i \in \mathcal{M} \wedge c_i \in \mathcal{C}^{\mathcal{I}}$ to a position $c_j \in \mathcal{R} \wedge c_j \in \mathcal{C}^{\mathcal{T}}$, 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 c_i and c_j with the smallest cost. In a case where two or more assignments have the same cost, we pick an assignment randomly. This approach is also called a greedy approach and has been used in the literature for homogeneous reconfiguration before (see [19]). A path is then calculated between cube $c_i \in \mathcal{M}$ and its assigned target position $c_j \in \mathcal{R}$.

3.3 Path Planning

This section describes the path planning approach we have used to compute paths for the assigned pair of cubes $c_i \in \mathcal{C}^{\mathcal{I}} \cap \mathcal{M}$ and $c_j \in \mathcal{C}^{\mathcal{T}} \cap \mathcal{R}$. We also formally define paths, the planning space, and describe constraints on the path planning. The input for the path planning stage is the assignment of the previous assignment stage that determines which cube $c_i \in \mathcal{C}^{\mathcal{I}}$ should be moved to which position $c_j \in \mathcal{C}^{\mathcal{T}}$. The output of the planning stage is a path that is then used as input for the ruleset generation discussed in Section 3.4.2.

We plan a path p_i for a single cube at a time, i.e. from $c_i \in \mathcal{M}$ to $c_j \in \mathcal{R}$. The path p_i is only allowed to contain positions in the planning space and use primitive motions to move the current cube c_i.

Definition 15. *A path is a concatenation of motions $m \in \{m_c, m_s\}$ (see Def. 7 and Def. 8) that move cube $c_i \in (\mathcal{C}^{\mathcal{I}} \setminus \mathcal{O}) \cap \mathcal{M}$ to position $c_j \in \mathcal{C}^{\mathcal{T}} \cap \mathcal{R}$. The length of the path p_i, or the total number of motions it contains, is denoted as $|p_i|$.*

Path planning requires the completion of two steps: the computation of the planning space and the actual search. The computation of the planning space is described in detail in the following section. The search through the planning space uses common A*, a best-first search employing the Manhattan distance as heuristics to guide the search to a solution. A heuristic representative of the search problem makes A* faster than uninformed search strategies such as depth-first or breadth-first search (see Section 3.3.3).

3.3.1 Planning space

The planning space $\mathcal{P}(\mathcal{C})$ available to the path planner is defined as $\mathcal{P}(\mathcal{C}) = \{c_i \in \mathcal{C} | c_j \notin \mathcal{C} \wedge dist(c_i, c_j) = \delta\}$. In other words, $\mathcal{P}(\mathcal{C})$ is the search space for the path planner and represents the space a cube c_i can move through during relocation from its initial position to the target position. Since $\mathcal{P}(\mathcal{C})$ is calculated as the one-hop neighborhood it includes the hull around $\mathcal{C}$ as well as any tunnels through the configuration. Tunneling is also allowed in [12], but in this paper tunnels have to be accounted for explicitly in the ruleset whereas in our approach tunnels are implicitly included in the planning space.

$\mathcal{P}(\mathcal{C})$ depends on the current configuration $\mathcal{C}$, and therefore needs to be updated whenever $\mathcal{C}$ changes, i.e. after each path is computed. This approach also allows for the straightforward inclusion of obstacles into the path planning. Obstacles $\mathcal{C}_{Obstacles}$ are represented the same way as a configuration - as a set of cubes. Therefore, to make our system aware of obstacles, we remove every $c_j \in \mathcal{P}(\mathcal{C})$ that is occupied by an obstacle, i.e. $c_j \in \mathcal{P}(\mathcal{C}) \cap \mathcal{C}_{Obstacles}$. In other words, we redefine $\mathcal{P}(\mathcal{C})$ as $\mathcal{P}_r(\mathcal{C}) = \{c_j | c_i \in \mathcal{C} \wedge c_j \notin (\mathcal{C} \cup \mathcal{C}_{Obstacles}) \wedge dist(c_i, c_j) = \delta\}$. If an assignment $a = \{c_{init}, c_{target}\}$ is given, the planning space has to be modified to exclude certain positions in the neighborhood $\mathcal{N}_1(c_{init}, \mathcal{C})$. These are the positions, which c_{init} could reach via primitive motions, but which would disconnect the structure if c_{init} actually

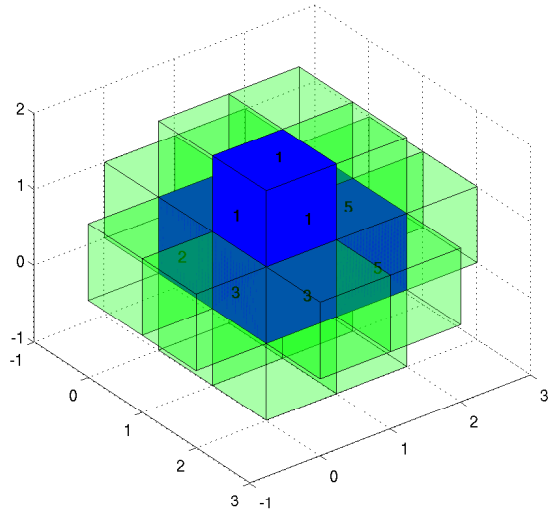


Figure 10: Example showing the planning space for a random configuration. Notice how certain positions adjacent to Node 1 are not part of the planning space.

moved there. The planning space can therefore be redefined as follows $\mathcal{P}_r(\mathcal{C}) = \{c_j | c_i \in \mathcal{C} \wedge c_i \neq c_{init} \wedge c_j \notin (\mathcal{C} \cup \mathcal{C}_{Obstacles}) \wedge dist(c_i, c_j) = \delta\}$ (see Fig. 10).

3.3.2 Constraints

This section outlines the constraints, which apply to the path planner and which guarantee that the configuration remains connected at all times. A path that obeys all constraints is called a feasible paths.

Motion constraints This constraint states that the path planner is only allowed to use the two primitive motions defined in Def. 7 and Def. 8 to move cubes from $c_i \in \mathcal{C}$ through the planning space $\mathcal{P}(\mathcal{C})$ to its target position $c_j \in \mathcal{C}^{\mathcal{T}}$. Any other motion would violate the motion constraints and result in an infeasible path. An example of possible primitive motions for a cube is shown in Fig. 11.

Connectivity constraints This constraint assures that the configuration remains connected at all times, i.e. that the connectivity $c(G) = 1$. In other words, we need to make sure that no motion of any cube disconnects the configuration.

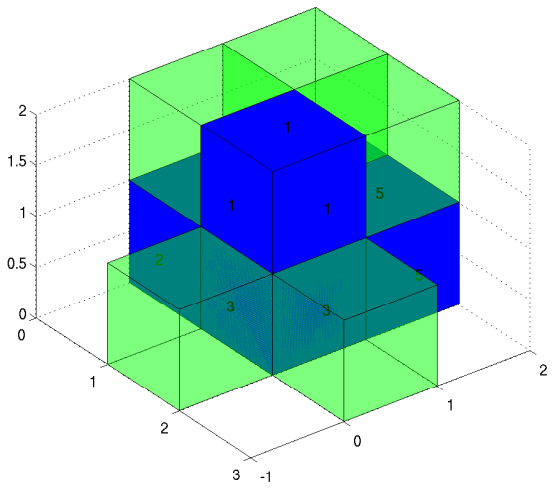


Figure 11: Example showing possible primitive motions for Node 1.

Theorem 1. *The configuration $\mathcal{C}$ remains connected at all times during reconfiguration if the initial configuration $\mathcal{C}^{\mathcal{I}}$ is connected.*

Proof. A configuration $\mathcal{C}$ can only become disconnected if a cube c_i executes a primitive motion that violates the connectivity constraint. Cubes always move along paths, which are planned in the planning space $\mathcal{P}(\mathcal{C})$. By construction of the planning space $\mathcal{P}(\mathcal{C})$ (see Section 3.3.1) and the fact that every position $c_j \in p_i$, where p_i is path i, is also $c_j \in \mathcal{P}(\mathcal{C})$, no path will contain a motion violating the connectivity constraint. Additionally, the assignment $a = \{c_{init}, c_{target}\}$ is chosen such that moving c_{init} does not disconnect the configuration and such that $c_{target} \in (\mathcal{P}(\mathcal{C}) \cap \mathcal{R}$ (see Section 3.2. Therefore, the configuration $\mathcal{C}$ will remain connected during reconfiguration, if $\mathcal{C}^{\mathcal{I}}$ is connected. □

Local blocking constraints This constraint ensures that no holes or enclosures are formed during the reconfiguration of the structure. Any position $c_j \in \mathcal{R}$ (as defined in Def. 14) that is a blocking position is removed from $\mathcal{R}$, where blocking positions are defined as follows.

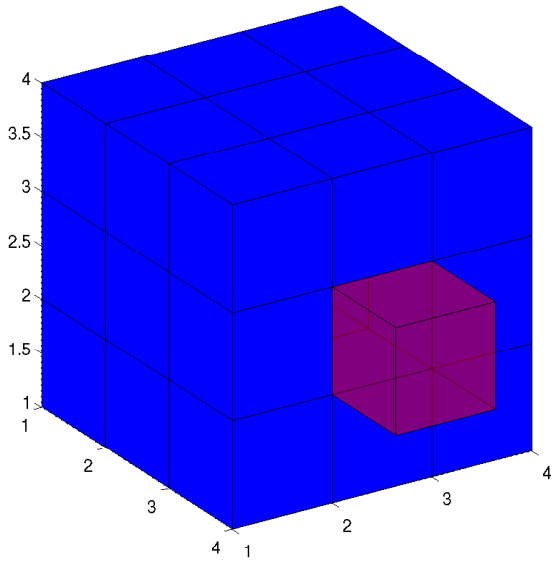


Figure 12: Example showing a blocking position (highlighted) for a hollow cube.

Definition 16. *A position* $c_b \in (\mathcal{C}^{\mathcal{T}} \cap \mathcal{R})$ *is said to be a blocking position if any of its neighbors* $c_k \in \mathcal{N}_1(c_b, \mathcal{C})$ *violate the following conditions.*

$$\begin{aligned} |\mathcal{N}_1(c_k, \mathcal{C})| < 5 \qquad & \text{if } c_k \in \mathcal{N}_{1,occupied}(c_b, \mathcal{C}) \\ |\mathcal{N}_1(c_k, \mathcal{C})| < 5 \qquad & \text{if } c_k \in \mathcal{N}_{1,unoccupied}(c_b, \mathcal{C}) \setminus \mathcal{C}^{\mathcal{T}} \end{aligned} .$$

where $\mathcal{N}_{1,occupied}(c_b, \mathcal{C}) = \{c_j \in \mathcal{C} | dist(c_b, c_j) = \delta\}$ *and* $\mathcal{N}_{1,unoccupied}(c_b, \mathcal{C}) = \{c_j \notin \mathcal{C} | dist(c_b, c_j) = \delta\}$.

In other words, this definition ensures that moving a cube to position c_b does not cause the neighborhood set of any of the neighboring positions to grow larger than 4. This allows us to avoid local blocking of positions. An example of a blocking position is shown in Fig. 12).

Note, however, that this is a local check. It is not equivalent to hole detection as used in graph theory since it checks only the local neighborhood, i.e. $\mathcal{N}_1(c_j, \mathcal{C})$ of a position c_j. Such restricted local knowledge can result in overlooked holes or enclosures in the configuration. For homogeneous reconfiguration with greedy assignment as done in this thesis, local blocking constraints were avoided the generation of

enclosures during reconfiguration experiments.

Planning space constraints Planning space constraints restrict $\mathcal{P}(\mathcal{C})$ as mentioned in Section 3.3.1 and are used to incorporate obstacles into the path planning. Therefore, any position $c_j \in (\mathcal{P}(\mathcal{C}) \cap \mathcal{C}_{Obstacles})$ is removed from $\mathcal{P}(\mathcal{C})$ so that positions occupied by obstacles can not be used for path planning.

3.3.3 Implementation

The path planner uses a best-first search, namely A*, and the Manhattan distance as heuristic. This metric was used because it most accurately represents the discrete lattice structure of our system and the possible movements of the cubes. The input for the path planner is the current configuration $\mathcal{C}$, the planning space $\mathcal{P}(\mathcal{C})$, and a valid assignment pair $a = \{c_{init}, c_{target}\}$ containing an initial cube c_{init} and a target position c_{target}. The output is a set of cube positions describing the path from c_{init} to c_{target}. In this thesis, 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. Other planners can of course be used, but we made this choice due to A*'s properties of optimality and completeness, which ensure us to find the optimal paths in polynomial time assuming that the heuristic meets the requirements defined in [60]. The Manhattan distance we use as heuristic does fulfill these criteria. Optimality can be traded for path length, i.e. there exists a trade-off between planning time and path length. Since for reconfigurable robots, computation is less expensive than motion from an energy consumption perspective, we chose optimality of paths over planning time. The result of the path planning stage is a set of paths that describe the complete reconfiguration from $\mathcal{C}^{\mathcal{I}}$ to $\mathcal{C}^{\mathcal{T}}$. This set of paths is then rewritten into a ruleset as discussed in the next section.

3.4 Rule Generation

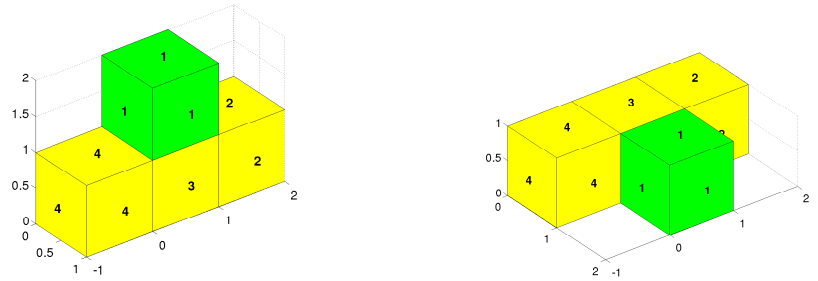
In this thesis, we employ graph grammatical concepts to bridge the gap between global information that is available during planning and local information that is available to the cubes during reconfiguration. The centralized path planning results are rewritten into rules that can be checked locally for 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. In this work, the initial graph is derived from the start configuration $\mathcal{C}^I$ and the reachable stable graph is derived from the target configuration $\mathcal{C}^T$. The production rules of the graph grammar are generated from the calculated paths as shown in Section 3.4.2. This section explains the structure of the automatically generated rules, defines a mapping from cubesets to the graphs we work with, describes the rule generation algorithm, and illustrates the ruleset execution.

3.4.1 Rule Structure

For the purpose of self-reconfiguration, a production rule for our system uses the rule structure shown in Def. 2 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 $\mathcal{N}_2(c_i, \mathcal{C}) = \{c_j \in \mathcal{C} | c_i \in \mathcal{C} \wedge dist(c_i, c_j) \leq \sqrt{2}\}$. $\mathcal{N}_2(c_i, \mathcal{C})$ 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.



(a) Left-hand side of a rule (b) Right-hand side of a rule

Figure 13: Visual representation of a rule that shows a convex motion of cube 1.

```
ID                  : 130
gl_struct           : [1x1 struct]
gl_labels           : '114,130,1,25'
gr_struct           : [1x1 struct]
gr_labels           : '114,131,1,25'
update neighbors    : []
```

Listing 3.1: Rule data structure

Definition 17. *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 $f(\mathcal{C}) = (V, E, l_G)$ with mappings g and h defined in Section 3.1. 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. 13 shows a graphical representation of $\mathcal{N}_2(c_i, \mathcal{C}) = f^{-1}(g_l)$ and $\mathcal{N}_2(c_i + m, \mathcal{C}) = f^{-1}(g_r)$ 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 (see example in Listing 3.1). Since the labels of g_l and g_r in the rules we generate are essential in guaranteeing the properties of our reconfiguration approach, we will present a detailed description of their structure. Listing 3.1 shows

that each label is composed of multiple, comma-separated data fields. These data fields 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* (see fields *gl_labels* and *gr_labels* in Listing 3.1). 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 $G_0 = (V, E, l_G) = f(\mathcal{C}^{\mathcal{I}})$ 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.

3.4.2 Rule Generation

The main contribution of this thesis is the automatic generation of a graph grammar Φ that describes the unambiguous reconfiguration $\mathcal{C}^{\mathcal{I}} \xrightarrow{\Phi} \mathcal{C}^{\mathcal{T}}$. The path planning and the rule generation are interleaved, which means that once a path p_i ($i \in \{1..|P|\}$ where $|P| = |\mathcal{C}^{\mathcal{I}} \setminus \mathcal{O}_{init}| = |\mathcal{C}^{\mathcal{T}} \setminus \mathcal{O}_{init}|$) 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 Φ 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 18. *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*

$G = (V, E, l_G)$ *the following way:*

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

The labels change from $l_G(v_i)$ *to* $l'_G(v_i)$ *as follows:*

$$l'_G(v_i) \rightarrow rule\_id = l_G(v_i) \rightarrow rule\_id + 1$$

Definition 19. *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. Therefore, it does not result in a cube movement in the configuration space. More formally,* r_f *rewrites the graph* $G = (V, E, l_G)$ *the following way:*

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

The labels change from $l_G(v_i)$ *to* $l'_G(v_i)$ *as follows:*

$$l'_G(v_i) \rightarrow rule\_id = l_G(v_i) \rightarrow rule\_id + 1$$
$$l'_G(v_i) \rightarrow flooding = 0$$

Definition 20. *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, i.e. it applies to every node independent of the node ID if all other label fields agree. A propagation rule does not change the edge set and therefore does not result in a cube movement in the configuration space. More formally,* r_p *rewrites the graph* $G = (V, E, l_G)$ *the following way:*

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

The labels change from $l_G(v_i, v_j)$ to $l'_G(v_i, v_j)$ as follows:

$$l'_G(v_i) \rightarrow rule\_id = l_G(v_i) \rightarrow rule\_id + 1$$
$$l'_G(v_i) \rightarrow path = l_G(v_i) \rightarrow path + 1$$
$$l'_G(v_i) \rightarrow flooding = 1$$
$$l'_G(v_j) \rightarrow flooding = 0$$

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 (see example in Listing 3.1, fields *gl_labels* and *gr_labels*). More formally, for each motion m_j ($j \in \{1..|p_i|\}$ as defined in Def. 15) 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 3.4.1. The labels of g_l are defined as follows:

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

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. Such a step is exemplified in Listing 3.1. The application of the shown rule with ID 130 changes the edge set of the immediate motion successor set of node 114 (specified by *gl_struct* and *gr_struct*) and updates its labels such that the next applicable rule is rule number 131 (see field *gl_labels and gr_labels*).

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 $\mathcal{C}^{\mathcal{T}}$ has been assembled. This means that the only reachable, stable graph as defined in Def. 6 is the graph representing the desired target configuration $\mathcal{C}^{\mathcal{T}}$.

Theorem 2. *The graph $G = (V, E, l_G) = f(\mathcal{C}^{\mathcal{T}})$ representing the target configuration $\mathcal{C}^{\mathcal{T}}$ is the only reachable, stable graph to the ruleset Φ.*

Proof. This proof is based on Theorem 1 in [59] 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 $S = \mathcal{N}_1(\mathcal{C}) = \{c_i | c_i \notin \mathcal{C} \wedge c_j \in \mathcal{C} \wedge dist(c_i, c_j) = 1\}$ and $\mathcal{R}$ is a subset of S, $\mathcal{R} \subset \mathcal{S}$. 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 [59] 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 $\mathcal{C}^{\mathcal{T}}$ can be assembled incrementally from $\mathcal{C}^{\mathcal{I}}$. Therefore, every cube $c_i \in \mathcal{C}^{\mathcal{I}} \setminus \mathcal{O}_{init}$ will be moved to its target position $c_j \in \mathcal{C}^{\mathcal{T}}$. 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, i.e. the order in which all cubes are relocated. The outcome of the planning stage, i.e. the execution of paths p_i for $i \in \{1, \ldots, N\}$ with $N = |\mathcal{C}^{\mathcal{I}} \setminus \mathcal{O}_{init}|$, therefore, unambiguously yields $\mathcal{C}^{\mathcal{T}}$.

The rule and path generation are interleaved. After each path p_i has been planned, each motion m_j of p_i is rewritten into a rule $r_{i,j}$ with a globally unique rule number. These rule numbers are unique, strictly monotonically increasing, and are encoded in the labelsets of $r_{i,j}$. As a result, the applicability of rule $r_{i,j}$ depends on the successful execution of rule $r_{i,j-1}$. Therefore, the same sequence of reconfiguration steps is achieved as in the planning stage and the execution of the ruleset can only result in the target configuration $\mathcal{C}^{\mathcal{T}}$. Therefore, we can conclude that the only reachable, stable graph is $(V, E, l_G) = f(\mathcal{C}^{\mathcal{T}})$. □

3.4.3 Ruleset Execution

The goal of the ruleset execution is the reconfiguration of $\mathcal{C}^{\mathcal{I}}$ into $\mathcal{C}^{\mathcal{T}}$. In other words, given a system (G_0, Φ) we want to execute the assembly sequence $G_0 \xrightarrow{r_1} G_1 \xrightarrow{r_2} G_2 \xrightarrow{r_3} \ldots \xrightarrow{r_n} G_{stable}$, where $G_0 = f(\mathcal{C}^{\mathcal{I}})$, $G_{stable} = f(\mathcal{C}^{\mathcal{T}})$, 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. Here, $c_i \in \mathcal{C}$ is the cube represented by node $v_i \in V$. 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 Φ are applicable to any node $v_i \in V$, i.e. until a stable graph is reached. Examples of reconfiguration sequences are shown in Section 4.

3.5 Summary

In this section, we summarize the reconfiguration planning stage of our algorithm and present a complexity analysis of its components. Fig. 14 visualizes the planning process as a flow-chart.

The planning stage can be divided into three substages - preparation for path planning, the path planning itself, and the ruleset generation. The preparation stage consists of the computation of the adjacency matrices, the movable set $\mathcal{M}$, the immediate target successor set $\mathcal{R}$, the overlapping set $\mathcal{O}$, and the assignment $a = \{c_{init}, c_{target}\}$. Since a configuration is represented as cubeset $\mathcal{C}$, the first step is rewriting it into adjacency matrices. The necessary cubesets $\mathcal{M}$, $\mathcal{R}$, and $\mathcal{O}$ as well as the assignment a are then computed. The computation of $\mathcal{M}$, $\mathcal{R}$, and the adjacency matrices feature a time complexity of $O(N^2)$ while the overlapping region $\mathcal{O}$ and the assignment a can be computed in $O(N)$ time. The path planning stage consists of the computation of the planning space $\mathcal{P}(\mathcal{C})$ and the actual path planning, both of which feature a time complexity of $O(N^2)$. The final stage of the reconfiguration planning process is the ruleset generation. After each path has been calculated it is rewritten into a ruleset. The rule generation features a time complexity of $O(pN)$, where N is the total number of cubes in the configuration and p is the average path length. This is a sub-quadratic complexity, since p is approximately proportional to the diameter of

the graph and not to the total number of nodes.

Fig. 14 shows one iteration of the planning process, i.e. one individual cube is moved and rules are generated for this one path. The time complexity for one iteration of the algorithm shown in Fig. 14 is $O(N^2)$ and for the complete reconfiguration process $O(N^3)$.

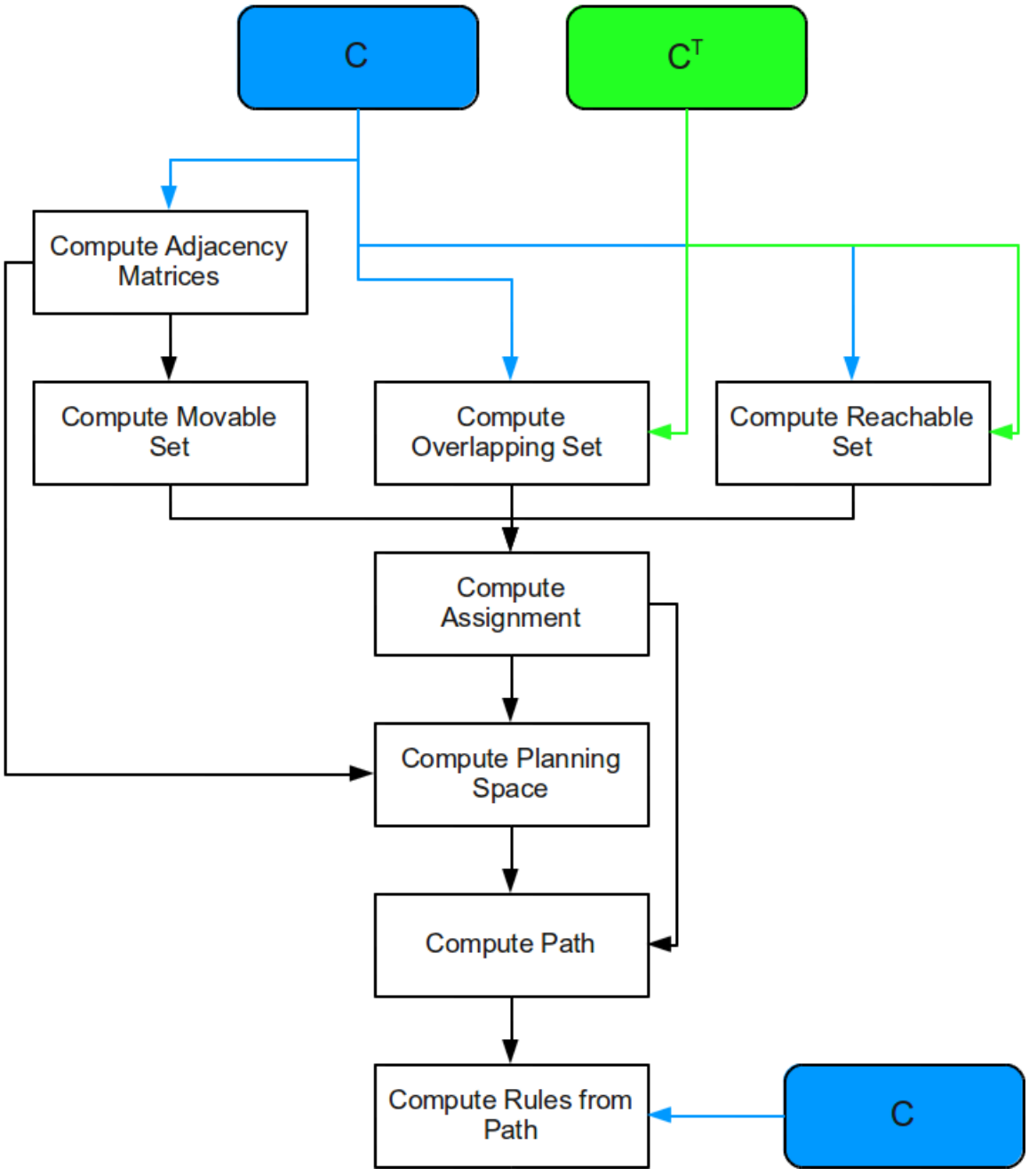


Figure 14: Flowchart describing the reconfiguration process

CHAPTER IV

SIMULATION AND EXPERIMENTS

This section presents experimental results obtained in simulation. Our test system was equipped with an Intel Core i5-540M dual core processor running at 2.53 GHz, 4GB of DDR3 memory, and an NVIDIA NVS3100M graphics chip. We used Ubuntu 11.04 as operating system and a Linux version of Matlab R2010a for the implementation and testing of our simulator. In principle, this version of Matlab does support multithreaded and multicore computation, but activates parallel computation only if the problem sizes are large enough. According to [47] this requires several thousand elements in the input arrays. The configurations we work with do not trigger Matlab's multithreading, which is why all the results presented in this section run on a single core and a single thread. We present several reconfigurations sequences and showcase the capabilities of our simulator to handle obstacle-constrained reconfiguration and ruleset switching for dynamic reconfiguration.

4.1 Self-Reconfigurable Furniture

This experiment shows the automatic reconfiguration of furniture. Using furniture composed of modular robots gives the user the possibility to change the shape of a collection of modules by issuing a single command and uploading a new ruleset. As an example of self-reconfigurable furniture, we present the reconfiguration of an office chair into a table. No locomotion is achieved during the reconfiguration process. Such reconfiguration on the spot without any locomotion is also referred to as static reconfiguration (see [12]). We will refer to the initial chair configuration as $\mathcal{C}^{\mathcal{I}}$ and the table configuration as $\mathcal{C}^{\mathcal{T}}$.

Both $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ are composed of 127 modules with 12 overlapping modules. The configurations were manually generated to fulfill two requirements. First, the number of cubes in both configurations has to be the same and second, the configuration has to be connected. The parameters of the configurations and results of the reconfiguration planning are summarized in Table 1. The data given in the table are the configuration size of both the inital and the target configuration $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ and the initial overlap of $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$. The average diameter represents an approximation of the graph diameter, which is defined as the "longest shortest path between any two vertices". More accurately the diameter of a graph G is defined as $\max_{v_i,v_j \in G} d(v_i, v_j)$, where $d(v_i, v_j)$ is the minimum length path between vertex v_i and v_j. Since the computation of the graph diameter requires the expensive computation of a path between any two vertices v_i and v_j of the graph G, we approximated the diameter as follows.

$$diam(G) \approx \max_{c_i,c_j \in \mathcal{C}} dist(c_i, c_j)$$

With this definition, we estimate the diameter of the graph G as the distance between the two cubes of the represented configuration $\mathcal{C}$ that are furthest apart. This is a coarse numeric representation of the shape of the configuration and an indicator of the average path length. We used the Manhattan distance as distance metric and reduced the complexity of computing the graph diameter from planning a path between any two vertices $v_i, v_j \in V$ to computing the Manhattan distance between $c_i, c_j \in \mathcal{C}$. In the experiments conducted in this Section, we have found that the average path length is correlated to the diameter approximation. Other data in Table 1 include the average path length of all calculated paths, the total path length as the sum of the length of all paths, the total number of generated rules, and the required time for the completion of the reconfiguration planning stage. The reconfiguration of 127 cubes from their initial chair configuration to the target table configuration took approximately 6.85 minutes to plan and features an average path length of about 6.58. This means that a

Table 1: Reconfiguration planning results for the self-reconfigurable furniture scenario

	Self-reconfigurable furniture
Configuration Size	127
Initial Overlap total/percent	12 / 9.45%
Average Diameter	21.6957
Average Path Length	6.5826
Total Length of all Paths	757
Number of Generated Rules	987
Runtime [min]	6.8565

cube had to move on average 6.58 lattice positions to relocate from its initial position $c_i \in \mathcal{C}^{\mathcal{I}}$ to its target position $c_j \in \mathcal{C}^{\mathcal{T}}$. A total number of 757 primitive motions were executed during the reconfiguration and a total number of 987 rules were generated to represent this reconfiguration sequence.

Fig. 15 shows the reconfigurations sequence of this scenario. It can be seen that the greedy approach always picks the two closest nodes in the initial and the target configuration. This approach disassembles the chair while it builds the table incrementally and creates a flow-like behavior. Both configurations feature an overlap that is not necessarily optimal in a sense of a maximal number of overlapping cubes. An optimization approach based on gradient descent or simulated annealing could translate $\mathcal{C}^{\mathcal{T}}$ with respect to $\mathcal{C}^{\mathcal{I}}$ to increase the number of overlapping cubes and thus decrease the number of planned paths and planning time.

4.2 Bucket of Stuff

This experiment demonstrates how a random two-dimensional configuration of cubes can self-assemble into a piece of furniture, for example an office chair. [65] calls this scenario *Bucket of Stuff*, where a number of modules are tossed onto the floor and self-assemble into the desired target shape. This reconfiguration scenario can be used

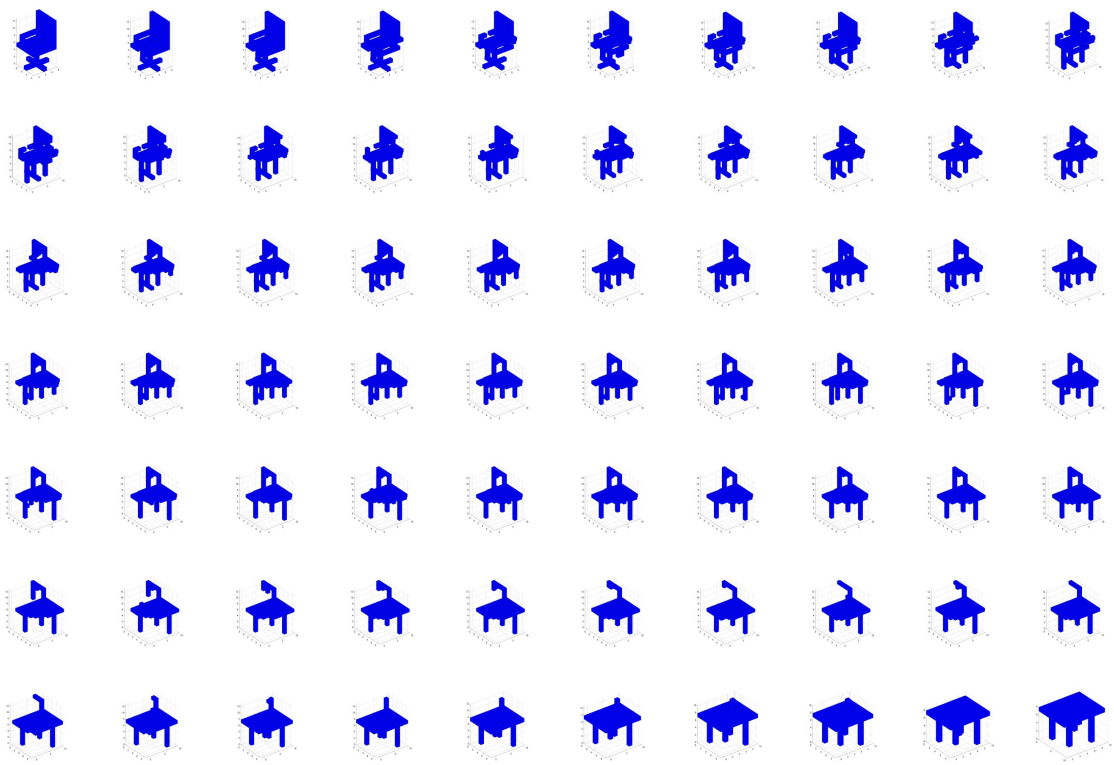


Figure 15: Reconfiguration sequence from a chair configuration to a table configuration in free space

for self-assembling furniture, tools, or on a larger scale even vehicles and buildings.

Both $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ are composed of 127 modules with 7 initially overlapping cubes. The initial configuration was randomly generated, while the target configuration was manually designed. Both configurations consist of the same number of cubes and are connected. The parameters of the configurations and results of the reconfiguration planning are summarized in Table 2. The parameters are similar to those in Section 4.1. The number of initially overlapping cubes is slightly smaller than in Section 4.1 and the average diameter of the configuration increased about 36% compared to the scenario shown in Section 4.1. Yet the average path length, total path length, and ruleset size nearly doubled. This behavior can be attributed to the greedy assignment approach, where cubes close to their target position are relocated first. Greedy assignment creates tree-like arrangement of cubes (see Fig. 16) where farther cubes are moved later during the reconfiguration. These paths are significantly longer and dominate the average path length. Additionally, $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$ exhibit less geometric similarity than in Section 4.1 because we reconfigure a two-dimensional configuration into a three-dimensional instead of reconfiguring a three-dimensional configuration into a three-dimensional.

Fig. 16 shows the reconfigurations sequence of this scenario. Note that in this experiment, the greedy assignment approach creates tree-like arrangements of cubes emanating from the base of the chair creating the impression that the cubes flow to and up the base of the chair. As can be seen in Fig. 16, the chair is built incrementally bottom-up as a result of the greedy assignment approach.

4.3 Pack and Go

Similar to the self-reconfigurable furniture scenario, this experiment shows how the result of reconfiguring one configuration into another can serve completely different

Table 2: Reconfiguration planning results for the "Bucket of Stuff" scenario

	Bucket of Stuff
Configuration Size	127
Initial Overlap total/percent	7 / 5.51%
Average Diameter	29.5583
Average Path Length	14.35
Total Length of all Paths	1722
Number of Generated Rules	1952
Runtime [min]	10.9557

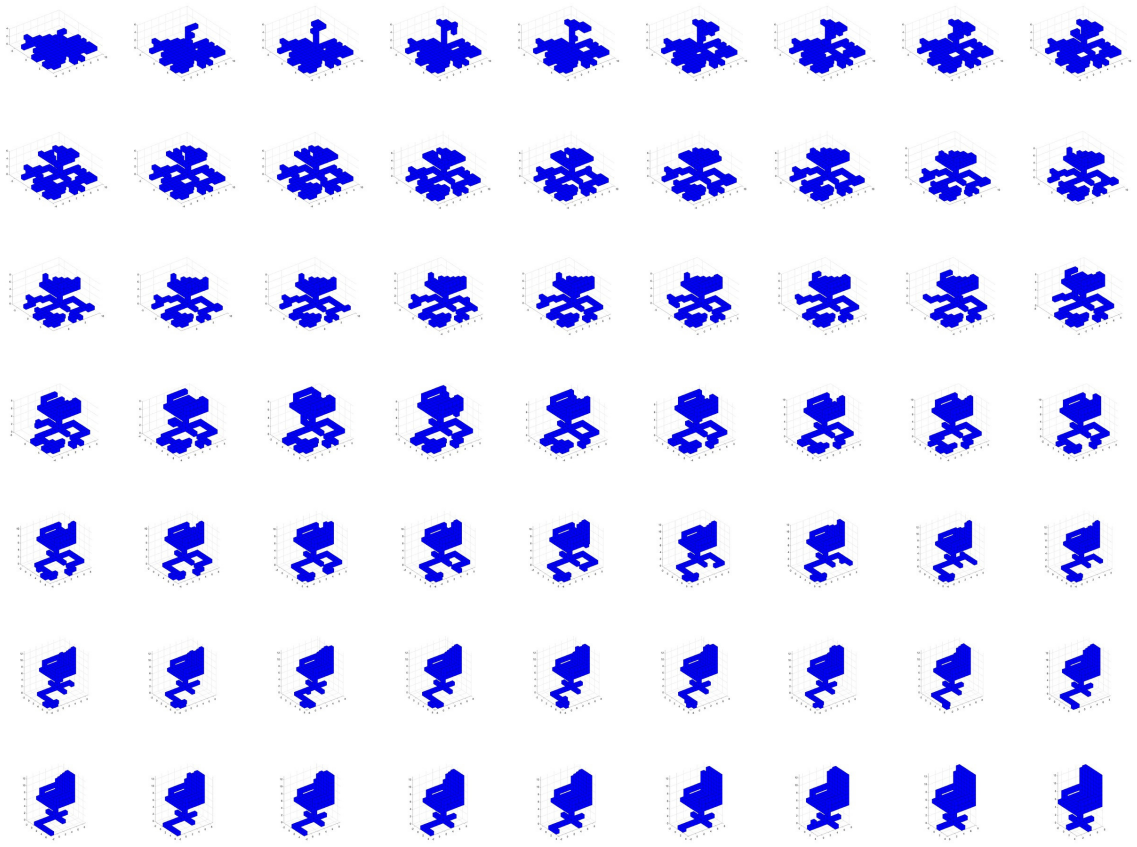


Figure 16: Reconfiguration sequence from a random two-dimensional configuration to a chair configuration

purposes. We reconfigure a house-shaped configuration $\mathcal{C}^{\mathcal{I}}$ into a truck-shaped configuration $\mathcal{C}^{\mathcal{T}}$ and demonstrate a possible use of self-reconfigurable robots in tomorrow's housing and transportation sector.

This experiment uses configurations containing 332 modules with 75 initially overlapping modules. Both configurations were manually designed such that they consist of the same number of cubes and are connected. The parameters of the configurations and results of the reconfiguration planning are summarized in Table 3. Even though the percentage of initially overlapping cubes is higher than in Table 1 and Table 2, the average path length is lower, and the total ruleset size is smaller, the runtime is significantly higher than in both experiments shown in Section 4.1 and Section 4.2. This can be explained by the higher number of paths that have to be calculated. Even though the average path length and time required for path planning is lower, the preparation stage for path planning (see Section 3.5) still has to be completed for every path. In this case, the total runtime is not dominated by the path planning, but by the preparation stage for path planning.

Fig. 17 shows the reconfiguration sequence of this scenario. In this experiment the overlapping region features a higher number of cubes, since the initial and the target configuration align better. Our assignment approach gives preference to positions in $\mathcal{C}^{\mathcal{T}}$ that are close to $\mathcal{C}^{\mathcal{I}}$, which can be seen in Fig. 17. The truck's cabin is assembled before all farther extension parts of the truck, such as the exhausts, the wheels, or the truck bed, are.

4.4 Locomotion

This experiment shows the reuse of rulesets, which is a requirement for locomotion. Locomotion or the use of self-reconfiguration for the purpose of moving a structure in space is also referred to as dynamic reconfiguration (see [12]). One ruleset is computed for the first reconfiguration step and then reused for consequent reconfiguration steps,

Table 3: Reconfiguration planning results for the "Pack and Go" scenario

	Pack and Go
Configuration Size	332
Initial Overlap total/percent	75 / 22.59%
Average Diameter	22.9105
Average Path Length	5.7432
Total Length of all Paths	1476
Number of Generated Rules	1733
Runtime [min]	73.8896

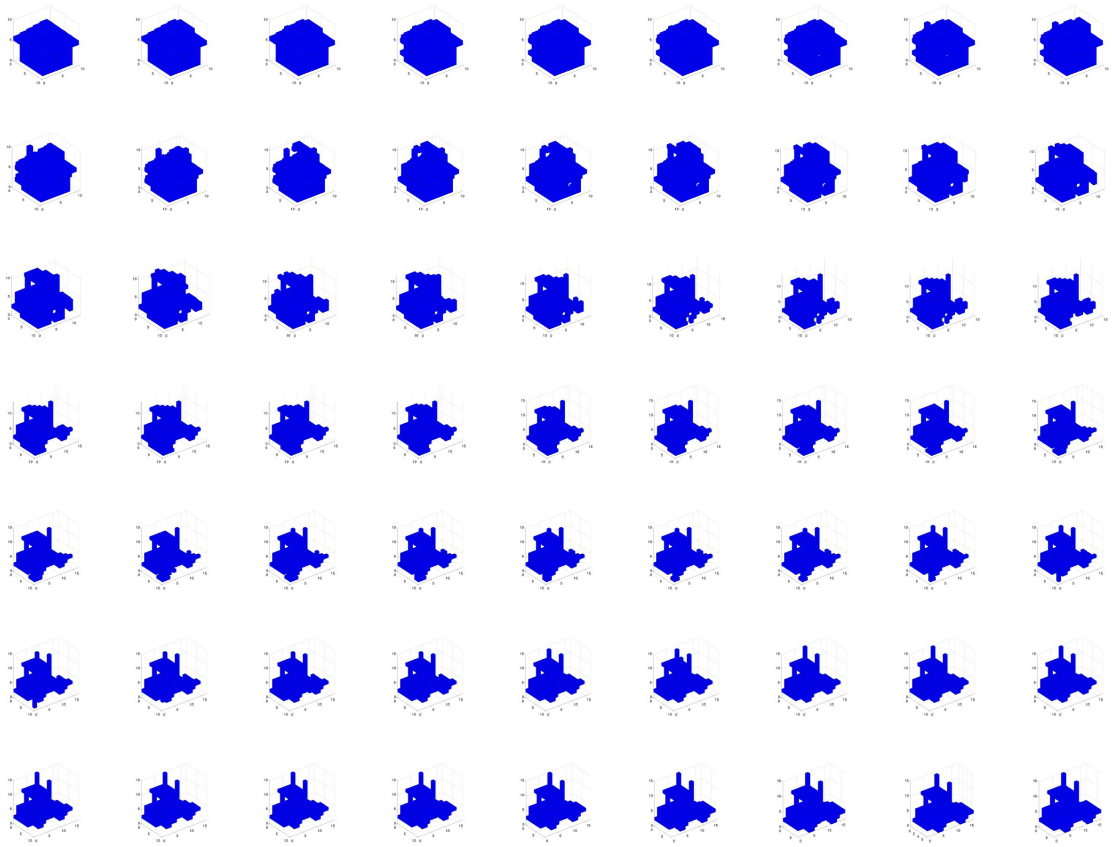


Figure 17: Reconfiguration sequence from a house configuration to a truck configuration

Table 4: Reconfiguration planning results for the locomotion scenario

	Locomotion
Configuration Size	125
Initial Overlap total/percent	25 / 20%
Average Diameter	14.2300
Average Path Length	6.0200
Total Length of all Paths	602
Number of Generated Rules	802
Runtime [min]	4.1411

thus reducing the planning effort. After every reconfiguration step, the labels of all cubes $c_i \in \mathcal{C}^{\mathcal{T}}$ are reset to their initial labels, which transforms $\mathcal{C}^{\mathcal{T}}$ into a spatially shifted version of $\mathcal{C}^{\mathcal{I}}$, denoted as ${\mathcal{C}^{\mathcal{I}}}_{shifted}$. Therefore, the same rules are applicable to ${\mathcal{C}^{\mathcal{I}}}_{shifted}$ as were applicable to $\mathcal{C}^{\mathcal{I}}$. The reason for this is the fact that rules only contain relative information, i.e. information about the the neighborhood of a cube, and not any absolute position information of cubes and their neighbors.

Both configurations in this experiment are cubic configurations containing 125 individual cubes. 25 cubes initially overlap for each reconfiguration step, i.e. are in $\mathcal{C}^{\mathcal{I}} \cap {\mathcal{C}^{\mathcal{I}}}_{shifted}$ and ${\mathcal{C}^{\mathcal{I}}}_{shifted} \cap \mathcal{C}^{\mathcal{T}}$, respectively. The parameters of the configurations and the results of a single step in the reconfiguration planning are summarized in Table 3. Note that the ruleset only has to be created once and can be reused for arbitrarily many reconfiguration steps. The results are similar to those in Section 4.1 with respect to configuration size and runtime. The average diameter is slightly smaller (14.23 compared to 21.69), which results in a lower average path length (6.02 compared to 6.58) and ruleset size (802 compared to 987). Fig. 18 shows two reconfiguration steps with the initial and the final target configuration shown as wireframes.

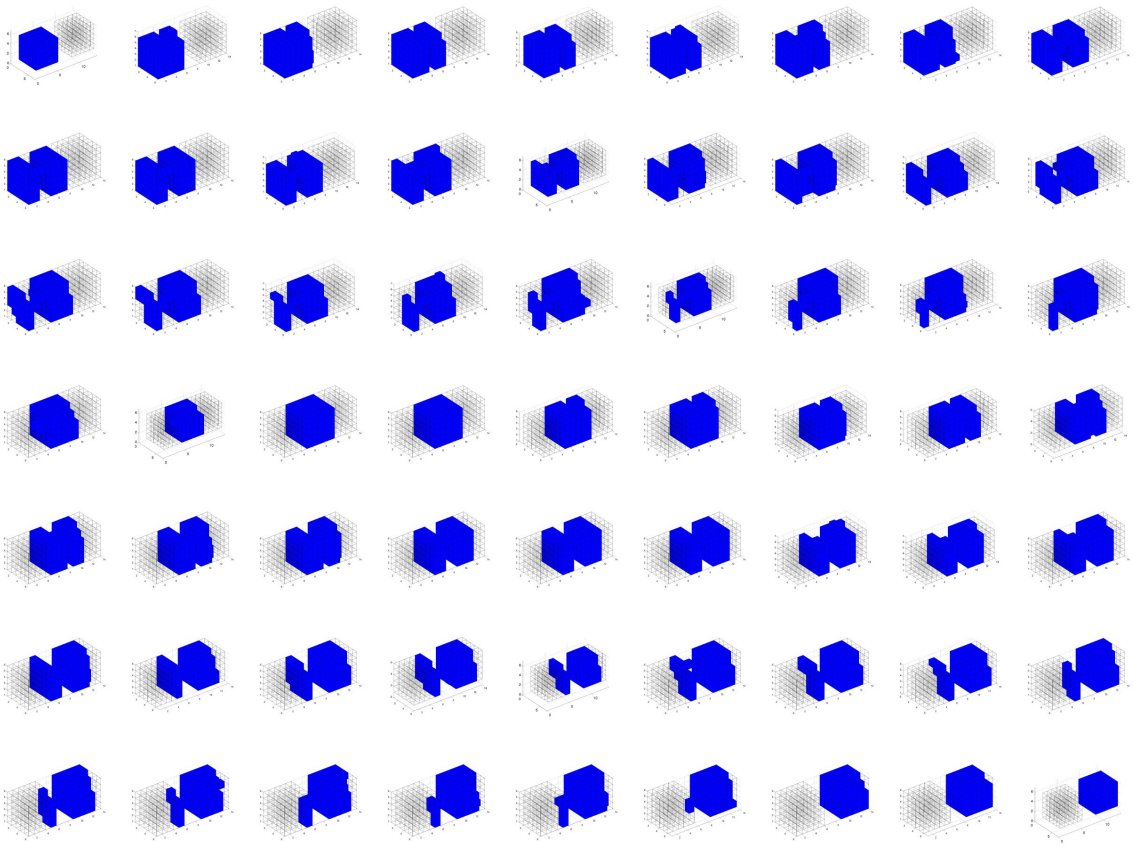


Figure 18: Reconfiguration sequence for locomotion

Table 5: Reconfiguration planning results for the obstacle-constrained scenario.

	Obstacle-constrained
Configuration Size	127
Obstacle Size	225
Initial Overlap total/percent	12 / 9.45%
Average Diameter	21.6957
Average Path Length	6.5826
Total Length of all Paths	757
Number of Generated Rules	987
Runtime [min]	7.1952

4.5 Reconfiguration in Obstacle-Constrained Space

This experiment shows the reconfiguration of a chair configuration into a table configuration in an obstacle-constrained space. More precisely, the obstacles in this reconfiguration sequence represent the ground plane. Obstacles are shown as black cubes and constrain the planning space of the reconfiguration. The overhead for the inclusion of obstacles into the reconfiguration planning is negligible, since it is done during the calculation of the planning space $\mathcal{P}(\mathcal{C})$ and features a time complexity of $O(N)$. Any cube position $c_j \in C_{obstacles}$ is removed from $\mathcal{P}(\mathcal{C})$ and will not be used for path planning.

The parameters of the configurations and results of the reconfiguration planning are summarized in Table 5 and are equivalent to those in Section 4.1. The key difference is the inclusion of the obstacle set, which increases the runtime by only 4.93% of the runtime given in Table 1. Fig. 19 shows the reconfiguration sequence of this scenario. Both the initial chair configuration $\mathcal{C}^{\mathcal{I}}$ and the target table configuration $\mathcal{C}^{\mathcal{T}}$ contain 127 modules, which are represented by blue cubes. The obstacle cubeset contains 225 modules and is represented by black cubes.

Figure 19: Reconfiguration sequence from a chair configuration to a table configuration in obstacle-constrained space. Obstacles are shown in black.

Table 6: Reconfiguration planning results for the dynamic reconfiguration scenario

	Random/Box	**Box/Line**	**Line/Target**
Configuration Size	125	125	125
Initial Overlap total/percent	1 / 0.8%	25 / 20%	11 / 8.8%
Average Diameter	17.9758	22.4500	31.8596
Average Path Length	7.4758	13.6	16.3333
Total Length of all Paths	927	1360	1862
Number of Generated Rules	1175	1560	2090
Runtime [min]	5.7488	5.2081	9.2103

4.6 Ruleset switching for dynamic reconfiguration

The term *dynamic reconfiguration* refers to a self-reconfiguration sequence that changes shape and uses the shape changing ability to move in space (see [12]). In other words, self-reconfiguration and locomotion are achieved at the same time or sequentially. This experiment shows how multiple rulesets can be used to achieve multiple reconfiguration and locomotion steps by switching between rulesets. One ruleset has been computed for each of the following steps: reconfiguration of a random initial configuration $\mathcal{C}^{\mathcal{I}}$ to an intermediate box configuration $\mathcal{C}^{Int_1}$, reconfiguration of $\mathcal{C}^{Int_1}$ to a two-dimensional line configuration $\mathcal{C}^{Int_2}$, and reconfiguration of $\mathcal{C}^{Int_2}$ to the target chair configuration $\mathcal{C}^{\mathcal{T}}$. Every module can access all rulesets and can switch between them. Switching is accomplished with propagation rules similar to those notifying each module of the latest finished path. The switching to a different ruleset is activated once the last path of a reconfiguration step is completed.

Fig. 20 shows the reconfiguration sequence of this scenario. All four configurations $\mathcal{C}^{\mathcal{I}}$, $\mathcal{C}^{Int_1}$, $\mathcal{C}^{Int_2}$, and $\mathcal{C}^{\mathcal{T}}$ contain 125 modules, which are represented by blue cubes. The parameters of the configurations and results of the reconfiguration planning are summarized in Table 6. This table also shows that with an increasing average diameter, the average path length and the total ruleset size increase.

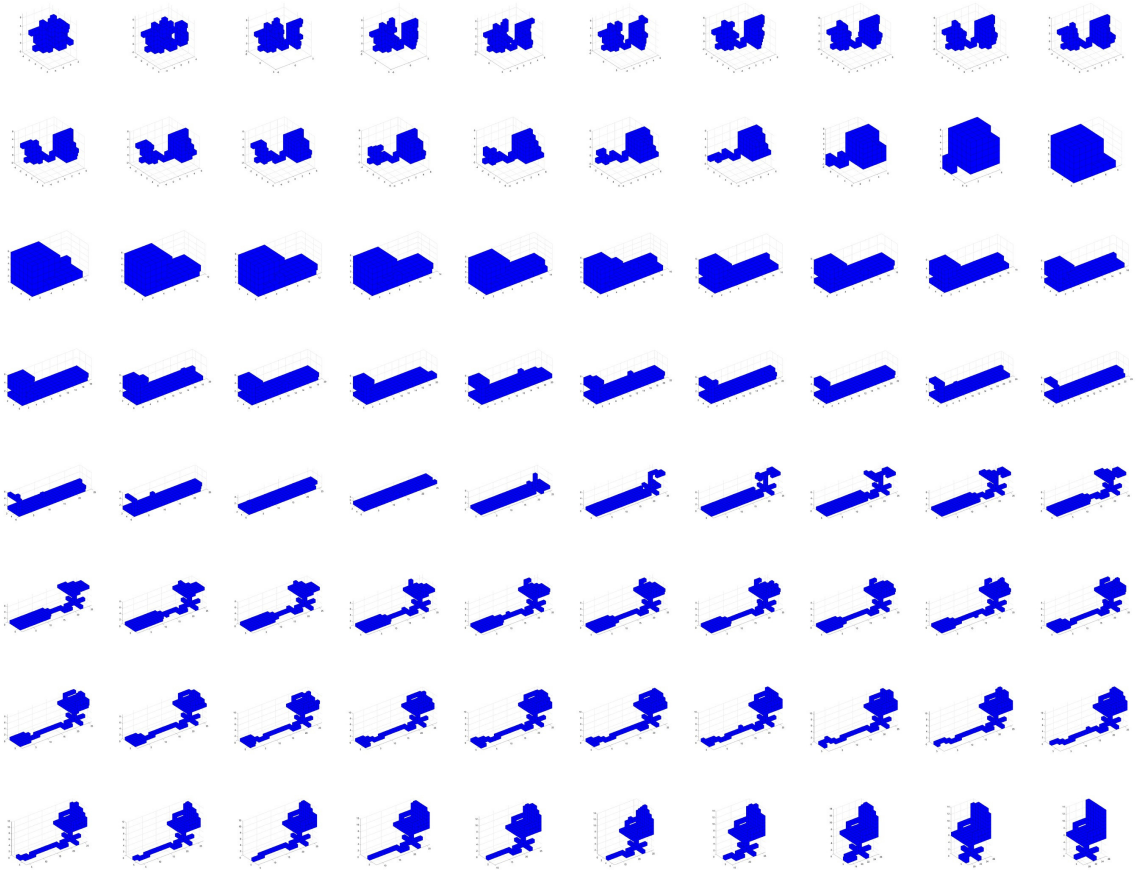


Figure 20: Reconfiguration sequence for dynamic ruleset switching

CHAPTER V

RESULTS

This section presents experimental results obtained with our simulator. We will show results for both the planning stage and the rule execution stage. As in Section 4, our test system was equipped with an Intel Core i5-540M dual core processor clocked at 2.53 GHz and 4GB of DDR3 memory. The test system was running Linux Ubunutu 11.04 as the operating system and Matlab version 2010a. All the presented execution times were measured on this system. Whereas the results from the centralized planning stage are representative of our approach, the timing results for the decentralized execution stage do not fully showcase the strengths of our approach. This is because graph grammars are inherently distributed tools that need to be executed in parallel to unfold their full potential. To be more precise, the generated rulesets have to be checked for applicable rules by every individual cube in parallel. Since we simulate these cubes on a conventional single processor machine this parallelism can not be exploited. Our algorithms check for applicable rules for every cube sequentially, which is why the runtime even for small configurations is high (see Section 5.2).

5.1 Planning Results

We conducted three series of experiments, namely the reconfiguration of box-shaped configurations of various sizes, the reconfiguration of randomly generated configurations of various sizes, and the reconfiguration of randomly generated configurations into box-shaped configurations of various sizes. The first two experiments show the scalability of our algorithm with coarse datasets whereas data collected from our third experiment is the basis for a detailed analysis of our algorithms.

5.1.1 Box and random reconfiguration

Two experiments were conducted, namely the reconfiguration of two overlapping configurations in the form of rectangular prisms (see Table 7) and the reconfiguration of two overlapping random configurations (see Table 8). The configurations ranged in size from 100 to 500 cubes in increments of 100 cubes. The runtimes are shown in Table 7 and Table 8, respectively.

In these tables, the field *Size* refers to the number of modules in the configuration, *Planned Paths* is the total number of planned paths, *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 rules in the ruleset, and *Runtime* is the time it took to generate the ruleset and complete the planning stage. No obstacles were used for both experiments, i.e. the reconfiguration was done in free space. As can be seen in Table 7, Table 8, Fig. 21, and Fig. 22, the size of the ruleset increases approximately linearly with the number of nodes, while the runtime of our algorithm increases approximately cubically for both the box and the random 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 a complete reconfiguration. The experimental results shown in Fig. 21 and Fig. 22 confirm the expected time complexity of $O(N^3)$ that we derived in Section 3.5.

5.1.2 Random initial configuration to box

This section presents results obtained from reconfiguring random configurations into box-shaped configurations of sizes 20 to 500. We determined the dimensions of these three-dimensional boxes based on a prime factorization of the configuration size. This approach can potentially introduce an additional source of variation in the results,

Table 7: Reconfiguration planning results for overlapping box configurations

Size	Planned Paths	Overlap [N] / [%]	Steps	Rules	Runtime [min]
100	70	30 / 30%	837	907	3.70
200	140	60 / 30%	1543	1683	16.65
300	210	90 / 30%	2426	2636	63.26
400	280	120 / 30%	3279	3559	135.64
500	350	150 / 30%	4275	4625	246.93

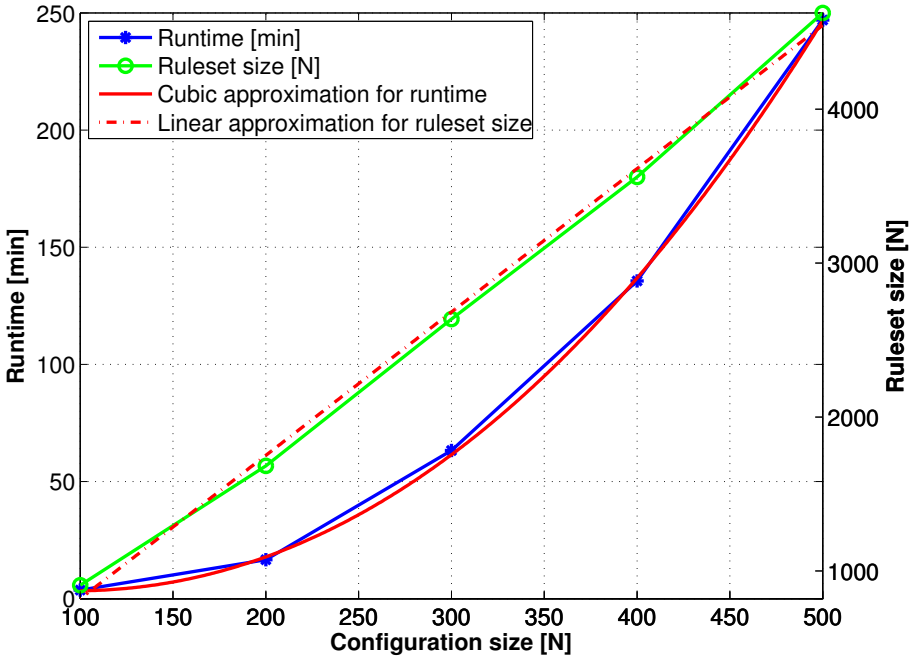


Figure 21: Number of generated rules and required runtime for ruleset generation of box configurations

Table 8: Reconfiguration planning results for overlapping random configurations

Size	Planned Paths	Overlap [N] / [%]	Steps	Rules	Runtime [min]
100	64	36 / 36%	352	416	2.25
200	86	114 / 57%	403	489	10.97
300	143	157 / 47.66%	893	1036	40.52
400	218	182 / 45.5%	1674	1890	120.84
500	269	231 / 46.2%	2327	2590	272.68

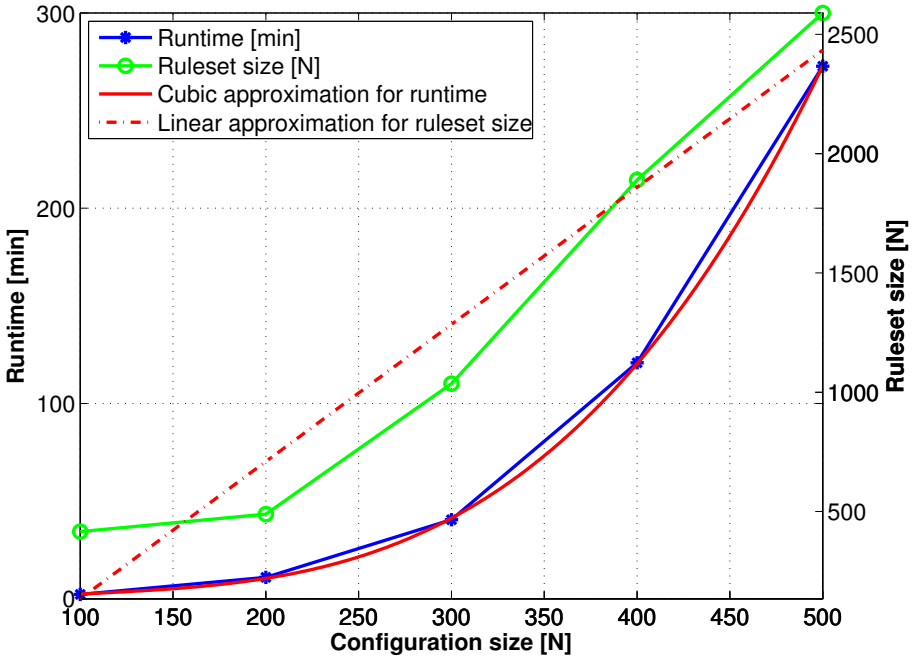


Figure 22: Number of generated rules and required runtime for ruleset generation of random configurations

since the planning effort can vary significantly based on the shape of the box. Typically, planning a reconfiguration from a random configuration to a near cubic box-shaped configuration requires a lot less planning effort than if the box was elongated along one dimension. The reason for this is that the average path length and therefore the path planning time differs greatly. This effect can be seen in Fig. 25, where spikes in the planning time and average diameter and path length are clearly visible. Fig. 26 also shows this effect.

Fig. 23 shows the results for a series of reconfigurations ranging in size from 20 to 500. Both the ruleset sizes as well as the planning times are shown. The spikes in the planning time and to a lesser extent in the ruleset size can be attributed to mainly two factors: the size of the initial overlap and the dimensions of the target box configuration. A large initial overlap reduces the number of paths that have to be planned and thus reduces planning time and total path length, which is proportional to the ruleset size. The dimensions of the box on the other hand can significantly

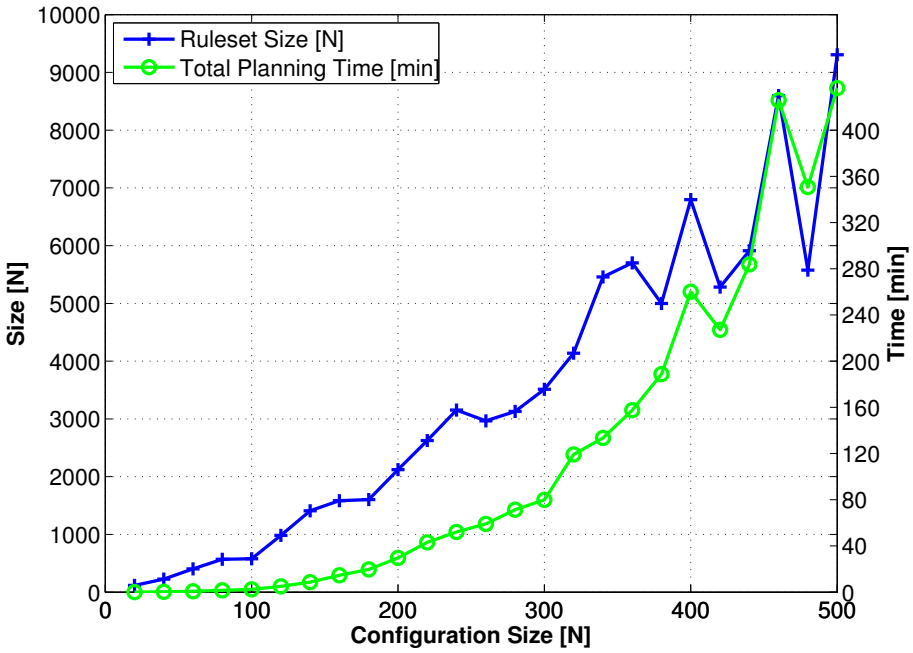


Figure 23: Ruleset size and total planning time for reconfigurations of sizes 20 to 500

increase the average and the total path length if the box is not approximately cubic but elongated along one dimension. Both factors can add their positive influence as for the reconfiguration of size 440, where planning time is significantly reduced (see Fig. 23 and Fig. 24). But it can also cancel each other out as for the reconfiguration of size 180 (see Fig. 23 and Fig. 24). The total planning time (as can be seen in Fig. 23) is correlated to the ruleset size and the total path length.

Fig. 25 shows the progression of the average path length and an approximation of the average diameter of reconfigurations of sizes 20 to 500. Additionally the average planning time for a single path for each configuration size is plotted as well. The average path length increases approximately linearly with the configuration size, as does the average diameter (see Fig. 25). As expected, the path planning time increases as the average path length increases. Generally speaking, larger configurations (both in the number of cubes and their geometrical dimensions) require longer paths to be planned. This implies longer average path planning time since A* has to expand

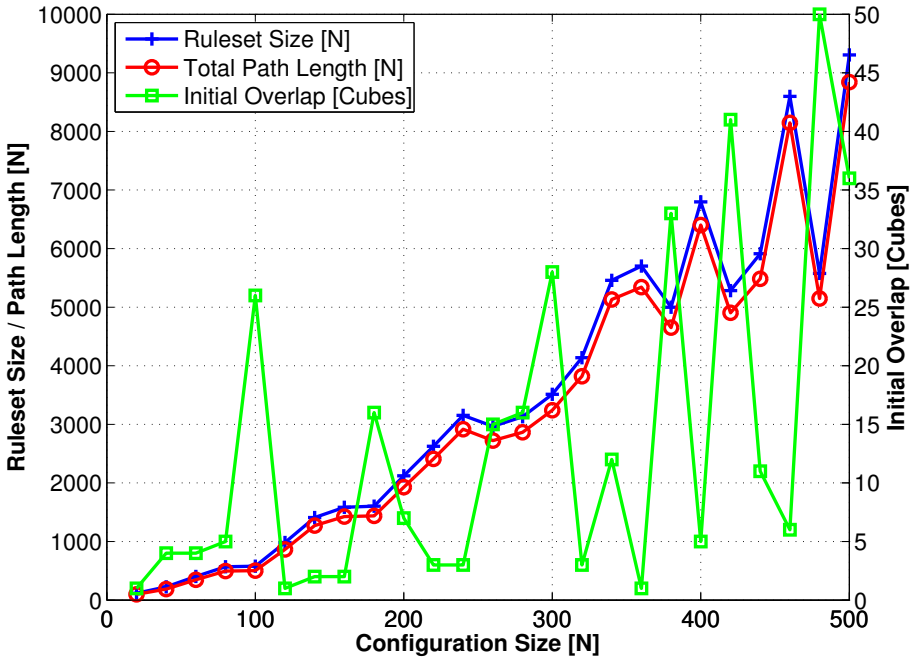


Figure 24: Ruleset size and total path length for configurations of sizes 20 to 500. The initial overlap of the initial and target configuration is shown in green.

more nodes before it finds the target node.

Fig. 26 shows the timing results from a total of 25 experiments with configurations ranging in size from 20 to 500. The primary components of our algorithm (according to Fig. 14) are shown in this plot. The time required to compute the movable set $\mathcal{M}$, the immediate target successor set $\mathcal{R}$, the planning space $\mathcal{P}(\mathcal{C})$, as well as the path planning time show similar behavior and suggest a quadratic dependency on the configuration size. This is consistent with the results achieved in Section 3.5. The time required to compute the overlapping set $\mathcal{O}$, the ruleset Φ, and the assignment show linear dependencies on the configuration size, which is also consistent with the results in Section 3.5. Note that the timing results shown in Fig. 26 are average values for the computation of a single path of a reconfiguration and do not represent the time required for an entire reconfiguration.

Fig. 27 shows the change in the size of various cubesets during a reconfiguration of size 500. Specifically, the size change of the overlapping set $\mathcal{O}$, the movable set $\mathcal{M}$, the

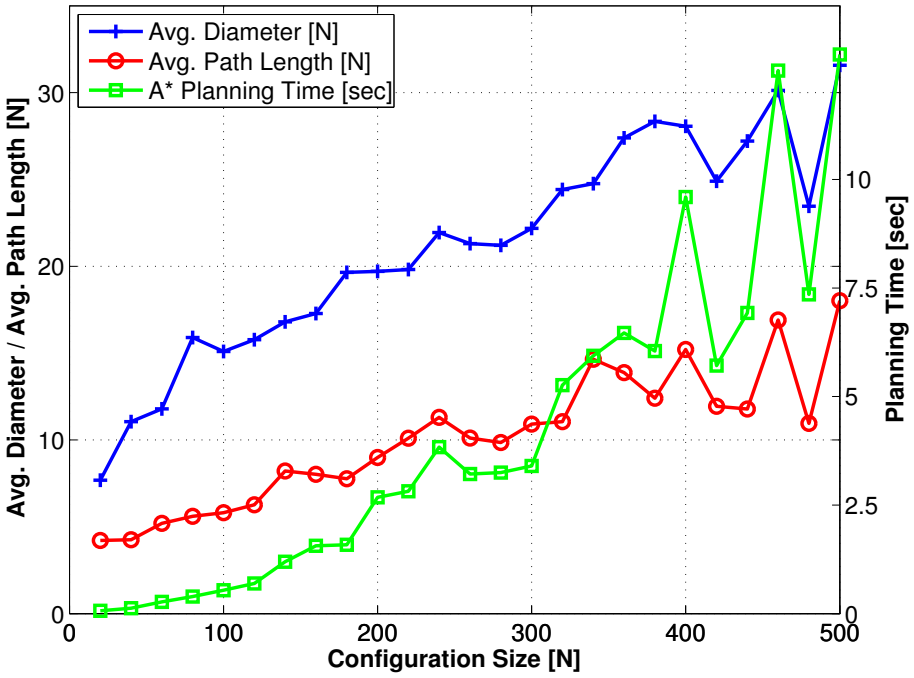


Figure 25: Average diameter, path length, and planning time for reconfigurations of sizes 20 to 500

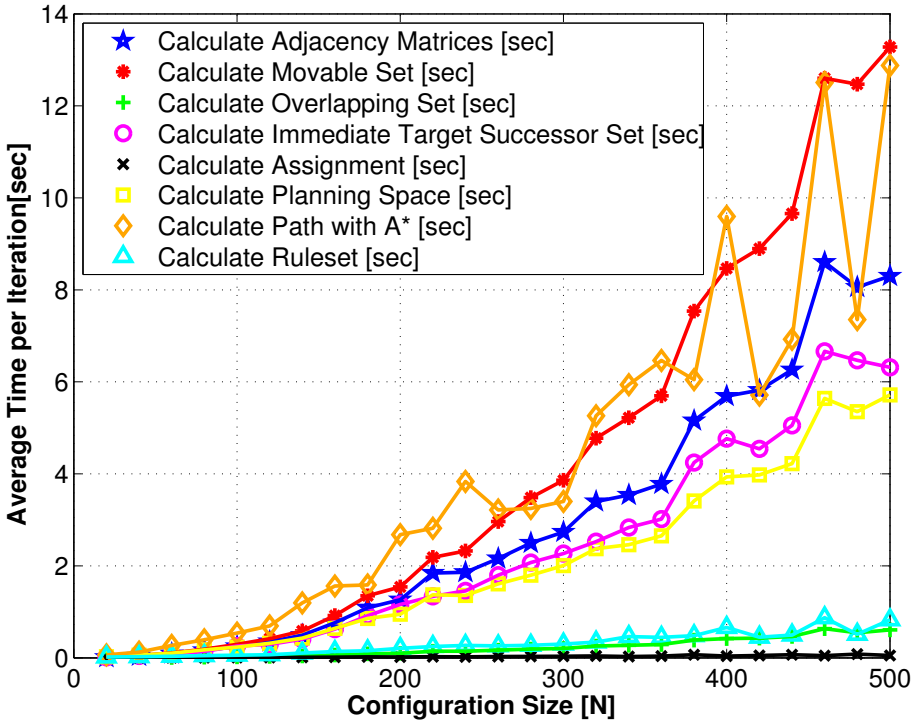


Figure 26: Timing results for reconfigurations of sizes 20 to 500

immediate target successor set $\mathcal{R}$, and the planning space $\mathcal{P}$ are shown. Additionally, the path length and an approximation of the current configuration's diameter are shown. Fig. 27 shows data of a reconfiguration of size 500, but the progression of sizes of the cubesets is consistent for all configuration sizes given similarly shaped initial and target configurations $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$. In fact, Fig. 27 shows a graphical signature of our reconfiguration approach and our choice of $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}^{\mathcal{T}}$. $\mathcal{P}$, $\mathcal{M}$, and $\mathcal{R}$ are proportional to the number of cubes on the surface of the structure. This fact is illustrated best by the progression of $\mathcal{P}$. Initially, $\mathcal{P}$ is calculated for a random configuration. As the overlapping region $\mathcal{O}$ grows linearly in size, the current configuration elongates (see for example the second reconfiguration step in Fig. 20) and the surface and therefore $\mathcal{P}$ grows. As the reconfiguration progresses and the current configuration increasingly resembles the target box configuration, the surface-to-volume-ratio drops, i.e. the number of cubes contained within the structure increases, while the number of surface cubes decreases. As a result, the size of $\mathcal{P}$ decreases as well. A similar outcome of the decreasing surface-to-volume-ratio is the decreasing size of the movable set and the immediate target successor set. The size of $\mathcal{R}$ goes to 0 as $\mathcal{C}$ becomes isomorphic to $\mathcal{C}^{\mathcal{T}}$ and all available positions $c_j \in \mathcal{C}^{\mathcal{T}}$ are filled. The size of $\mathcal{M}$ on the other hand does not, since even in the target configuration $\mathcal{C}^{\mathcal{T}}$ there are movable cubes, i.e. surface cubes. $\mathcal{O}$ shows a very typical progression that is a basic assumption in our reconfiguration approach. It grows linearly, one cube at a time, because we move cubes sequentially one at a time. In every reconfiguration step, one cube c_i is moved from $\mathcal{C}^{\mathcal{I}}$ to $\mathcal{C}^{\mathcal{T}}$. The gradual increase in the path length can be attributed to our greedy reconfiguration approach, in which the priority of relocating a cube is proportional to the cube's distance to its target position. In other words, cubes close to their respective target positions are moved first.

Fig. 28 shows the dependence of the sizes of $\mathcal{M}$, $\mathcal{O}$, $\mathcal{R}$, and $\mathcal{P}$ on the configuration size. The plot suggests approximately linear growth for $\mathcal{M}$, $\mathcal{R}$, and $\mathcal{P}$ and a random

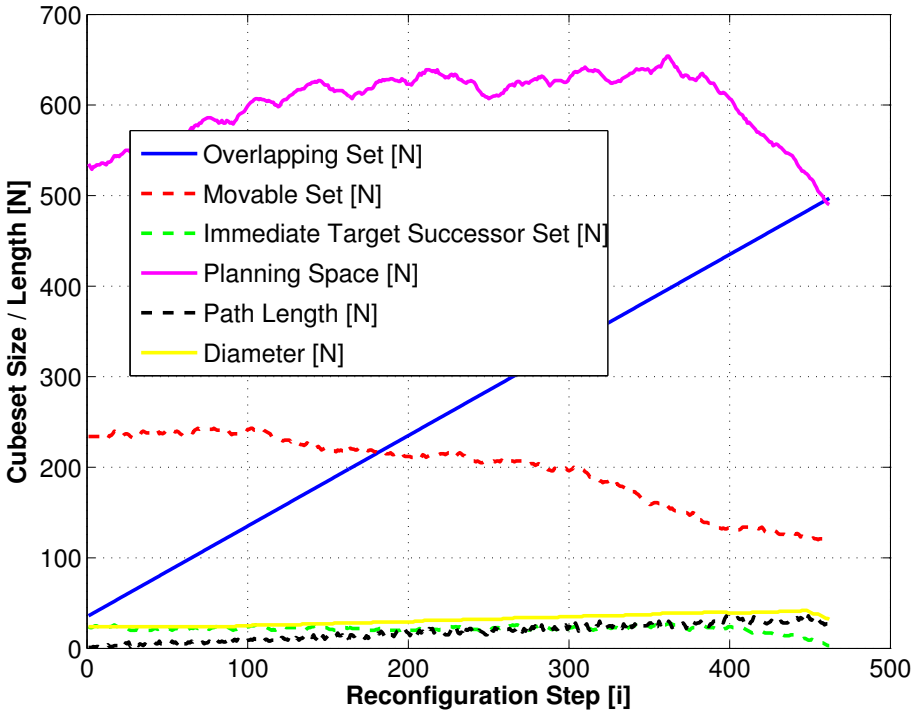


Figure 27: Cubeset sizes break down for a single reconfiguration of size 500 requiring 462 paths to be planned.

behavior for $\mathcal{O}$. This can be explained by the fact that the sizes of $\mathcal{M}$, $\mathcal{R}$, and $\mathcal{P}$ are directly proportional to the surface area of the configuration. The size of the initial overlapping set, however, depends on the relative positions of both configurations in the configuration space and the shape of those configurations. Since we have used randomly generated initial configurations for the experiments shown in this section, the size of the initial overlapping set is random as well. The spikes shown in the size progression of $\mathcal{O}$ in Fig. 28 are manifestations of that randomness.

5.2 Ruleset Execution

In this section, we present the results of the ruleset execution stage for the reconfiguration from the initial and target configuration shown in Fig. 30. Due to the high computational effort necessary to reconfigure even relatively small configurations sequentially on a single processor machine as mentioned in the introduction of this

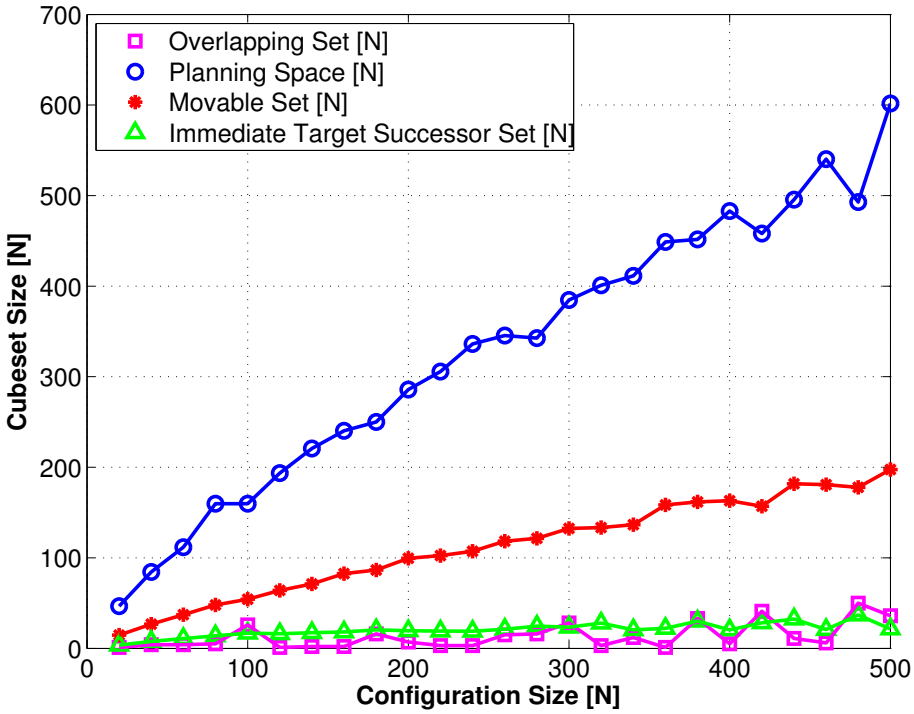


Figure 28: Cubeset sizes break down for reconfigurations of size 20 to 500. Shown are average values for each reconfiguration size.

chapter, we show only one rule-based reconfiguration.

In the current execution scheme, the algorithm checks for applicable rules sequentially, i.e. after the rule applicability check for cube c_i is completed, the algorithm proceeds to check for rules for cube c_{i+1}. If an applicable motion rule is found for cube c_i, c_i checks the ruleset again for additional applicable motion rules until no more applicable motion rules are found for and c_i's path is therefore completed. If an applicable propagation rule is found for cube c_i, the rule is applied and all the labels of c_i's neighbors are updated. Multiple cycles of ruleset checks are required to propagate a certain propagation rule through the entire configuration. Also, multiple propagation rules can be active throughout the configuration at the same time. This is shown in Fig. 29, where different active propagation rules are shown by cubes colored in blue an red. Each alternating color shows a different active propagation rule. Even though different propagation rules are active at the same time throughout the

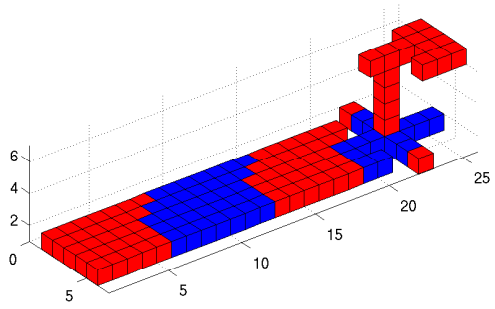


Figure 29: Multiple propagation rules being active simultaneously in the current configuration.

configuration, there is still just one single rule applicable to any cube at any given time. Therefore, the rule-based reconfiguration is unambiguous and deterministic. Unlike our work, [6] and [32] allow multiple rules to be applicable to the modules of their configurations at the same time. The outcome of their algorithms is therefore not uniquely determined.

As mentioned earlier, the simulation of the distributed ruleset execution stage happens sequentially. So even though graph grammars are designed in such a way that every cube can check the ruleset for applicable rules and execute them in parallel, the architecture of the test system lacks the necessary parallelism. A significant speedup can be achieved for the parallel execution. Assuming that just one cube moves at a time, the most significant speedup could be achieved for the propagation of information through the configuration. In our simulations, the time required for propagation is proportional to the configuration size. After a cube c_i executes a propagation rule and updates all its neighbors, the ruleset check is done for all other cubes $c_j \in \mathcal{C}$ where $j > i$. If any of c_i's neighbors fulfills this requirement the propagation proceeds with cubes c_j, otherwise the ruleset check cycles through all cubes and restarts at $i = 0$.

In the worst case, i.e. for a line configuration, each execution of a propagation rule

requires every cube to check the ruleset once. In the best case, i.e. a spheric configuration with the propagating cube being in the center, the information can be propagated in every direction at the same time making the propagation time roughly proportional to half the diameter of the configuration.

Generally speaking, the number of executed propagation rules can be computed as follows. Given the number of calculated paths p and the number of cubes c_i in the configuration N, the number of executed propagation rules is upper bounded by pN. This stems from the fact that for every completed path, every cube c_i in the configuration has to execute a propagation rule until the whole configuration is updated. The total number of executed motion rules is equal to the total path length. The results of the reconfiguration planning and ruleset execution stage for the reconfiguration shown in Fig. 30 are shown in Table 9. Note that the planning stage is completed in 9.21 minutes whereas the sequential ruleset execution on our test system took 33.31 hours to complete. Of those 33.31 hours, just a total of 1.11 minutes are spent for the execution of motion and 4.31 minutes for the execution of propagation rules or 0.06 % and 0.22 % of the total ruleset execution time, respectively. The remaining time is spent for checking the ruleset for applicable rules for every cube $c_i \in \mathcal{C}$. A total of 1951 ruleset checks are done for each cube $c_i \in \mathcal{C}$ or almost one check per cube per rule. These results suggest that the ruleset execution stage could be significantly sped up by a parallelization of the ruleset checking.

For parallel execution, the time required would be proportional to the size of the ruleset and the time each cube takes to check the ruleset for applicable rules. Assuming the time required to check the ruleset is negligible, the propagation through the whole configuration would be executed near instantaneously. In this case, the reconfiguration time would be determined by the time required to execute the motion rules.

Table 9: Reconfiguration results from the ruleset execution stage of the reconfiguration shown in Fig. 30

	Ruleset Execution
Configuration Size	125
Initial Overlap total/percent	11 / 8.8%
Number of Calculated Paths	114
Average Diameter	31.8596
Average Path Length	16.3333
Total Length of all Paths	1862
Number of Generated Rules	2090
Runtime Planning [min]	9.2103
Number of Ruleset Checks per Node	1951
Number of Propagation Rule Executions	14375
Number of Motion Rule Executions	1862
Total Number of Executed Rules	16237
Average Propagation Execution Time [sec]	0.0180
Average Motion Execution Time [sec]	0.0359
Runtime Ruleset Execution [h]	33.3149

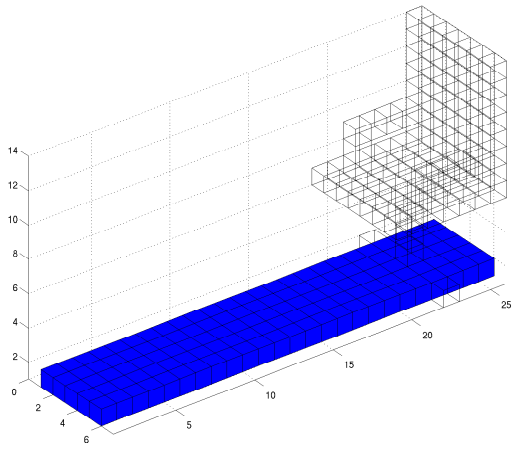


Figure 30: Initial (opaque) and target (wireframe) configuration for the ruleset execution stage

CHAPTER VI

CONCLUSION

In this thesis, we have introduced an approach to automate reconfiguration planning and to generate graph grammar-based rulesets. We have shown that our approach can reconfigure arbitrary connected configurations $\mathcal{C}^{\mathcal{I}}$ into any other arbitrary connected configuration$\mathcal{C}^{\mathcal{T}}$. We treat the reconfiguration problem as a two-stage process containing planning and execution. The centralized planning stage of our approach necessitates global knowledge of the configuration to generate the ruleset, while the decentralized execution stage works with local neighborhood information only. This is also the main advantage of our approach. While the generation of the ruleset features a time complexity of $O(N^3)$, the ruleset can be executed in a highly parallel fashion with each node checking simultaneously for applicable rules, given a hardware system with the required parallel computing capabilities. Since the size of the ruleset grows approximately linearly in N, this approach scales well.

Contrary to other approaches presented in the literature, our method of reconfiguring modular robots combines several approaches and offers multiple advantages. Our algorithms can handle arbitrary connected input configurations, can compute rulesets for both static and dynamic reconfiguration, and can automatically generate these rulesets. Rulesets can be reused for further reconfiguration steps and our system allows the switching between rulesets to enable a modular robot to perform different functions. Additionally we can guarantee that the self-reconfiguration process will reach the target configuration and provide an unambiguous and deterministic reconfiguration. Lastly, one major advantage of our graph grammar-based approach to self-reconfiguration is the straightforward extensibility to heterogeneous systems.

CHAPTER VII

OUTLOOK AND FUTURE WORK

We are currently investigating the possibility of parallely executable paths. Multiple modules of the movable set could then be moved to their respective target positions in parallel. While approaches to the parallel planning problem already exist, it still remains an open question how these parallel paths can be automatically rewritten into a ruleset and how we can still guarantee a unique reconfiguration sequence.

Currently, our algorithm generates one rule for every movement of every individual module. We plan on implementing a mechanism to reduce the number of rules by summarizing rules that describe the same movement into a single rule. This ruleset trimming mechanism could significantly reduce the number of generated rules and speed up execution.

Another way to reduce the total number of generated rules and the total path length is the improvement of the assignment algorithm. Currently we employ a greedy approach, which always picks the two closest cubes in the current configuration with respect to the target configuration. A more efficient approach would be to minimize the total distance traveled of all cubes during the reconfiguration. Solving this optimization problem would require to compute the actual path length, which is computationally very expensive. Alternatively, we could optimize with respect to a path length approximation. We assume that the Manhattan distance between two cube positions could serve as a valid approximation.

Future work also includes dealing with physical constraints such as the weight of cubes, forces exerted on cubes by gravity or the weight of other connected cubes, as well as their inertia. We also plan on including the effects of mechanical and electrical

constraints of cubes into our simulator. Interesting questions arise when we deal with the available rotational and translational power of joint motors, available battery life of the individual cubes, energy transfer between cubes, or the optimization of the reconfiguration process with respect to energy consumption.

Lastly, one major advantage of our approach is that it allows for the easy incorporation of heterogeneous modules. Heterogeneous module capabilities can be easily handled in the centralized path planning by changing the assignments of $c_i \in \mathcal{M}$ and $c_j \in \mathcal{R}$ accordingly and then encoding the various node capabilities in the labels of the rules. Therefore, our next steps includes the definition of the theoretical basis for heterogeneous reconfiguration planning and the implementation of a self-reconfiguring heterogeneous system based on graph grammars.

REFERENCES

[1] Asadpour, M., Ashtiani, M., Sproewitz, A., and Ijspeert, A., "Graph signature for self-reconfiguration planning of modules with symmetry," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pp. 5295 –5300, Oct. 2009.

[2] Asadpour, M., Sproewitz, A., Billard, A., Dillenbourg, P., and Ijspeert, A., "Graph signature for self-reconfiguration planning," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 863 –869, Sept. 2008.

[3] Baillie, J., Demaille, A., Hocquet, Q., Nottale, M., and Tardieu, S., "The urbi universal platform for robotics," *Workshop Proceedings of SIMPAR 2008*, pp. 580–591, 2008.

[4] Belta, C., Bicchi, A., Egerstedt, M., Frazzoli, E., Klavins, E., and Pappas, G., "Symbolic planning and control of robot motion [grand challenges of robotics]," *Robotics Automation Magazine, IEEE*, vol. 14, pp. 61 –70, march 2007.

[5] Bhat, P., Kuffner, J., Goldstein, S., and Srinivasa, S., "Hierarchical motion planning for self-reconfigurable modular robots," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pp. 886 –891, oct. 2006.

[6] Bishop, J., Burden, S., Klavins, E., Kreisberg, R., Malone, W., and Nguyen, T., "Programmable parts: A demonstration of the grammatical approach to self-organization," in *In Proc. of the 2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pp. 2644–2651, IEEE Computer Society Press, 2005.

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

[8] Brandt, D. and Ostergaard, E. H., "Behaviour subdivision and generalization of rules in rule-based control of the atron self-reconfigurable robot," 2004.

[9] Butler, Z., Murata, S., and Rus, D., "Distributed replication algorithms for self-reconfiguring modular robots," *Proceedings of DARS*, 2002.

[10] Butler, Z., Kotay, K., Rus, D., and Tomita, K., "Cellular automata for decentralized control of self-reconfigurable robots," in *In Proc. IEEE ICRA Workshop on Modular Robots*, pp. 21–26, 2001.

[11] Butler, Z., Kotay, K., Rus, D., and Tomita, K., "Generic decentralized control for a class of self-reconfigurable robots," in *In Proc of IEEE ICRA*, pp. 809–816, 2002.

[12] Butler, Z., Kotay, K., Rus, D., and Tomita, K., "Generic decentralized control for lattice-based self-reconfigurable robots," *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 919–937, 2004.

[13] Casal, A. and B, M. Y., "Self-reconfiguration planning for a class of modular robots," 1999.

[14] Chatzigeorgiou, D., Loizou, S., and Kyriakopoulos, K., "R-cell: A module for a self-reconfigurable robotic system," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 895 –900, Sept. 2008.

[15] Christensen, D., Brandt, D., Stoy, K., and Schultz, U., "A unified simulator for self-reconfigurable robots," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 870 –876, Sept. 2008.

[16] Ehrig, H., "Introduction to the algebraic theory of graph grammars (a survey)," in *Graph-Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science, ch. 1, pp. 1–69, 1979.

[17] Fahmy, H. and Blostein, D., "A survey of graph grammars: theory and applications," in *Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on*, pp. 294 –298, Aug. 1992.

[18] Fitch, R. and Butler, Z., "Scalable locomotion for large self-reconfiguring robots," in *Robotics and Automation, 2007 IEEE International Conference on*, pp. 2248 –2253, April 2007.

[19] Fitch, R., Butler, Z., and Rus, D., "Reconfiguration planning for heterogeneous self-reconfiguring robots," in *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol. 3, pp. 2460 – 2467, Oct. 2003.

[20] Fukuda, T., Nakagawa, S., Kawauchi, Y., and Buss, M., "Self organizing robots based on cell structures - ckbot," in *Intelligent Robots, 1988., IEEE International Workshop on*, pp. 145 –150, oct-2 nov 1988.

[21] Gerkey, B. P., Vaughan, R. T., and Howard, A., "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323, 2003.

[22] Gerkey, B. P., Vaughan, R. T., Sukhatme, G. S., Stoy, K., Howard, A., and Mataric, M. J., "Most valuable player: A robot device server for distributed control," 2001.

[23] Hsu, D., Kindel, R., claude Latombe, J., and Rock, S., "Randomized kinodynamic motion planning with moving obstacles," 2000.

[24] Huang, Y. and Gupta, K., "Rrt-slam for motion planning with motion and map uncertainty for robot exploration," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 1077 –1082, sept. 2008.

[25] Jones, C. and Mataric, M. J., "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*, pp. 721–726, IEEE, 2003.

[26] Jr., J. J. K. and Lavalle, S. M., "Rrt-connect: An efficient approach to single-query path planning," in *Proc. IEEE Intl Conf. on Robotics and Automation*, pp. 995–1001, 2000.

[27] Karaman, S. and Frazzoli, E., "Incremental sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1005.0416, 2010.

[28] Karaman, S., Walter, M. R., Perez, A., Frazzoli, E., and Teller, S., "Anytime motion planning using the rrt*," in *In Proc. IEEE Intl Conference on Robotics and Automation (ICRA)*, 2011.

[29] Klavins, E., "Automatic synthesis of controllers for distributed assembly and formation forming," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 3, pp. 3296 –3302, 2002.

[30] Klavins, E., "Self-assembly from the point of view of its pieces," *American Control Conference*, 2006.

[31] Klavins, E., "Programmable self-assembly," *Control Systems, IEEE*, vol. 27, pp. 43 –56, Aug. 2007.

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

[33] Klavins, E., "Universal self-replication using graph grammars," *MEMS, NANO, and Smart Systems, International Conference on*, vol. 0, pp. 198–204, 2004.

[34] Koenig, N. and Howard, A., "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[35] Koenig, S. and Likhachev., M., "D* lite.," *In Proceedings of the AAAI Conference of Artificial Intelligence (AAAI)*, pp. 476–483, 2002.

[36] Kotay, K. and Rus, D., "Algorithms for self-reconfiguring molecule motion planning," in *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, vol. 3, pp. 2184 –2193 vol.3, 2000.

[37] Kotay, K. and Rus, D., "Generic distributed assembly and repair algorithms for self-reconfiguring robots," in *International Conference on Intelligent Robots and Systems*, vol. Vol.3, pp. 2362 – 2369, 2004.

[38] Kotay, K. and Rus, D., "Efficient locomotion for a self-reconfiguring robot," in *In Proc. of Int. Conference on Robotics and Automation (ICRA*, pp. 2963–2969, 2005.

[39] Kurokawa, H., Murata, S., Yoshida, E., Tomita, K., and Kokaji, S., "A 3-d self-reconfigurable structure and experiments," in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 2, pp. 860 –865 vol.2, oct 1998.

[40] Kurokawa, H., Kamimura, A., Yoshida, E., Tomita, K., and Kokaji, S., "M-tran ii: Metamorphosis from a four-legged walker to a caterpillar," in *in Proc. 2003 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, 2003*, pp. 2454–2459, 2003.

[41] Kurokawa, H., Kamimura, A., Yoshida, E., Tomita, K., Murata, S., and Kokaji, S., "Self-reconfigurable modular robot (m-tran) and its motion design," in *In Seventh International Conference on Control, Automation, Robotics And Vision (ICARCV02)*, pp. 51–56, 2002.

[42] Kurokawa, H., Tomita, K., Kamimura, A., Kokaji, S., Hasuo, T., and Murata, S., "Distributed self-reconfiguration of m-tran iii modular robotic system," *The International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 373–386, 2008.

[43] Larkworthy, T. and Ramamoorthy, S., "An efficient algorithm for self-reconfiguration planning in a modular robot," in *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, pp. 5139–5146, IEEE, 2010.

[44] Lavalle, S. M., "Rapidly-exploring random trees: A new tool for path planning," tech. rep., 1998.

[45] Mackay, D. and Suffield, D., "Path planning with d*lite implementation and adaptation of the d*lite algorithm," 2005.

[46] Martelli, A., "On the complexity of admissible search algorithms," *Artificial Intelligence*, vol. 8, no. 1, pp. 1 – 13, 1977.

[47] Mathworks, "Which matlab functions benefit from multithreaded computation?." Online Article, Nov. 2009.

[48] McNew, J. M. and Klavins, E., "A grammatical approach to cooperative control," 2005.

[49] McNew, J. M. and Klavins, E., "Locally interacting hybrid systems with embedded graph grammars," *45th IEEE Conference on Decision and Control*, pp. pp. 6080–87, 2006.

[50] McNew, J. M., Klavins, E., and Egerstedt, M., "Solving coverage problems with embedded graph grammars. hybrid systems: Computation and control," 2007.

[51] Mesbahi, M. and Egerstedt, M., *Graph Theoretic Methods in Multiagent Networks*. Princeton University Press, July 2010.

[52] Moeckel, R., Jaquier, C., Drapel, K., Dittrich, E., Upegui, A., and Ijspeert, A. J., "Exploring adaptive locomotion with yamor, a novel autonomous modular robot with bluetooth interface," *Industrial Robot*, vol. 33, no. 4, pp. 285–290, 2006.

[53] Muhammad, A. and Egerstedt, M., "Connectivity graphs as models of local interactions," *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 1, pp. 124 – 129 Vol.1, dec. 2004.

[54] Murata, S., Kurokawa, H., Yoshida, E., Tomita, K., and Kokaji, S., "A 3-d self-reconfigurable structure," in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, vol. 1, pp. 432 –439 vol.1, may 1998.

[55] Ostergaard, E. H. and Lund, H. H., "Distributed cluster walk for the atron self-reconfigurable robot," in *In Proceedings of the The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, (Amsterdam, Holland), pp. 291–298, March 10-13, 2004.

[56] Ostergaard, E. H., Kassow, K., Beck, R., and Lund, H. H., "Design of the atron lattice-based self-reconfigurable robot," *Autonomous Robots*, vol. 21, pp. 165–183, Sept. 2006.

[57] Prevas, K. C., Ünsal, C., Önder Efe, M., and Khosla, P. K., "A hierarchical motion planning strategy for a uniform self-reconfigurable modular robotic system," in *In Proceedings, IEEE International Conference on Robotics and Automation*, pp. 787–792, 2002.

[58] Rozenberg, G., ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.

[59] Rus, D. and Vona, M., "Crystalline robots: Self-reconfiguration with compressible unit modules," *Autonomous Robots*, vol. 10, pp. 107–124, Jan. 2001.

[60] Russell, S. J. and Norvig, P., *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[61] Shen, W.-M., Krivokon, M., Chiu, H., Everist, J., Rubenstein, M., and Venkatesh, J., "Multimode locomotion via superbot robots," in *In Proceedings, IEEE International Conference on Robotics and Automation*, pp. 2552–2557, 2006.

[62] Stoy, K., Shen, W.-M., and Will, P., "Using role-based control to produce locomotion in chain-type self-reconfigurable robots," *Mechatronics, IEEE/ASME Transactions on*, vol. 7, pp. 410 –417, dec. 2002.

[63] Tang, S., Zhu, Y., Zhao, J., and Cui, X., "The ubot modules for self-reconfigurable robot," in *Reconfigurable Mechanisms and Robots, 2009. ReMAR 2009. ASME/IFToMM International Conference on*, pp. 529 –535, June 2009.

[64] van Leeuwen, J., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.

[65] Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., and Chirikjian, G. S., "Modular self-reconfigurable robot systems – challenges and opportunities for the future," *IEEE Robotics and Autonomation Magazine*, vol. March, pp. 43–53, 2007.

[66] Yim, M., Zhang, Y., Lamping, J., and Mao, E., "Distributed control for 3d metamorphosis," *Autonomous Robots*, vol. 10, pp. 41–56, Jan. 2001.

[67] Yoshida, E., Murata, S., Kurokawa, H., Tomita, K., and Kokaji, S., "A distributed reconfiguration method for 3d homogeneous structure," in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 2, pp. 852 –859 vol.2, oct 1998.

[68] Yoshida, E., Matura, S., Kamimura, A., Tomita, K., Kurokawa, H., and Kokaji, S., "A self-reconfigurable modular robot: Reconfiguration planning and experiments," *The International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 903–915, 2002.

[69] Yoshida, E., Murata, S., Kamimura, A., Tomita, K., Kurokawa, H., and Kokaji, S., "A motion planning method for a self-reconfigurable modular robot," 2001.

[70] Zucker, M., Kuffner, J., and Branicky, M., "Multipartite rrts for rapid replanning in dynamic environments," in *in IEEE ICRA*, pp. 1603–1609, 2007.